Association for Computing Machinery

*Advancing Computing as a Science & Profession*

September 19, 2024
Vienna, Austria

# MPLR '24

**Proceedings of the 21st ACM SIGPLAN International Conference on**

# Managed Programming Languages and Runtimes

*Edited by:*
**M. Anton Ertl and Christoph M. Kirsch**

*Sponsored by:*
**ACM SIGSOFT, AITO**

*Co-located with:*
**ISSTA '24**

Association for Computing Machinery, Inc.
1601 Broadway, 10th Floor
New York, NY 10019-7434
USA

Copyright © 2024 by the Association for Computing Machinery, Inc (ACM). Permission to make digital or hard copies of portions of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted.

To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permission to republish from: Publications Dept. ACM, Inc.
Fax +1-212-869-0481 or E-mail permissions@acm.org.

For other copying of articles that carry a code at the bottom of the first or last page, copying is permitted provided that the per-copy fee indicated in the code is paid through the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, USA.

ACM ISBN: 979-8-4007-1118-3

Cover photo:
Title: " Riesenrad im Prater in Wien, Österreich"
Photographer: Manfred Werner (Tsui), 2020
License: Creative Commons Attribution-Share Alike 4.0 International
https://creativecommons.org/licenses/by-sa/4.0/deed.en
Cropped from original:
https://commons.wikimedia.org/wiki/File:Riesenrad_Wiener_Prater_2020-07-12_b.jpg

**Production:** Conference Publishing Consulting
D-94034 Passau, Germany, info@conference-publishing.com

# Welcome from the Chairs

Welcome to MPLR 2024, the 21st ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, held in Vienna, Austria on Thursday, September 19, 2024, co-located with ISSTA/ECOOP 2024. MPLR is a successor to the conference series on Managed Languages and Runtimes (ManLang, 2017 and 2018) which in turn is a successor to the conference series on Principles and Practice of Programming in Java (PPPJ, 2002 through 2016). MPLR is a premier forum for presenting and discussing novel results in all aspects of managed programming languages and runtime systems, which serve as building blocks for some of the most important computing systems around, ranging from small-scale (embedded and real-time systems) to large-scale (cloud-computing and big-data platforms) and anything in between (mobile, IoT, and wearable applications).

Our program includes a keynote talk by Ben L. Titzer, 7 full papers, and 4 shorter work-in-progress papers. These were selected from 11 full paper submissions and 1 work-in-progress submission (3 full paper submissions were selected as work-in-progress papers). All papers received at least 3 reviews from the program committee, listed below. Full papers were evaluated based on relevance, novelty, technical rigor, and contribution to the state of the art. Work-in-progress papers were reviewed based more on novelty and probable interest to the community. The program also includes 2 posters of which one is a poster version of an accepted paper. The posters received at least 2 light reviews each. Reviewing was double blind, the second time for MPLR. The review process was all online with consensus reached on each submission. We used the HotCRP management system and Conference Publishing handled the proceedings. We are grateful to the program committee for their diligence in reviewing, the quality of the reviews, and productive nature of the discussion process.

We hope that readers will find these proceedings interesting and that conference attendees will enjoy MPLR 2024, be they physically or virtually present!

**M. Anton Ertl**
General Chair
TU Wien, Austria

**Christoph M. Kirsch**
Program Chair
University of Salzburg, Austria and
Czech Technical University, Prague, Czechia

# MPLR 2024 Program Committee

| | |
|---|---|
| Shoaib Akram | Australian National University, Australia |
| Stephen M. Blackburn | Google and Australian National University, Australia |
| Daniele Bonetta | Vrije Universiteit Amsterdam, The Netherlands |
| Maxime Chevalier-Boisvert | Shopify, Canada |
| Benjamin Chung | JuliaHub, USA |
| David F. Bacon | Google, USA |
| Olivier Flückiger | Google, Switzerland |
| Michael Homer | Victoria University of Wellington, New Zealand |
| Antony Hosking | Australian National University, Australia |
| Yusuke Izawa | Tokyo Metropolitan University, Japan |
| Ranjit Jhala | University of California, San Diego, USA |
| Jeehoon Kang | KAIST, South Korea |
| Doug Lea | State University of New York (SUNY) Oswego, USA |
| Ana Milanova | Rensselaer Polytechnic Institute, USA |
| Fabian Mühlböck | Australian National University, Australia |
| Tomoki Nakamaru | University of Tokyo, Japan |
| Hila Peleg | Technion, Israel |
| Maoni Stephens | Microsoft, USA |
| Ben L. Titzer | Carnegie Mellon University, USA |
| Tomoharu Ugawa | University of Tokyo, Japan |
| Christian Wimmer | Oracle Labs, USA |
| Tobias Wrigstad | Uppsala University, Sweden |
| YungYu Zhuang | National Central University, Taiwan |

# MPLR Steering Committee

| | |
|---|---|
| Antony Hosking | Australian National University, Australia |
| Christos Kotselidis | Pierer Innovation, Austria / The Univ. of Manchester, UK |
| Stefan Marr | University of Kent, UK |
| Herbert Kuchen | University of Münster, Germany |
| Jeremy Singer | University of Glasgow, UK |
| Elisa Gonzalez Boix | Vrije Universiteit Brussel, Belgium |
| Tobias Wrigstad | Uppsala University, Sweden |
| Eliot Moss | University of Massachusetts Amherst, USA |
| Rodrigo Bruno | INESC-ID/Técnico ULisboa, Portugal |

# Contents


## Frontmatter

**Welcome from the Chairs** . . . iii

**MPLR 2024 Organization** . . . iv

## Keynote

**Can WebAssembly Be Software's Final Substrate? (Keynote)**
Ben L. Titzer — *Carnegie Mellon University, USA* . . . 1

## Optimization

**Lazy Sparse Conditional Constant Propagation in the Sea of Nodes**
Christoph Aigner, Gergö Barany, and Hanspeter Mössenböck — *JKU Linz, Austria; Oracle Labs, Austria* . . . 2

**Mutator-Driven Object Placement using Load Barriers**
Jonas Norlinder, Albert Mingkun Yang, David Black-Schaffer, and Tobias Wrigstad — *Uppsala University, Sweden; Oracle, Sweden* . 14

**Interactive Programming for Microcontrollers by Offloading Dynamic Incremental Compilation**
Fumika Mochizuki, Tetsuro Yamazaki, and Shigeru Chiba — *University of Tokyo, Japan* . . . 28

## Programming

**mruby on Resource-Constrained Low-Power Coprocessors of Embedded Devices**
Go Suzuki, Takuo Watanabe, and Sosuke Moriguchi — *Tokyo Institute of Technology, Tokyo, Japan* . . . 41

**Imagine There's No Source Code: Replay Diagnostic Location Information in Dynamic EDSL Meta-programming**
Baltasar Trancón y Widemann and Markus Lepper — *TH Brandenburg, Germany; semantics, Germany* . . . 48

**Existential Containers in Scala**
Dimitri Racordon, Eugene Flesselle, and Matthieu Bovel — *EPFL, Switzerland* . . . 55

**Quff: A Dynamically Typed Hybrid Quantum-Classical Programming Language**
Christopher John Wright, Mikel Luján, Pavlos Petoumenos, and John Goodacre — *University of Manchester, United Kingdom* . . . 65

## Analysis

**Towards Realistic Results for Instrumentation-Based Profilers for JIT-Compiled Systems**
Humphrey Burchell, Octave Larose, and Stefan Marr — *University of Kent, United Kingdom* . . . 82

**Toward Declarative Auditing of Java Software for Graceful Exception Handling**
Leo St. Amour and Eli Tilevich — *Virginia Tech, Blacksburg, USA* . . . 90

**Dynamic Possible Source Count Analysis for Data Leakage Prevention**
Eri Ogawa, Tetsuro Yamazaki, and Ryota Shioya — *University of Tokyo, Japan; IBM Research, Tokyo, Japan* . . . 98

**The Cost of Profiling in the HotSpot Virtual Machine**
Rene Mueller, Maria Carpen-Amarie, Matvii Aslandukov, and Konstantinos Tovletoglou — *Huawei Zurich Research Center, Switzerland; Kharkiv National University of Radio Electronics, Ukraine; Independent Researcher, Switzerland* . . . 112

**Author Index** . . . 127

# Can WebAssembly Be Software's Final Substrate? (Keynote)

Ben L. Titzer
Carnegie Mellon University
USA
btitzer@andrew.cmu.edu

## Abstract

Since the dawn of computing, many formats for executable programs have come and gone. The design of an executable format encounters design choices and tradeoffs such as expressiveness, ease of parsing/decoding/execution, the level of abstraction, and performance. With the advent of WebAssembly, a portable low-level compilation target for many languages, an intriguing question arises: can we finally standardize a universal binary format and software virtual machine? After many years, I believe that we finally can. Unlike language-specific bytecode formats whose abstraction level serves only one language family well, or machine-code formats that serve specific ISAs and operating systems well, WebAssembly sits between these levels of abstraction. In this talk I will share my vision for a future where all software sits on a standardized, well-specified, formally-verified substrate that allows innovation above and below, and unlocks high performance and portability for all programming languages.

**ACM Reference Format:**
Ben L. Titzer. 2024. Can WebAssembly Be Software's Final Substrate? (Keynote). In *Proceedings of the 21st ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '24), September 19, 2024, Vienna, Austria.* ACM, New York, NY, USA, 1 page. https://doi.org/10.1145/3679007.3691540

This work is licensed under a Creative Commons Attribution 4.0 International License.

*MPLR '24, September 19, 2024, Vienna, Austria*
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1118-3/24/09
https://doi.org/10.1145/3679007.3691540

# Lazy Sparse Conditional Constant Propagation in the Sea of Nodes

Christoph Aigner
Johannes Kepler University
Linz, Austria
christoph.aigner@jku.at

Gergö Barany
Oracle Labs
Vienna, Austria
gergo.barany@oracle.com

Hanspeter Mössenböck
Johannes Kepler University
Linz, Austria
hanspeter.moessenboeck@jku.at

## Abstract

Conditional constant propagation is a compiler optimization that detects and propagates constant values for expressions in the input program taking unreachable branches into account. It uses a data flow analysis that traverses the program's control flow graph to discover instructions that produce constant values.

In this paper we document our work to adapt conditional constant propagation to the Sea of Nodes program representation of GraalVM. In the Sea of Nodes, the program is represented as a graph in which most nodes 'float' and are only restricted by data flow edges. Classical data flow analysis is not possible in this setting because most operations are not ordered and not assigned to basic blocks.

We present a novel approach to data flow analysis optimized for the Sea of Nodes. The analysis starts from known constant nodes in the graph and propagates information directly along data flow edges. Most nodes in the graph can never contribute new constants and are therefore never visited, a property we call lazy iteration. Dependences on control flow are taken into account by evaluating SSA $\phi$ nodes in a particular order according to a carefully defined priority metric.

Our analysis is implemented in the GraalVM compiler. Experiments on the Renaissance benchmark suite show that lazy iteration only visits 20.5 % of all nodes in the graph. With the constants and unreachable branches found by our analysis, and previously undetected by the GraalVM compiler, we achieve an average speedup of 1.4 % over GraalVM's optimized baseline.

***CCS Concepts:*** • **Software and its engineering → Just-in-time compilers**; *Dynamic compilers*; Correctness.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*MPLR '24, September 19, 2024, Vienna, Austria*
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1118-3/24/09
https://doi.org/10.1145/3679007.3685059

***Keywords:*** data flow analysis, constant propagation, compilers, optimization, Sea of Nodes

**ACM Reference Format:**
Christoph Aigner, Gergö Barany, and Hanspeter Mössenböck. 2024. Lazy Sparse Conditional Constant Propagation in the Sea of Nodes. In *Proceedings of the 21st ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '24), September 19, 2024, Vienna, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3679007.3685059

## 1 Introduction

Constant Propagation is a compiler optimization that tries to perform as many calculations as possible at compile time, minimizing execution time by not needing to calculate those values at run time. Because detecting every compile time constant is generally an undecidable problem [5], the best one can do is to employ an algorithm with reasonable time complexity that, although not finding every possible constant, finds most constants and does so without reporting non-constant values as constant.

The state of the art in constant propagation is Wegman and Zadeck's *Sparse Conditional Constant* (SCC) algorithm [7]. It uses a data flow analysis on a program represented as a control flow graph (CFG) consisting of basic blocks of instructions in Static Single Assignment (SSA) form [3]. The analysis is *sparse* as it exploits SSA form to associate constants found with SSA variables, as opposed to earlier non-SSA algorithms which associated analysis results with pairs of variables and program points. SCC is *conditional* in that the found constants are also taken into account when evaluating branch conditions and to mark unreachable program paths which do not need to be analyzed. Skipping unreachable paths, in turn, allows finding more constants, so that this composition of simple constant propagation and unreachable code elimination is more powerful than arbitrary iterations of the separate analyses [1].

In this work we adapt sparse conditional constant propagation to the Sea of Nodes program representation in the GraalVM compiler (see Section 2.3). Graal IR is in SSA form, but most instructions 'float' rather than being assigned to specific basic blocks [4]. While SCC propagates values across the SSA data flow graph, it also needs the CFG for marking control flow edges as reachable or unreachable, and for visiting all instructions in a block. Our algorithm also propagates

values over the data flow edges of the Graal IR graph, but control flow reachability is propagated differently.

The main contributions of this work are:

- Data flow analysis using *lazy iteration*: Most nodes in the IR graph never need to be visited because they will never produce a constant, even if they can be executed. This is in contrast to Wegman and Zadeck's algorithm, which needs to visit every reachable instruction in the program.
- The *worklist priority ordering*: Optimistic data flow analysis using lazy iteration is only correct if SSA $\phi$ nodes are visited in a particular order, for which we developed a priority ordering (Section 3.5.2).

We present experimental data that shows that lazy iteration is very effective, visiting only 20.5 % of all nodes in the IR on average. It finds new constants not previously detected by the GraalVM compiler, which only features a non-optimistic version of constant propagation that does not compute optimistic fixed points over loops. The detected constants and unreachable CFG edges lead to an average speedup of 1.4 % on the standard Renaissance benchmark suite.

## 2 Background

### 2.1 Sparse Conditional Constant Propagation

The following summary of sparse conditional constant propagation is based on Wegman and Zadeck [7].

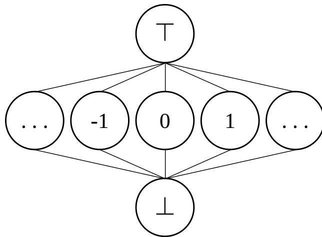



**Figure 1.** Three-tier value lattice.

SCC uses a three-tier data flow lattice shown in Figure 1. The lattice element for each variable is initially $\top$, representing a value that is yet unknown but may be determined to be a constant by the analysis. The constant elements in the middle tier represent values that will always evaluate to the given constant at runtime. The $\bot$ element denotes values for which the analysis cannot guarantee a constant value. Lattice values are combined using the meet (greatest lower bound, $\sqcap$) operation. Thus values can only be 'lowered' until a fixed point is reached.

The analysis tracks reachability of CFG edges using a two-tiered lattice, initially assuming that all edges are unreachable. At a control flow split, the analysis evaluates the split condition using its current information. Depending on whether the result is constant or not, only one or all outgoing control flow edges are marked as reachable and enqueued in a worklist for further iteration. Once a CFG edge is marked as reachable, it cannot become unreachable again.

**Listing 1.** Example of optimistic conditional constant propagation [1]. The if-branch inside the loop is never taken, and the function always returns 1.

```
public static int exampleCC(int a) {
    int x = 1;
    do {
        if (x != 1) {
            x = 2;
        }
    } while (a-- >= 1);
    return x;
}
```

Sparse conditional constant propagation is an optimistic analysis: When a control flow join point is reached but analysis information for some of the control flow predecessors is not yet available, the analysis can optimistically assume that the corresponding $\phi$ inputs are $\top$ and continue propagating information under this assumption. If the corresponding control flow paths are reachable, the algorithm guarantees that they will be traversed by the analysis at some point. If at that point the analysis provides a new value for a $\phi$ input, the optimistic assumption is invalidated, and program parts are re-analyzed with the new, lowered, information.

Optimistic analyses can discover more information than pessimistic ones, but they can only guarantee correctness if the analysis runs to completion, while pessimistic analysis can be interrupted at any time and still provide correct information [1]. Sparse conditional constant propagation is optimistic and integrates constant propagation with the analysis of unreachable code: Unreachable code paths are never analyzed and, by being associated with $\top$ data flow information, do not affect the analysis at all.

Listing 1 shows a small example adapted from [1] for a constant that cannot be found by any sequence of dead code elimination and simple constant propagation but can be found by conditional constant propagation. This is because of a cyclic dependency between data flow and control flow analysis. To detect `x` as constant, it must be known that the assignment statement `x = 2` is unreachable. To detect this statement as unreachable, it must be known that `x` is constant. This constant can only be found by first optimistically assuming `x` to be constant and subsequently verifying that the assumption was correct.

### 2.2 GraalVM

GraalVM[1] is a high-performance polyglot virtual machine. It executes programs written in Java, other languages that

[1] https://www.graalvm.org/

compile to JVM bytecode, and any other programming language implemented using GraalVM's Truffle language implementation framework. All input languages are transformed into a uniform internal representation and compiled to high-performance native code by the GraalVM compiler. GraalVM supports both just-in-time (JIT) and ahead-of-time (AOT) compilation of JVM languages.

### 2.3 Graal IR

The GraalVM compiler's intermediate representation (Graal IR) [4] is based on the *Sea of Nodes* concept [1]. This IR represents a program as a directed graph. Each node in the graph represents an operation. Edges between the nodes represent data and control dependences. Nodes are doubly-linked, i. e., from each node we can iterate efficiently both over its inputs and its usages.

Control flow edges only exist between so-called *fixed* nodes that must be strictly ordered because they have side effects (e. g., memory writes or method calls) or because they represent control flow transfers (branches, operations raising exceptions, or control flow merge points). All other nodes are *floating* nodes that are only constrained by data flow dependences. Floating nodes that are not connected via dependences are not ordered with respect to each other. Floating nodes are not assigned to any particular basic block. For final code generation and for certain optimizations, the GraalVM compiler computes a full schedule of the graph which assigns all nodes to blocks and imposes a strict order on them. Scheduling is an expensive operation, therefore most optimizations should work without a schedule.

Loops in the input program must be *reducible*, i. e., have a single loop entry. Graal IR uses a `LoopBegin` node to represent this entry point. Exits from the loop are represented as `LoopExit` nodes, and backedges are represented by `LoopEnd` nodes. Two-way control flow splits are represented as `If` nodes (multi-way `Switch` nodes are also available). The merging of control flow paths except loop backedges is represented as `Merge` nodes.

Graal IR uses SSA form, with $\phi$ nodes at loop begin and merge nodes, with one input per control flow predecessor. A $\phi$ node is a floating node but is connected to its fixed merge point by a control dependence edge. At a loop begin, the $\phi$'s first input is always the initial value on loop entry. At the point in the compilation pipeline when our constant propagation runs, the graph is in *loop-closed SSA form*: Values defined inside a loop must not be used directly outside the loop. Instead, special *proxy nodes* at loop exits mark the points where a value flows out of the loop.

Figure 2 shows a slightly simplified Graal IR generated for the method from Listing 1. In calculations, a constant input is directly shown in the node instead of connecting the node to the appropriate constant node in order to reduce the number of edges in the figure.

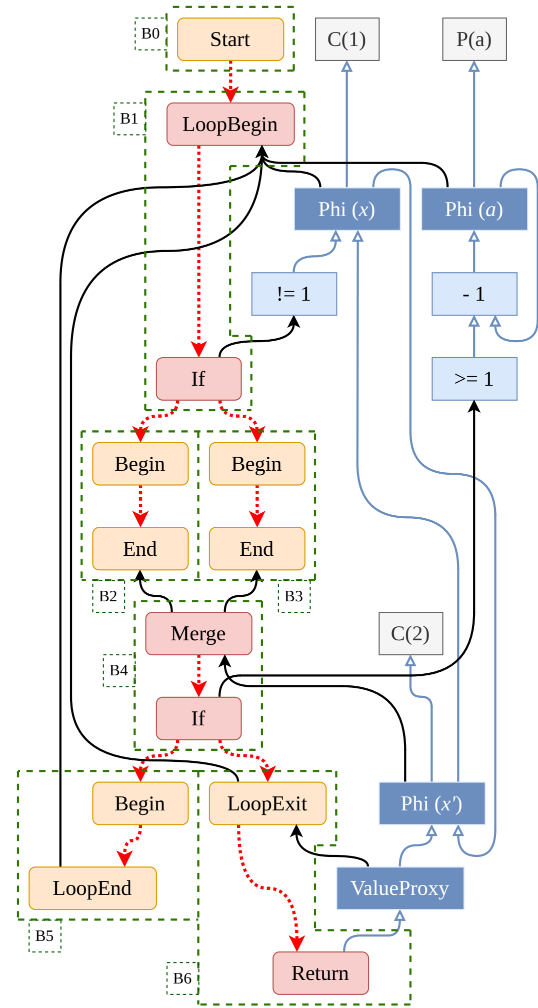



**Figure 2.** IR for the example presented in Listing 1.

A red dotted downward arrow represents a control flow successor, a black upward arrow depicts a control dependence, while a blue upward arrow with an empty head represents a data dependence. All nodes with rounded corners are fixed nodes and can therefore be directly attributed to a basic block, denoted by a dashed green outline enclosing multiple nodes. All nodes with sharp corners are floating nodes. Light gray nodes denote constants (`C`) or method parameters (`P`).

The `LoopEnd` node causes a jump back up to the `LoopBegin` node with which it is connected via a control dependence edge. The `LoopExit` node denotes control flow leaving the associated loop. `ValueProxy` nodes, which are connected with a loop exit, are inserted for values used inside the loop before any usage outside the loop.

The GraalVM compiler already performs constant propagation, but does not compute optimistic fixed points for loops. It also includes a general data flow analysis framework, but only for fixed nodes. This is sufficient for implementing its

Partial Escape Analysis [6] but is not appropriate for analyses that reason about floating arithmetic nodes, as would be needed for constant propagation. The GraalVM compiler currently has no data flow analysis framework that would take floating nodes into account. This work is the first step in a project to formulate such a general framework without needing to compute a schedule of the graph.

# 3 Approach

The approach presented in this paper aims to minimize analysis time of the graph by only lazily iterating over the parts of the graph relevant for finding constants while maintaining the power of *Conditional Constant Propagation* (CCP). This is in stark contrast to previous algorithms like *Sparse Conditional Constant Propagation* (SCCP) and *Sparse Simple Constant Propagation* (SSCP) which evaluate the entire (reachable portion of) the graph [7]. Additionally, while SCCP in the form described in the paper by Wegman and Zadeck [7] would require us to calculate a full schedule and to subsequently iterate over the entire reachable portion of the graph, our *Lazy Sparse Conditional Constant Propagation* (LSCCP) algorithm works directly on an unscheduled version of the Sea of Nodes representation.

Leaving parts of the graph unevaluated means that LSCCP needs to deal with unevaluated values as inputs for nodes and cannot assume that such nodes will be analyzed again later when complete information about all reachable inputs is available. Thus it needs to discern whether an input is unevaluated because the analysis has not reached it yet, or because this input will never generate a constant value and will therefore never be evaluated throughout the analysis.

The idea to mitigate this problem is to generally evaluate the program graph in a forward traversal order consistent with the order of execution which we refer to as a 'top-down' order (Section 3.5). This allows assumptions substituting missing information about inputs to be made safely.

Since LSCCP operates on an unscheduled Sea of Nodes representation where no global order of operations is known, we designed a priority metric for the work list based only on the available information. The priority metric enforces an order at critical points of the analysis where value flow analysis and reachability analysis influence each other ($\phi$ nodes and control split nodes). This allows value flow analysis and reachability analysis to be kept up to date with each other to provide each other with the best information possible.

Unlike classical data flow analysis where the order of evaluation is only relevant for the speed of the analysis, in our algorithm the order in which nodes are evaluated is essential for correctness.

## 3.1 Evaluation Domains

LSCCP uses two separate lattices to represent information in its value flow and reachability analysis respectively.

***Value lattice.*** Values are represented by a lattice as in SCC. To avoid confusion with the values of the reachability lattice, we refer to the value lattice's $\top$ element as `UNSEEN` (no value known yet), the middle tier values as `CONSTANT`, and the $\bot$ element as `UNRESTRICTED` (no constant can be guaranteed). We use the name `UNSEEN` as a shorthand: It denotes both nodes that have never been visited by the analysis as well as nodes that have been visited but that have `UNSEEN` inputs.

***Reachability lattice.*** In LSCCP all CFG edges are annotated with a reachability. Unlike SCC's two-tiered CFG reachability lattice, our reachability lattice contains three elements: `UNKNOWN` ($\top$), `UNREACHABLE` and `REACHABLE` ($\bot$). Using a lattice we can use logic in the reachability analysis that is similar to the value flow analysis when dealing with unevaluated inputs.

## 3.2 General Value Propagation

Initially all nodes in the graph are marked as `UNSEEN`. Due to the fact that value propagation starts at constants and not the CFG entry, evaluation may hit cases where a node's input is `UNSEEN` while it would be `UNRESTRICTED` if the input had been evaluated.

For example, consider evaluating the inequality check node representing the comparison $Phi(x) \neq 1$ in the graph in Figure 2. The first time we encounter this node in the analysis, one input is a constant 1 while the other input ($Phi(x)$) is still `UNSEEN`. We do not want to prematurely lower this to `UNRESTRICTED` since the `UNSEEN` input will become a constant later on, in which case we will want to produce a constant value for this node.

When visiting such a node, we treat all `UNSEEN` inputs as `UNRESTRICTED`. This still allows us to correctly treat an expression like `a * b` as 0 if `a` has been evaluated to 0 while `b` is still marked as `UNSEEN`. However, if any input was `UNSEEN` and the result of the visit would be `UNRESTRICTED`, we still propagate an `UNSEEN` result to signal that the result may be lowered to a constant later.

This allows us the flexibility to revisit nodes while preserving the overall invariant that a node's lattice value may only change to a lower value. In contrast to SCCP, an `UNSEEN` value for a node does not imply that the node has not been visited by the analysis yet.

## 3.3 Handling of $\phi$ nodes

The evaluation of $\phi$ nodes depends on both the value lattice elements for the $\phi$'s inputs and the reachability lattice element of the control flow edge associated with each input. For inputs coming from edges marked `UNKNOWN`, an assumption needs to be taken to evaluate the $\phi$ node. In general, a $\phi$ node can be evaluated using a *pessimistic* or an *optimistic* assumption regarding reachability. This means that incoming control flow edges with `UNKNOWN` reachability can be either

pessimistically interpreted as REACHABLE or optimistically interpreted as UNREACHABLE. A pessimistic assumption only retains maximum precision when we are sure that UNKNOWN edges will not be lowered to UNREACHABLE in the future.

**3.3.1 Straight-Line $\phi$ nodes.** Generally non-loop $\phi$ nodes are evaluated pessimistically. The top-down order of evaluating the graph ensures that the reachability information has already been calculated when evaluating the $\phi$ node, given that this $\phi$ node does not depend on backedges for which reachability can not yet be known and is subject to change. To show why we need pessimistic evaluation here, consider the following structure inside a loop:

```
if (condition)
  if (true)
    x1 = 1;
  else
    x2 = 2;
else
  x3 = 3;
x4 = ϕ(x1, x2, x3);
```

Here condition is not constant and will never be visited by our analysis because it cannot be evaluated at compile time, thereby leaving the reachability of the x3 input of the $\phi$ instruction UNKNOWN. The inner condition can be evaluated, leaving x1 as REACHABLE and x2 as UNREACHABLE. If we now optimistically assumed x3 to be UNREACHABLE, we would propagate the constant 1 into x4 which is an incorrect result. We would not recover from this mistake because condition will stay unevaluated throughout the analysis, thereby not triggering a reevaluation.

For a pessimistic evaluation to be admissible, it is required that UNKNOWN inputs are stable, which is the case for straight-line $\phi$ nodes as described in Section 4.2.

**3.3.2 Loop $\phi$ nodes.** Our handling of loop $\phi$ nodes combines both optimistic and pessimistic evaluation: We treat them optimistically when first entering a loop, but pessimistically after visiting the loop body.

In contrast to non-loop $\phi$ nodes, final reachability information for loop $\phi$ nodes is not yet available when a loop $\phi$ is evaluated for the first time when reaching a loop begin node. As the running example from Listing 1 demonstrates, a pessimistic assumption at this point would lose precision: Evaluating $Phi(x)$ pessimistically would never allow the CONSTANT 1 to enter the loop, thereby inhibiting the path through B2 to be found unreachable. This discovery is needed, however, to conclude that x is not modified inside the loop. Therefore, the loop $\phi$ must be evaluated optimistically to find that x is constant.

On the other hand, SCCP's optimistic analysis relies on the fact that every reachable block in the program will be visited, and every reachable control flow edge will be explicitly marked as reachable during the analysis. This allows the analysis to find the final value for a loop $\phi$ once the reachability information has stabilized. We cannot do the same kind of optimistic analysis since our analysis does not visit and mark all reachable control flow edges.

Therefore, we handle loop $\phi$ nodes as follows: When a loop is first entered, the loop begin's $\phi$ nodes are evaluated optimistically. This means that we assume that any UNKNOWN backedge may in fact be unreachable, and we ignore the associated input values. This allows us to propagate any constants entering the loop into the first loop iteration. At the same time, we schedule any loop $\phi$ with a not-UNSEEN lattice value for reevaluation after the entire loop (see Section 3.5.2). The organization of the worklist guarantees that the entire body is evaluated as far as possible before revisiting the loop $\phi$ nodes. At this point, the reachability of any UNKNOWN loop ends is guaranteed to be final (Section 4.2), and the loop $\phi$ node can safely be evaluated pessimistically, i. e., assuming that any still UNKNOWN backedge is now *reachable*. This may replace optimistically assumed constant values with the correct UNRESTRICTED value.

### 3.4 Reachability Propagation

To detect conditional constants, in addition to value flow, reachability needs to be taken into account. This information is tracked on a per edge basis because tracking it on a per block basis can lead to imprecisions, inhibiting detection of constants as shown by Click [1]. Processing, however, is done on blocks instead of edges by taking the reachabilities of the block's predecessor edges and its input values into account to calculate the reachability of its successor edges.

For a block to be considered reachable, it either has to be the start block of the CFG, or it has to have at least one predecessor edge that is not marked as UNREACHABLE. We can ignore backedges in this case because they can only occur on loop begins which have to be traversed to reach these backedges in the first place.

If a block has more than one successor, it must end with a control split node. In this case, the control split node is evaluated given its inputs and the successor edges are marked with the appropriate reachability lattice element. Similar to value flow explained in Section 3.2, immediately lowering successor edges from UNKNOWN to REACHABLE while predecessor edges are still UNKNOWN may inhibit future discovery of UNREACHABLE edges later on. Therefore, if an edge is considered reachable while predecessor edges are still UNKNOWN, we propagate UNKNOWN to signal that this edge might still be lowered later on.

To ensure that the control split nodes to be evaluated do in fact have up-to-date information from the value flow analysis, instead of immediately propagating reachability through them, they are scheduled using the worklist. While propagating reachability along the CFG, the reachability of

input edges of $\phi$ nodes may change. This new information triggers a reevaluation of all affected $\phi$ nodes.

### 3.5 Top-Down Analysis of the Graph

LSCCP evaluates nodes directly involved or closely related to control flow (such as $\phi$ nodes) in a forward traversal order consistent with the order of execution. Loops are analyzed to completion before information is propagated out of the loop, and predecessor blocks of control flow merges are evaluated before the merge. It creates conditions suitable for making assumptions about unevaluated inputs and to ensure that the best correct result can be calculated. This is achieved by the use of a priority-ordered worklist.

In the presentation that follows, lower numeric values denote higher priorities.

**3.5.1 Base Priority Metric.** In a quick pass over the CFG, a base priority is calculated for each block. This base priority is based on the minimal visit depth of a CFG block in a reverse postorder traversal [2]. A block's depth is calculated by incrementing the maximum of the depths of its predecessors by 1, ignoring loop backedges. The start block has a depth of 0.

The LSCCP base priority metric follows the same structure but extends it by adding one extra condition: If a block is a loop exit, its base priority does not only depend on its immediate predecessors but also all loop ends of the associated loop. All these loop ends are therefore regarded as predecessors of the loop exit while calculating the base priority. This effectively moves the loop exit below the entire loop in the priority.

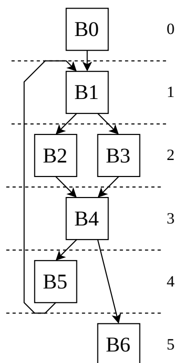



**Figure 3.** CFG for the IR presented in Figure 2.

Recalling the example introduced in Section 2.1, Figure 3 depicts the CFG from Figure 2. To the right of the CFG, the base priority of the blocks in the given line is given. Blocks that do not affect each other (in this example B2 and B3) can have the same base priority since the order in which they are processed does not affect the result of the analysis. The loop exit block B6 has a lower priority than any block in the loop, including the backedge block B5. This ensures that the loop is fully evaluated before evaluating any usages of values which depend on the given loop.

**3.5.2 Priority-Ordered worklist.** LSCCP uses a single priority-ordered worklist for both value flow and reachability analysis. This worklist internally consists of two queues. The first one is an unordered queue used for scheduling pure value flow nodes, such as arithmetic nodes. Elements are first removed from the unordered queue. Only if this queue is empty are elements from the second queue taken. Whenever a node from either worklist is visited, its usages are enqueued in the appropriate worklists if the analysis information associated with the current node changed.

This second queue is a priority queue. Elements are visited highest priority first (lowerst numeric priority value first).

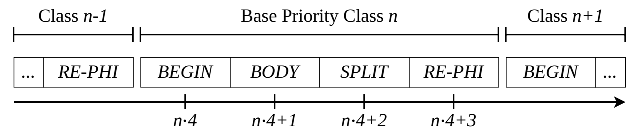



**Figure 4.** Internal layout of a block's base priority class.

The actual priorities used in the second queue are laid out as presented in figure 4. To maintain an approximation of the order of nodes within a basic block, the block's base priority *n* is quadrupled to allow us four *priority classes* per block. A priority class is a set (an equivalence class) of nodes with the same numeric priority value. We are not interested in the ordering of nodes within a priority class, and we do not need to represent the classes as explicit data structures. We only need nodes in higher-priority classes to be processed before nodes in lower-priority classes, which is ensured by the queue.

Nodes at the start of a CFG block and $\phi$ nodes are scheduled in the *BEGIN* class of their block's base priority class to ensure they are processed before any nodes in the current block that might use the value produced by this node.

Normal fixed nodes that can produce constant values (e. g., fixed division nodes that may raise an exception) are then scheduled using the *BODY* class. Finally, control split nodes that terminate blocks are scheduled after all fixed nodes in a block in the *SPLIT* class to ensure that all values that originate in the current block, in particular the condition controlling the split, are evaluated before the split itself.

In addition to scheduling a node given its base priority and position within a block, the worklist offers a second scheduling mode for $\phi$ nodes on loop begins (loop $\phi$ nodes for short). As will be discussed below (Section 3.3.2), loop $\phi$ nodes must be re-evaluated after the loop body has been fully evaluated. Therefore every loop $\phi$ is scheduled again in the `RE-PHI` class in the loop's last loop end (backedge) block in the priority queue.

Recalling the IR graph from Figure 2, $Phi(x)$ is initially scheduled with priority 4, which is the *BEGIN* class of the base priority class 1 of its associated loop begin. Scheduling the `If` node with the condition `x==1` shows why maintaining an order within a basic block is necessary. The `If` node uses a value which depends on $Phi(x)$ in the same basic block. Therefore the `If` needs to be evaluated *after* the given $\phi$ node in the *SPLIT* class, resulting in a scheduling priority of 6. To reschedule $Phi(x)$, we first obtain the maximum base priority class of any associated loop end (which is 4 for B5). Then, to ensure we capture all values produced by this block, we schedule it using the *RE-PHI* class resulting in an effective priority of 19.

If the node $Phi(a)$ were ever to be scheduled, it would receive the same priority as $Phi(x)$ because their relative positions in the graph are the same. This collision does not matter: Control split nodes and $\phi$ nodes that receive the same priority are guaranteed to be independent and can therefore be evaluated in any order. In practice, the data structure used is a priority queue, therefore nodes with the same priority are evaluated in a first-in-first-out manner.

The worklist keeps track of currently scheduled nodes, ensuring each node only exists in the list once. If a node is trying to be scheduled a second time with a different priority, the original priority is kept.

Overall our priority-ordered worklist captures the same relative ordering information between nodes that we would need from a schedule of the graph. However, as we only need ordering information between certain fixed nodes and $\phi$ nodes, and as nodes are only added to the priority queue on demand as required by the analysis, the computation of this ordering information is much cheaper than the computation of a full schedule.

### 3.6 Putting Everything Together

Finally, we present the full Lazy Sparse Conditional Constant Propagation in Algorithm 1.

We start by setting the value lattice elements of all nodes (denoted by $\lambda(node)$) to `UNSEEN` and the reachability lattice elements of all CFG edges (denoted by $\Lambda(edge)$) to `UNKNOWN`. Then we initialize the worklist with all constant nodes in the graph. LSCCP processes nodes until the worklist is empty. Finally, we replace all nodes for which we found new constants.

### 3.7 Example of LSCCP analysis

In this section we present a full run of LSCCP on the running example program with its graph shown in Figure 2. Table 1 shows the states of the worklist throughout the example run. The worklist is separated into the value queue holding floating arithmetic nodes, and the priority queue holding fixed nodes as well as $\phi$ and proxy nodes.

**Algorithm 1** LSCCP

```
1:  procedure LSCCP
2:      initialize all nodes with UNSEEN
3:      initialize all CFG edges with UNKNOWN
4:      initialize worklist with all constants
5:      while worklist has items do
6:          c ← worklist.next()
7:          if c is a ϕ-node then
8:              ProcessPhi(c)
9:          else if c is a control flow node then
10:             ProcessControlFlowNode(c)
11:         else
12:             ProcessValueFlowNode(c)
13:     replace all nodes found to be constant

14: procedure ProcessPhi(phi)
15:     if phi is a loop ϕ node ∧ λ(phi) = UNSEEN then
16:         new ← λ(first input of phi)
17:         reschedule phi if lowered
18:     else
19:         new ← meet(λ(reachable inputs of phi))
20:     UpdateValueLatticeElement(phi, new)
21: procedure ProcessControlFlowNode(flow)
22:     for all e in successor edges of flow do
23:         if CFG block of flow is reachable then
24:             new ← true if flow is no control split or e
                      is reachable according to
                      λ(flow.condition) else false
25:         else
26:             new ← false
27:         UpdateReachability(e, new)
28: procedure ProcessValueFlowNode(val)
29:     new ← evaluation of val given its inputs
30:     UpdateValueLatticeElement(val, new)

31: procedure UpdateValueLatticeElement(node, elem)
32:     if elem < λ(node) then
33:         λ(node) ← elem
34:         schedule usages of node
35: procedure UpdateReachability(edge, reachable)
36:     target ← target of e
37:     if reachable is false then
38:         new ← UNREACHABLE
39:     else if Λ(e) = UNREACHABLE then
40:         new ← REACHABLE
41:     else return
42:     Λ(edge) ← new
43:     schedule ϕ-nodes at the start of target
44:     schedule target
```

**Table 1.** Worklist states throughout the example run

| Step | Current Node | Evaluation result | Value Queue (after) | Priority Queue (after) |
|---|---|---|---|---|
| 0 | (initial state) | | C(1), C(2) | |
| 1 | C(1) | 1 | C(2), ≠1, -1, ≥1 | 4: $Phi(x)$ |
| 2 | C(2) | 2 | ≠1, -1, ≥1 | 4: $Phi(x)$, 12: $Phi(x')$ |
| 3 | ≠1 | UNSEEN | -1, ≥1 | 4: $Phi(x)$, 12: $Phi(x')$ |
| 4 | -1 | UNSEEN | ≥1 | 4: $Phi(x)$, 12: $Phi(x')$ |
| 5 | ≥1 | UNSEEN | | 4: $Phi(x)$, 12: $Phi(x')$ |
| 6 | $Phi(x)$ | 1 | ≠1 | 12: $Phi(x')$, 19: $Phi(x)$ |
| 7 | ≠1 | true | | 6: If(B1), 12: $Phi(x')$, 19: $Phi(x)$ |
| 8 | If(B1) | $\Lambda$(B1 ⟶ B2) := UNREACHABLE | | 8: Begin(B2), 12: $Phi(x')$, 19: $Phi(x)$ |
| 9 | Begin(B2) | $\Lambda$(B2 ⟶ B4) := UNREACHABLE | | 12: $Phi(x')$, 15: If(B4), 19: $Phi(x)$ |
| 10 | $Phi(x')$ | 1 | | 15: If(B4), 19: $Phi(x)$, 20: ValueProxy |
| 11 | If(B4) | | | 19: $Phi(x)$, 20: ValueProxy |
| 12 | $Phi(x)$ | 1 | | 20: ValueProxy |
| 13 | ValueProxy | 1 | | 21: Return |
| 14 | Return | | | |

***Steps 0–2.*** First we start by adding the two constant nodes `C(1)` and `C(2)` to the worklist. As these nodes are neither fixed nodes nor $\phi$ nodes, they are inserted into the value queue of the worklist. Now we remove `C(1)` from the worklist, set its value to a `CONSTANT` 1 and schedule all its usages. The inequality, subtract and greater-equal nodes are scheduled in the value queue of the worklist (recall that for brevity the usage of C(1) in these nodes was not indicated by edges), while $Phi(x)$ is scheduled in the priority queue with priority 4 (= $1 \cdot 4 + 0$, see Section 3.5.2). Evaluating `C(2)` similarly causes its value to be set to a `CONSTANT` 2, and its usage $Phi(x')$ is scheduled with priority 12 (= $3 \cdot 4 + 0$).

***Steps 3–5.*** Evaluating the previously scheduled inequality, subtract and greater-equal nodes yields `UNSEEN`: Each of these nodes has one `UNSEEN` and one constant input. Evaluation results in an unknown value which is propagated as `UNSEEN` to signal the possibility that the value may still become a constant in the future (see Section 3.2). Because this does not change the lattice values for these nodes, their usages are not scheduled.

***Step 6.*** Now the value queue of the worklist is empty, therefore we remove the first element of the priority queue, which is $Phi(x)$. As explained in Section 3.3.2, we optimistically assume the second input of $Phi(x)$ to be `UNREACHABLE` and propagate the `CONSTANT` 1 through this node. To check this assumption later on, we reschedule $Phi(x)$ with priority 19 (= $4 \cdot 4 + 3$). Because we lowered the value of $Phi(x)$, all its usages will be scheduled, causing the inequality and $Phi(x')$ to be scheduled. Since $Phi(x')$ already exists in the worklist, it is not inserted a second time (see Section 3.5.2).

***Step 7.*** The next node to be evaluated is the inequality node. This node now has two constant inputs and can be evaluated to a `CONSTANT true`, scheduling the associated `If` node with priority 6 (= $1 \cdot 4 + 2$).

***Steps 8–9.*** Evaluating the `If` node scheduled right before, we see that the false branch of this condition is unreachable because of the `CONSTANT true` input. Therefore, we set the edge from B1 to B2 to `UNREACHABLE` and schedule the begin node of B2 with priority 8 (= $2 \cdot 4 + 0$) to represent the entire block for reachability analysis, see Section 3.6. This node is immediately removed from the worklist and since the only predecessor edge is `UNREACHABLE`, the successor edge from B2 to B4 is marked `UNREACHABLE` and the `If` node at the end of B4 is scheduled (see Section 3.6) with priority 15 (= $3 \cdot 4 + 3$) to represent B4 for reachability analysis. Additionally, $Phi(x')$ at the start of B4 would be scheduled if it were not already in the worklist.

***Step 10.*** Now $Phi(x')$ is the next node to be evaluated. Its inputs are a `CONSTANT` 2 on the first input and a `CONSTANT` 1 on the second input. However, the first input value is considered unreachable because it corresponds to the CFG edge from B2 to B4, therefore the `CONSTANT` 1 from the second input can be propagated. We would schedule its usage (i.e. the loop $Phi(x)$ with priority 4), but because it is already in the worklist with priority 19, we do not re-schedule it. The second usage of $Phi(x')$ is the `ValueProxy` node which gets scheduled in its loop exit block with priority 20 (= $5 \cdot 4 + 0$, see Section 5).

***Step 11.*** The next node to be evaluated is the `If` node at the end of the basic block B4. B4 is considered reachable because it does not exclusively have `UNREACHABLE` edges as predecessor edges (the edge from B2 to B4 is `UNKNOWN` which is interpreted as `REACHABLE`, see Section 3.4). This in combination with the `UNSEEN` input coming from the greater-equal

node results in both successor edges being considered reachable, resulting in no change to their associated reachability lattice level.

***Step 12.*** Then $Phi(x)$ is reevaluated to check if the previous assumption for this node still holds. Both incoming control flow edges are `UNKNOWN` and therefore considered reachable, but both inputs to the $\phi$ are also associated with a `CONSTANT 1`. Therefore, this node is evaluated to `CONSTANT 1` which confirms our optimistic assumption made earlier and the value of $Phi(x)$ is not lowered further.

***Steps 13–14.*** The next node to process is the `ValueProxy` which propagates its input `CONSTANT 1`, resulting in the `Return` node to be scheduled with priority 21 ($= 5 \cdot 4 + 1$, see Section 3.5.2). Finally, the `Return` node is processed. Since it does not produce a value it is not analyzed further, thereby leaving the worklist empty.

***End.*** This concludes the analysis for conditional constants. We found four nodes ($Phi(x)$, inequality, $Phi(x')$ and `ValueProxy`) that can be replaced with constants at their usages. Additionally we found two CFG edges (B1 to B2, B2 to B4) and one basic block (B2) to be unreachable. These can be eliminated from the CFG.

# 4 Correctness and Precision

This section provides reasoning for the assumptions made in Section 3 that are needed to ensure that LSCCP finds at least as many constants as SCCP while not erroneously reporting values as constant.

## 4.1 Isolated Analyses

Looking at the general value flow analysis presented in Section 3.2 in isolation, we can conclude that for any given input, this analysis produces a correct result that is at least as high as the one generated by SCCP when interpreting `UNSEEN` as `UNRESTRICTED`, because we propagate constants as soon as possible while never blocking future analysis. All evaluation functions are designed to lower a value from `UNSEEN` as soon as possible without blocking future discovery of constants. If at any point in the analysis all inputs of a general value flow node were not `UNSEEN` and no `CONSTANT` value can be calculated, the evaluation function is required to result in `UNRESTRICTED` for all future queries, to uphold the assumption that during the evaluation of the first loop iteration we have the maximum amount of values assumed to be constant for Section 4.2.

Inspecting the pure reachability analysis (without control splits), it is easy to conclude that propagating the blocks reachability onto the successor edge if the block itself is considered unreachable, generates a correct result for these edges when interpreting `UNKNOWN` as `REACHABLE`. Since any unreachable edge must transitively depend on a constant value which in turn again depends on a constant, our analysis will find all of these edges.

## 4.2 Combination of value and reachability analyses

Reachability analysis and value flow analysis interact at control split nodes and $\phi$ nodes. Control splits depend on the reachability of their predecessor edges as well as the condition or value on which they split to produce results in the reachability analysis. The analysis of $\phi$ nodes depends on the reachability of the predecessor edges of their connected `Merge` as well as their input values to produce a result in the value flow analysis. To analyze such nodes without generating incorrect results while still finding at least as many constants as SCCP, we need to be able to draw reliable conclusions about the true values of `UNSEEN` and `UNKNOWN` inputs.

***Control splits.*** These nodes are handled very similarly to pure reachability analysis (recall Section 3.4). Given the value which the control split depends on is correct, it is easy to see that evaluation of these nodes yield the best correct result. The ordering in the worklist in combination with the initial optimistic evaluation of the loop $\phi$ nodes (recall Section 3.3.2) ensure that, any of the split's successor edges that is not `UNREACHABLE` on the first evaluation, will always be reachable throughout the analysis.

***Straight-line $\phi$ nodes.*** Straight-line $\phi$ nodes can safely assume that during their first evaluation all inputs associated with $\top$ lattice elements will stay that way throughout the analysis due to the worklist ordering, the initial optimistic evaluation of loop $\phi$ nodes and the evaluation functions of general value flow analysis as shown in Section 4.1. Such inputs can therefore be safely assumed as their $\bot$ counterparts resulting in a correct result that is at least as good as the one produced by SCCP.

***Loop $\phi$ nodes.*** When re-evaluating a loop $\phi$ node we rely on the fact that during the first evaluation of the loop, we work with the maximum amount of values that are assumed to be constant (which is ensured by the initial optimistic evaluation of said loop $\phi$ node), which would cause all CFG edges that might at some time be `UNREACHABLE` to be lowered to this reachability lattice element. This allows us to safely conclude that all edges that are `UNSEEN` after the first evaluation of the loop, are sure to stay that way throughout the analysis, which we rely on as mentioned in Section 3.3.2. This creates the necessary preconditions to treat the loop $\phi$ node as a straight-line $\phi$ node upon re-evaluation, meaning it can be evaluated pessimistically, guaranteeing correctness upon convergence without lowering precision below SCCP.

## 4.3 Conclusions

Because all parts of the analysis end up generating correct results we can conclude that the analysis as a whole generates a correct result. Termination is guaranteed because

the transfer functions for all nodes are monotone and the lattices have finite height.

Because all lattice elements that are UNSEEN or UNKNOWN throughout the analysis (which are the only lattice elements in the analysis for which this assumption is taken) can be safely assumed as their $\perp$ counterparts, and because the value lattice tracks at least as many elements as SCCP, the result generated by LSCCP is at least as good as the one generated by SCCP.

## 5 Implementation

We developed a prototype implementation of the LSCCP algorithm in the GraalVM compiler.

In our implementation, we assign priorities to loop exit nodes and the associated proxies (see Section 2.3) so that these are only visited once the loop's fixed point has been reached. Otherwise, if optimistic constant values or reachability information were to leak out of loops, we would waste effort analyzing code after the loop which would have to be re-analyzed once the fixed point is reached.

Additionally, in the procedures UpdateValueLatticeElement and UpdateReachability, checks were inserted to guarantee that monotonicity is upheld. In UpdateValueLatticeElement $elem \leq \lambda(node)$ must hold, otherwise monotonicity would be violated while in UpdateReachability the condition $reachable \vee \Lambda(edge) \neq$ REACHABLE must hold for monotonicity to be upheld.

### 5.1 Four-tier Value Lattice

In our implementation, the data representation of the value lattice uses the GraalVM compiler's existing 'stamp' infrastructure for representing ranges of values. While we do not propagate ranges in general, we use the fact that integer stamps provide a 'known to be nonzero' flag. Thus, in contrast to previous constant propagation algorithms which usually operate on a value lattice with three tiers (as presented in Section 2.1), our value lattice for integers actually has four levels: An extra level below all nonzero constants is added expressing that a value is not known to be constant but known not to be zero. Similarly, we track floating point values with a four-level lattice with a 'not-NaN' level, as well as object references with a 'not-`null`' level. We refer to these not-zero, not-NaN, and not-`null` values as the 'non-special' tier in the lattice.

The additional non-special tier is useful for evaluating common conditions such as `value != 0` (e. g. before a division) or `reference != null` (before a memory access that would otherwise raise an exception). The non-special tier is also useful for tracking Boolean values. In the JVM, Booleans are internally treated as 32-bit integer values where `true` is defined as any `value != 0`.[2] Therefore, `true` values are harder to track because they may not have a constant value associated with them internally, but the non-special tier in the value lattice allows for easy reasoning.

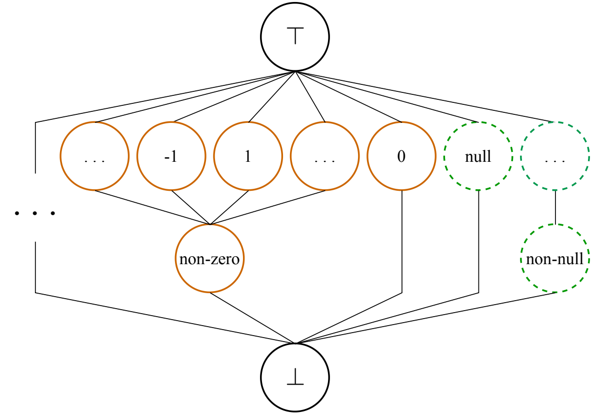



**Figure 5.** Four-tiered value lattice used by LSCCP.

The four levels described above are illustrated in Figure 5. The lattice elements between $\top$ and $\perp$ are color-coded to represent types. Integer values are outlined in orange (solid) while object references are outlined in green (dashed). The three dots on the left-hand side indicate that there are more types (e.g. floating point values) in this lattice than shown in the figure.

### 5.2 Conditional Nodes

The GraalVM compiler has conditional nodes representing the computation `condition ? trueValue : falseValue`, which are a fusion of a control split followed by an immediate merge and a phi node. This floating node poses an interesting problem because it has no ties into the control flow portion of the graph and is therefore hard to schedule with priority using the worklist.

While in the case of $\phi$ nodes, the condition and its associated control flow are guaranteed to be evaluated before the $\phi$ node, this is not the case with conditional nodes. Consider the case of a conditional `condition ? 1 : 2`, where the condition is still associated with the UNSEEN state. We might want to propagate the non-zero value for the result of this expression. However, if the condition became a known constant later, we would have to change this result from non-zero to a constant. This would violate monotonicity, as the non-zero tier is below the constants in the lattice, and values must only change to lower lattice elements.

To resolve this issue, we prevent the evaluation of a conditional node to non-zero if its condition is associated with UNSEEN. While this may inhibit further discovery of values, tests showed that this case rarely shows up in practice. It would be possible to handle this case precisely by adding another kind of 'non-zero' tier *above* the constant layer in the value lattice. We decided that this case was not worth

[2] https://docs.oracle.com/javase/specs/jvms/se22/jvms22.pdf section 2.11.1

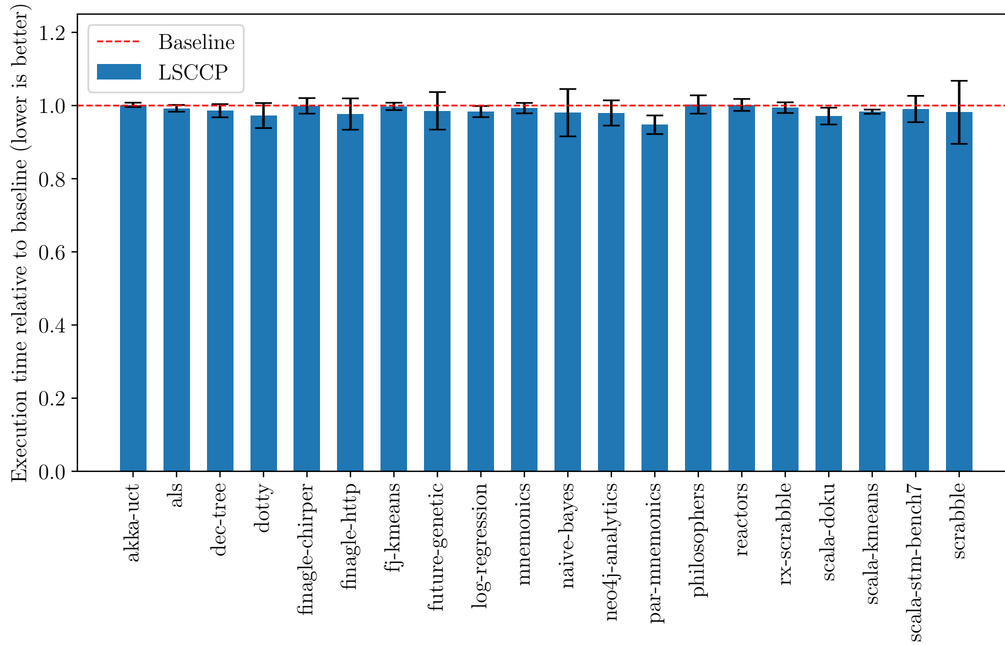



**Figure 6.** Results obtained for the Renaissance Benchmark Suite.

the extra complexity and prefer the slight imprecision from propagating UNSEEN instead.

## 6 Evaluation

Our implementation passes the unit test suite of GraalVM and the compilation of the entire Java Standard Library.

### 6.1 Benchmarking setup

We evaluated our implementation of LSCCP on the Renaissance benchmark suite[3]. The system used for testing runs Ubuntu 22.04 on an Intel 11th Gen Intel Core i7-1165G7 processor equipped with 64GB of RAM. A command line switch was built into GraalVM to disable the newly implemented conditional constant propagation phase to test baseline and the implementation of LSCCP on the same build. For each benchmark, warm-up runs are executed, followed by timed runs which contribute to the final result. The per-benchmark default number of warm-up and timed runs specified by the Renaissance benchmarking harness were used in this evaluation. In an effort to reduce the effect of noise, the entire benchmark suite was executed 15 times for both baseline and LSCCP. Baseline and LSCCP runs were carried out alternately to increase fairness.

### 6.2 Results

Figure 6 shows average results over the 15 benchmark runs for the 20 benchmarks of the Renaissance benchmark suite that are supported on the platform used for testing. The error bar indicates the standard deviation encountered over all timed iterations of the 15 LSCCP runs. As expected, LSCCP performs slightly better (though still mostly within margin of error) than baseline for most cases. No benchmark has seen any reliably measurable performance regression. The mean increase in performance measured over all 20 benchmarks in this suite was 1.4 % (minimum -0.28 %, maximum 5.25 %, median 1.43 %). LSCCP found on average 0.15 % of the values in the graphs to be constant, excluding constants found earlier through non-optimistic constant propagation.

The 'par-mnemonics' benchmark has seen the largest performance improvement of 5.25 %. In this benchmark, 0.29 % of all nodes were newly evaluated to be constant by LSCCP. The worst performance regression was measured for 'philosophers' with 0.28 %, for which only 0.03 % of all nodes were newly evaluated to constant by LSCCP. Since the conditional constant propagation phase does not change the graph if no new constants were found, the difference produced by LSCCP was negligible for this benchmark. We conclude that this result is within margin of error to baseline.

Our evaluation also showed that lazy iteration is very effective, as LSCCP only evaluates 20.5 % of the nodes in the graph on average. This is a significant improvement over the

[3] https://renaissance.dev/

previous state of the art, which would evaluate the entire reachable portion of the graph, which makes up 99.3 % of all basic blocks in the evaluated benchmarks. For the nodes that are visited by the analysis, the average number of visits per node is 1.1, indicating that the analysis quickly converges towards a fixed point.

Overall our optimization improves peak performance by an average of 1.4 % over GraalVM's optimized baseline on the Renaissance benchmarks, although this is mostly within the measurement noise on these benchmarks.

### 6.3 Discussion

As discussed before, GraalVM already performs conditional constant propagation, but only computing pessimistic fixed points for loops. Therefore any difference in our results vs. GraalVM's baseline analysis can only come from certain degenerate loop patterns, where our optimistic LSCCP analysis can prove that the loop will be exited on its first iteration, or that a variable used in the loop is a loop-invariant constant.

We do not provide a more detailed comparison of the two approaches: As GraalVM performs its non-optimistic constant propagation on the fly while simplifying nodes during other phases, there is no distinct non-optimistic constant propagation phase. Therefore, a more direct comparison against GraalVM's existing constant propagation would not be possible to do fairly since constant folding and propagation is deeply intertwined with the 'canonicalization' cleanups that run many times during compilation. Most compiler phases expect the program to be in canonical shape before they process them. Removing constant folding from canonicalization would have a very large detrimental impact on most of the compiler. Running a specialized baseline constant propagation pass at one or a few points in the compilation pipeline would be possible, but it would not make up for optimization opportunities lost from compiler phases that were unable to do their work on non-canonical inputs. Any such restructuring of the compiler would produce entirely artificial results.

## 7 Conclusions and Future Work

We presented a formulation of conditional constant propagation in a Sea of Nodes, exploiting the properties of its floating nodes and carefully organizing the iteration order of the analysis to connect data flow to control flow. Our approach features *lazy iteration* to reduce the portion of the graph necessary to be evaluated for finding all conditional constants. The evaluation of our prototype showed that lazy iteration is very effective, only evaluating 20.5 % of the graph.

In a next step, this analysis could be extended to allow `if` and `switch` statements to generate new data flow facts in the respective branches for the usages of the values they depend on. To achieve this, an approximation of a schedule must be calculated for values those values to find the correct usages to inject the data flow facts into.

This work is part of a larger project aimed at implementing a general data flow analysis framework in the GraalVM compiler. To our knowledge, this is the first general data flow analysis on the data flow component of a Sea of Nodes graph. Click's thesis [1] presents a powerful combined analysis that identifies constants, unreachable code, and congruences between values in the Sea of Nodes. However, this is a custom analysis that is not formulated in terms of a general data flow analysis framework.

## Acknowledgments

We would like to thank the anonymous reviewers and our shepherd Tomoki Nakamaru for their feedback that has helped us improve an earlier version of this paper.

This research project was partially funded by Oracle Labs. We thank all members of the Virtual Machine Research Group at Oracle Labs. Oracle, Java, GraalVM, and HotSpot are trademarks or registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners. We also thank all researchers at the Johannes Kepler University's Institute for System Software for their support of and feedback on our work.

## References

[1] Cliff Click. 1995. *Combining Analyses, Combining Optimizations.* Ph. D. Dissertation. Rice University. https://hdl.handle.net/1911/96451

[2] Keith D. Cooper and Linda Torczon. 2004. *Engineering a Compiler*. Morgan Kaufmann.

[3] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (1991), 451–490. https://doi.org/10.1145/115372.115320

[4] Gilles Duboscq, Lukas Stadler, Thomas Würthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. 2013. Graal IR: An Extensible Declarative Intermediate Representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop.* http://ssw.jku.at/General/Staff/GD/APPLC-2013-paper_12.pdf

[5] John B. Kam and Jeffrey D. Ullman. 1977. Monotone data flow analysis frameworks. *Acta Informatica* 7 (1977), 305–317. https://doi.org/10.1007/BF00290339

[6] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2018. Partial Escape Analysis and Scalar Replacement for Java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Orlando, FL, USA) *(CGO '14)*. Association for Computing Machinery, New York, NY, USA, 165–174. https://doi.org/10.1145/2544137.2544157

[7] Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant Propagation with Conditional Branches. *ACM Trans. Program. Lang. Syst.* 13, 2 (1991), 181–210. https://doi.org/10.1145/103135.103136

Received 2024-05-25; accepted 2024-06-24

# Mutator-Driven Object Placement using Load Barriers

Jonas Norlinder
Uppsala University
Uppsala, Sweden
jonas.norlinder@it.uu.se

Albert Mingkun Yang
Oracle
Sweden
albert.m.yang@oracle.com

David Black-Schaffer
Uppsala University
Uppsala, Sweden
david.black-schaffer@it.uu.se

Tobias Wrigstad
Uppsala University
Uppsala, Sweden
tobias.wrigstad@it.uu.se

## Abstract

Object placement impacts cache utilisation, which is itself critical for performance. Managed languages offer fewer tools than unmanaged languages in the way of controlling object placement due to the abstract view of memory. On the other hand, managed languages often have garbage collectors (GC) that move objects as part of defragmentation.

In the context of OpenJDK, Hot-Cold Objects Segregation GC (HCSGC) added locality improvement on-top of ZGC by piggybacking on its loaded value-barrier based design. In addition to the open problem of tuning HCSGC, we identify a contradiction in two of its design goals and propose LR, that addresses both these problems.

We implement LR on-top of ZGC and compare it with GCs in OpenJDK and with the best performing HCSGC configuration using DaCapo, JGraphT and SPECjbb2015. While using less resources, LR outperforms HCSGC in 18 configurations, matches performance in 17, and regresses in 3.

***CCS Concepts:*** **• Software and its engineering → Garbage collection**.

***Keywords:*** garbage collection, locality, cache performance

**ACM Reference Format:**
Jonas Norlinder, Albert Mingkun Yang, David Black-Schaffer, and Tobias Wrigstad. 2024. Mutator-Driven Object Placement using Load Barriers. In *Proceedings of the 21st ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '24), September 19, 2024, Vienna, Austria.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3679007.3685060

This work is licensed under a Creative Commons Attribution 4.0 International License.

*MPLR '24, September 19, 2024, Vienna, Austria*
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1118-3/24/09
https://doi.org/10.1145/3679007.3685060

## 1 Introduction

Garbage collection (GC) provides a natural opportunity to adjust the placement of objects in memory, taking into account the complexities of varying lifetimes and access patterns across individual objects, object types and program phases. The growing gap between processor speed and memory speed [31] stresses the importance of such movement to improve cache locality. In this paper, we identify limitations in a state-of-the-art design for using garbage collection to improve data locality and demonstrate how to address them to both improve performance and reduce overhead.

Using GC to improve object locality has been explored extensively [10, 11, 15, 18, 25, 30], with the Hot-Cold Objects Segregation GC (HCSGC [33]) being one of the most recent proposals. HCSGC provides two main ways of improving locality: 1) it tracks which objects were recently accessed by mutators—hot objects—and segregates these objects from cold objects during evacuation and, 2) it piggybacks on ZGC's load value barrier based design to have mutators move objects they access into local allocation buffers during GC. The first increases cache utilisation by increasing the probability that objects on a cache line are accessed. The causes objects accessed by a mutator to be placed adjacent in access-order so repeated accesses enjoy good cache locality.

Because HCSGC move objects to improve locality as part of ZGC's GC, only objects on sparsely populated pages will be moved by mutators because ZGC only evacuates sparse pages. HCSGC supports classifying pages that have mostly cold objects as sparse (even if they are not sparse) by assigning a weight to cold live bytes or evacuating all pages in the heap in every GC cycle. How to properly select a weight for a given program is an open problem and likely no generic solution exists. However, HCSGC showed that evacuating all pages in the heap leads to good performance despite the extraneous copying, as it is built on-top of a concurrent GC. The drawback is that a substantial number of cold objects will be moved by GC threads, without any immediate benefits to locality. Therefore, selecting all pages for evacuation only works when there is ample of headroom in the machine.

We present a locality-improving extension to ZGC that does not need tuning of a threshold value and uses substantially less CPU than HCSGC. We call our system LR. Our implementation, LR-ZGC, consistently outperforms HCSGC and incurs less overhead on the system, by copying fewer cold objects compared to evacuating all pages in the heap, as the best-performing configurations for HCSGC.

1. We identify a contradiction in the goals of the design of HCSGC that cannot be addressed simultaneously: moving hot objects to put them in mutator-accessed order for locality improvements; and moving only objects on sparsely populated pages to avoid work that does not lead to memory reclamation (§2.6).
2. We propose LR, which can address both goals simultaneously and show how it reduce movement of cold objects while giving mutators the opportunity to move any object they access (§3).
3. We evaluate LR through our implementation called LR-ZGC, which is built on-top of OpenJDK. We compare against existing production-grade throughput-oriented and latency-oriented collectors on OpenJDK with benchmarks constrained to large heaps (§§4 to 5). An artefact containing our implementation can be found at [23].

## 2 Background

### 2.1 The Memory Wall and Cache Locality

The memory wall [31] refers to the performance gap between processors and memory. This gap leads to performance being memory-bound, as the processor spends much of its time waiting for the much slower memory to deliver data. Cache memories can often hide these latencies by taking advantage of the *temporal locality* (re-using the same value within a short time) and *spatial locality* (using values that are close together in the address space within a short time) of programs. Data that can be served from cache is retrieved several orders of magnitude faster than from main memory.

Performance-sensitive programming often involves optimising the layout of data structures to ensure *cache friendliness*. This may involve storing values that are used together adjacently in memory or changing the order of loops to increase data reuse. Such optimisations are complex, as predicting the access patterns critical for performance can be difficult and may vary between phases of a program or be determined by user input. Even if a priori knowledge can be provided by an oracle, constructing the perfect memory layout is an NP-complete problem [24], making it impossible to construct a general algorithm that can optimally solve this problem. Automated techniques for improving cache performance show promising results [8, 10, 11, 13, 18, 25, 30, 33]. In languages where the programmer cannot explicitly influence object placement in memory, automated techniques are important for cache friendliness.

### 2.2 Creating and Maintaining Cache Locality

The combination of application access patterns and how the runtime allocates and moves objects determines the data locality and cache performance. For example, applications frequently access objects in the order they were created, suggesting that allocators should make an effort to allocate objects in address order based on object creation order. Indeed, Abuaiadh et al. [1] showed that disturbing this order could drastically reduce application throughput and Norlinder et al. [22] showed that keeping object creation order improves throughput by 53% and reduces cache misses by 79% in the sunflow benchmark in DaCapo [4].

Bump-pointer allocation is common in managed languages, gives good locality and deliver high allocation throughput. It maintains a pointer to the beginning of free space and allocation "bumps" the pointer by the requested size. This scheme has gained popularity not only because of its simplicity but also its implication for object locality. If an application accesses objects in the order they were created, bump-pointer allocation gives rise to good locality by design.

Using GC to moving objects during garbage collection to improve locality is well-documented [2, 13, 33] and there are several proposals to improve the organisation of objects in memory during garbage collection. For example, Blackburn et al. [3] found that copying objects in depth-first order rather than in breadth-first order in a copying collector yields better performance, and Courts [13] and Yang et al. [33] improve throughput by letting mutators move hot objects.

### 2.3 Barriers and their Use to Detect Access Patterns

Concurrent GC permits garbage collection without stopping the program. This requires coordination between the program and the collector. This coordination is opaque to the programmer but may introduce performance regressions. One technique for such coordination is by inserting *barriers* in the programs. Barriers force a program to interact with the collector at key places to ensure correctness. There are different types of barriers with different trade-offs in overhead and capabilities. Yang et al. [35] and Blackburn and Hosking [5] have studied the overhead of different barriers and show load barriers, barriers that are inserted around loads of references from the heap, are more expensive than write barriers, inserted in stores of references to the heap. The fundamental reason for this is that programs tend to have more reads than writes.

Load barriers enable evacuation concurrently with application threads. In the context of cache locality, load barriers can provide information on the mutators' access patterns and permits objects to be moved in the application's access order. Previous work has successfully used information obtained from load barriers to group objects concerning their access frequency [18, 33] and temporal use together [11].

### 2.4 Compressed Pointers and Cache Performance

Most systems use a 64-bit addressing architecture. While this increases the addressable memory, it incurs a 2× overhead on every pointer in memory (64 bits instead of 32 bits), regardless of whether the application needs the additional addressable memory. Venstermans et al. [29] showed that Java heaps grew by 40% when transitioning from 32-bit to 64-bit mode, data-cache misses increased at all levels, and throughput declined. One approach to mitigate the costs of using a 64-bit architecture is to compress pointers to 32-bits by representing addresses as offsets from the start address of the heap. In OpenJDK, for example, all collectors except ZGC support and automatically use "compressed oops" for heaps less than 32 GB in size (larger heaps are possible at the cost of additional fragmentation due to an increase in object alignment). Compressed oops works similarly to barriers: each time a pointer is loaded from the heap it is decompressed (by applying the heap's start address and adjusting for alignment) to a full 64-bit pointer and each time a pointer is written to the heap, it is compressed. This allows an application to store pointers in memory (and in cache) in half the space, at the cost of compressing respective decompressing them at reads and writes to the heap.

ZGC does not support compressed pointers because it uses the higher bits of the pointers to store metadata to support concurrent garbage collection.

### 2.5 ZGC: The Z Garbage Collector

For brevity, we focus on the parts of ZGC that are most relevant to this work. For a more complete and detailed explanation, please refer to Yang and Wrigstad [34].

ZGC is a mostly concurrent, low-latency, parallel, mark–evacuate GC in OpenJDK. Its goal is to deliver low tail latency and happily trades decreased throughput for lower latency. ZGC is similar in many ways to Pauseless [12] and C4 [28]. OpenJDK contains both in a non-generational (currently default) and a generational version of ZGC. We limit ourselves to non-generational ZGC for a better comparison with HCSGC, leaving a generational version for future work.

ZGC GC cycles are most commonly triggered by the allocation rate rule, due to the predicted duration of the next GC cycle being too close to the predicted time when the system would otherwise run out of memory (using a conservative prediction of the allocation rate to account for sudden spikes). (HCSGC and LR inherit the exact same mechanisms.) The cycle begins by GC workers walking the object graph, marking all live object. The liveness information is then used to select memory regions, or pages in ZGC parlance, that are sparsely populated for *evacuation*. These sparse pages are the *from-space* and ZGC allocates new page(s) as *to-space*. Once all live objects on a from-space page are moved to to-space, the page is reclaimed. ZGC uses loaded value barriers [28] to prevent mutators from following pointers which have been invalidated due to concurrent object movement. ZGC tracks pointer validity by "colouring" pointers on the heap. A pointer is guaranteed to be valid it is has the *good colour*; else is has a *bad colour* and is invalid. Colouring is done by using four higher bits in each pointer to encode its colour.

The loaded value barrier inspects the colour bits each time a pointer is loaded from the heap. A good-coloured pointer can be used directly, but a bad-coloured pointer first requires checking if the pointer points to an object that may have been, or is scheduled to be, moved. If the object has already been moved, the correct pointer is looked-up in a forwarding table. Otherwise, the object is moved before its new address is loaded. In both cases, the original pointer is patched to point to the correct location (with the good colour) for future access. (The barrier is "self-healing".)

A ZGC cycle consists of three phases, *marking/remapping* (M), *eviction candidate selection* (C) and *evacuation* (E), as shown in Fig. 1. The phases are concurrent with mutators and before each phase is a brief stop-the-world pause.

**2.5.1 Marking and Remapping (M).** During marking and remapping phase the live heap is walked. At the start of each cycle, all pointers are invalidated by changing what is the good colour. Colour invalidation is needed to support concurrent marking. As a side-effect of marking, colours are updated to the good colour and addresses to moved objects are remapped to their new locations.

**2.5.2 Selection of Evacuation Candidates (C).** In the EC selection phase, ZGC iterates over pages and uses the liveness information from the M phase to identify sparsely populated pages. Only sparse pages are selected for evacuation to minimize the number of object that must be moved to be able to free the pages. These pages are added to the evacuation candidates set (EC set) to be reclaimed after their live objects have been moved in the E phase. The default fragmentation tolerance in the heap is 25%. Pages are added to the EC in order of decreasing sparsity, such that the fragmentation of the remaining pages is less than the tolerance.

**2.5.3 Evacuation (E).** In this phase, GC threads evacuate the pages in the EC set by moving all their live objects. When all live objects from a page have been moved, the page may be reclaimed. During the evacuation process, ZGC stores forwarding information for the moved objects on the heap. This, together with the loaded value barrier and pointer colouring, allows the pages to be reclaimed immediately once all their live objects have been moved without waiting for all pointers to the old locations to be updated first.

### 2.6 HCSGC: Hot–Cold Segregation GC

HCSGC's [33] goal is to place *hot objects* (recently accessed by mutators) close in memory to improve locality and thereby enhance cache performance. HCSGC is built on-top of ZGC and uses ZGC's loaded value barriers to drive movement of

hot objects. Evacuation in HCSGC follows ZGC as shown in Fig. 1a. This means that during this phase, mutators and GC workers "compete" to move objects. As the former leads to locality improvements and the second does not, HCSGC permits delaying evacuation by GC workers to the start of the next GC cycle, as shown in Fig. 1b. This increases the probability that object movement is performed by mutators.

Since ZGC only condemns sparse pages, evacuation does not improve locality of objects on dense pages. To this end, HCSGC permits assigning weights to cold live bytes so that dense pages with few hot objects appear sparse, allowing movement of objects on this page: if a page has 100 kB live bytes in cold objects and the weight is 0.5, we count those live bytes as 50 kB. How to find a good weight for cold bytes for a particular application is an open problem and there is no intuition for programmers to make a selection as they are not in control of how objects are placed in memory. Picking a too low weight risks moving more cold objects as more pages are added to the EC set; picking a too high weight risks prevents hot objects from being moved. This requires tracking whether an object is hot or cold, which incurs a small performance and memory overhead.

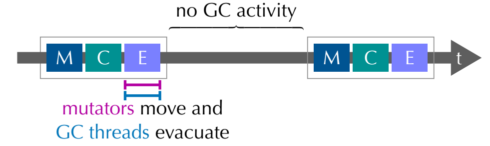
In this paper, we focus on the HCSGC configuration that avoids the problem of finding a good weight: evacuate all pages on the heap each GC cycle. The performance of this configuration was among the best, except in the case of a saturated machine [33].[1] In this configuration, HCSGC delays evacuation by GC workers until the next GC cycle (as in Fig. 1b). To ensure that all hot objects can be moved, all pages are included in the EC set. This means all objects will be moved each GC cycle, it adds considerable extra work but avoids the cost of tracking per-object hotness.

The evaluation of HCSGC [33] shows that this configuration performs the best for most benchmarks. Notably, this is the opposite of what ZGC does: HCSGC constructs the largest EC set (all objects) and lets mutators move the hot ones while GC threads move the others. ZGC, in contrast, only moves objects on sparse pages and GC threads move as many objects as possible to off-load mutators. The drawback of the HCSGC approach is unnecessary movement of many cold objects, and its performance decreases in CPU/memory-constrained environments. Moreover, numerous forwarding tables are allocated, which increases memory overhead.

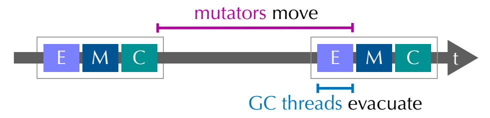
HCSGC ends up trying to achieve conflicting objectives:

1. Move all hot objects to place them in mutator-accessed order to increase locality, which requires that all (even dense) pages are in the EC set.
2. Evacuate only sparsely populated pages to avoid wasting time on movement that does not lead to memory reclamation.

[1]From this point, we use configuration 4 from HCSGC as our baseline. In this configuration, per-object hotness is not tracked, all objects on all pages are moved each GC cycle, and evacuation by GC workers is delayed.




**(a)** Phase order and object movement in ZGC.




**(b)** Phase order and object movement in HCSGC and LR.

**Figure 1.** Phases in ZGC, HCSGC and LR. Mutators move objects in the loaded value barrier when an invalid pointer is loaded from the heap. GC threads evacuate objects in address-order by iterating over a livemap.

Since HCSGC is built upon the infrastructure in ZGC, the first objective demands a large EC set, while the latter demands a small EC set. In the next section, we propose a design that solves this problem by trading improved locality for increased fragmentation; it permits hot objects to be moved while permitting the cold objects to stay in place.

## 3 The Design of LR

In this section, we describe the design of our system which we call LR, short for *Locality-Refined*. LR overcomes several of the limitations of HCSGC and removes the need for tuning knobs. As will be demonstrated using our implementation of LR called LR-ZGC in §4, LR delivers higher performance than HCSGC overall at a lower overhead.

LR is a GC-based technique for improving locality that addresses the trade-off between moving objects to *reclaim memory* and to *improve locality*. LR is inspired by Courts [13] and Yang et al. [33] and, as HCSGC, uses mutators to drive the locality optimisation. However, unlike HCSGC, LR decouples locality optimisation from memory reclamation and addresses each objective separately. Specifically, in LR, the ZGC EC set retains its sole purpose of being used only to reclaim memory. To address locality optimisation, LR introduces a new LOC set—the locality optimisation candidates set—which consists of all non-EC pages whose objects may be moved. For technical reasons that are specific to ZGC, objects allocated in cycle $n$ cannot be moved until cycle $n + 1$. Thus, LR (just like HCSGC) only supports locality improvement of objects on pages at least as old as the previous cycle. We refer to these as the *movable pages*. A mutator accessing objects in the LOC set sees the object as movable, causing it to be moved if its reference is loaded to the stack. A GC worker is unaware of the LOC set and only works on the disjoint EC set.

Movement of objects in LR works as in HCSGC (Fig. 1b). An object in LOC may be moved *at most* once by a mutator (if accessed) but never by a GC thread and an object in the EC set is guaranteed to be moved *exactly* once by either a mutators or by a GC thread (depending on who accesses it first—mutator or GC worker). Since the EC set only contains sparsely populated pages and only hot objects in the LOC set are moved, object movement work is significantly reduced compared to HCSGC's movement of all objects. Additionally, this approach does not miss locality optimisation for densely populated pages and does not incur the additional cost of hotness tracking. The cost is an increase in fragmentation due to moving hot data out of dense pages, creating a hole.

The use of separate EC and LOC sets achieves both objectives simultaneously and without conflict: the EC set includes only sparsely populated pages, such that evacuating them maximises the amount of memory reclaimed for minimal work, and only the hot objects from the LOC set are moved, thereby achieving the full mutator-driven locality orgnisation without moving any cold objects.

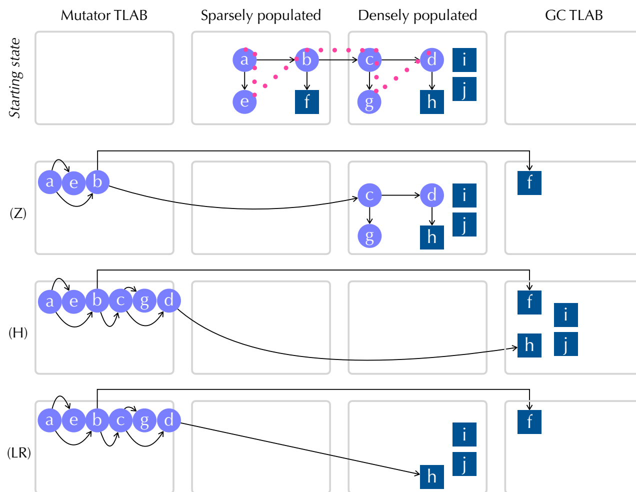



**Figure 2.** Comparison of how ZGC, HCSGC and LR move objects. Mutators move objects to mutator TLABs; GC threads evacuate to GC TLABs.

Fig. 2 compares the movement strategies in ZGC, HCSGC and LR and its impact on what objects are moved and by who (mutator or GC). It contains a sparse page (in EC set) and a dense page, plus two thread-local allocation buffers (TLABs)—one for a single mutator and one for a single GC worker—where all allocations happen. In the figure, all allocations are due to object movement. The dotted line shows the order in which a mutator accesses the round objects, from left to right. Square objects are not accessed by mutators.

Whether the page where an object resides is included in EC or LOC, and whether a mutator or a GC worker is first to access it determines if an object is moved and by whom (and thus to where). ZGC's EC set only include sparsely populated pages. Thus, as shown in Case (Z) of Fig. 2, we can typically expect only a small number of the objects to be moved in mutator access order ($a$, $e$ and $b$ are put in access order in the Mutator TLAB), as only those on sparse pages are moved *and* the mutator has to get to them before the GC threads do. (The figure shows the best case for ZGC, where the mutator always gets to the objects first.) In HCSGC, all objects will be moved, so even those that are on the densely populated pages ($c, d, g, i, h,$ and $j$) will be moved.[2] Due to the delay of the evacuation phase showed in Fig. 1, the probability that the mutator getting to move objects in access order is increased, as shown in case (H) where $a$, $e$, $b$, $c$, $g$, and $d$ are moved in order. However, objects that were not accessed by the mutator ($f, h, i,$ and $j$) are moved by the GC, even if they were on a densely populated page—the price of having to put a page in from-space to make it movable. For LR, all objects touched by the mutator are moved as with HCSGC, but cold objects on densely populated pages remain in place (due to the pages being in the LOC set). This means that only objects that could benefit from access order placement ($a, b, c, d, e, g$) or that would free up a sparse page ($f$) are moved.

This example shows why LR and HCSGC will typically move more objects than ZGC. Notably, in HCSGC, *all* object movement is subject to "competition" between the mutator and the GC thread. In LR, this is only the case for objects on the sparsely populated page (since this page is selected for evacuation to reclaim memory).

### 3.1 Trading Fragmentation for Possible Cache Performance Improvement

The LOC set trades increased heap fragmentation for improved cache performance. The increase comes from moving an object leaving an unusable gap in its place. In the next section, we discuss how we can use this fragmented space.

There is a built-in feedback loop to mitigate this. If a page becomes very fragmented due to objects being moved from the page in order to improve their locality, the page will eventually be added to the EC set by ZGC's criteria and reclaimed in the next cycle.

Just like in the case for HCSGC, LR's design is based on the assumption that the application exhibits sufficiently stable and recurring access patterns that the added cost of moving objects according to those patterns can be earned back through improved cache performance when they are repeated. In § 3.3, we discuss possible mitigations for cases where this assumption turns out to be false.

[2] Assuming a good weight for cold live bytes could be obtained for HCSGC (so dense pages can be in EC), or if HCSGC is configured to always put all pages in the EC set. If not, HCSGC would behave like ZGC, except that mutators are more likely to win the competition to move objects.

### 3.2 Forwarding Information

In all three cases, objects are moved concurrently with mutator accesses, which necessitates storing forwarding information for the moved object to ensure correctness. Next, we look at how to manage forwarding information for moved objects in the LOC set. A straightforward solution is to use the same strategy as ZGC uses for pages in the EC set: off-heap forwarding tables. A downside to this approach is that these tables can consume a substantial amount of memory [22]. The motivation for using off-heap forwarding tables—to enable the reclamation of the page as soon as the last object is moved off it, rather than having to wait until all references have been updated— does not apply, as we do not condemn LOC pages, only move objects that are accessed by mutators.

A more memory-efficient approach is to store the new address in the old location of a moved object, as in Cheney's copying GC [9]. In this design, the old objects' locations hold the forwarding addresses until the end of the GC cycle, at which point all old pointers will have been remapped and the forwarding addresses are no longer needed. In contrast to storing forwarding information on a page that is being evacuated in order to be reclaimed, this does not delay the earliest possible reclamation of these pages.

In ZGC, only pages in EC have a forwarding table. ZGC, HCSGC and LR all check whether a page is in EC by looking up its forwarding table. Since in LR, all non-EC pages are in LOC, absence of a forwarding table means membership in LOC, meaning forwarding addresses for objects on that page are stored in the original copy of the moved object.

### 3.3 Limitations and Future Work

Movement of objects on LOC pages piggybacks on ZGC's loaded value barriers which only move objects during a GC cycle. Thus, locality optimisation in LR is controlled by the number of GC cycles. Considering that the purpose of GC cycles (reclaiming memory) and object reorganisations (locality optimisation) do not completely align, it would be more flexible if they were fully decoupled. That would make locality optimisation independent of running GC, similar to work by Chen et al. [8]. This is an interesting direction for future work and will require developing new heuristics for when to perform locality improvements. We speculate that the information collected via loaded value barriers could be used to construct a metric for locality; if its value drops below a threshold, trigger a pure locality optimisation cycle.

The current design does not have a mechanism for checking whether this assumption holds for a particular program or part of the heap. With such a mechanism, it would be possible to back-off trying to improve locality for particular threads, parts of the memory, or all together. Such a mechanism could be based on monitoring cache performance counters before and after "location improvement cycles", for example. We leave further exploration of this to future work.

Additionally, we might consider filtering which objects are moved to improve locality. Most likely, moving objects that span several cache lines will have a limited impact on cache performance as accesses to proceeding objects will only bring a small portion of the object into cache. ZGC—and by extension HCSGC and LR—all divide objects into three size classes: small, medium and large (*c.f.,* [34]). We only perform locality optimisations for objects on small pages.Thus, currently, LR puts movable pages in LOC if they hold "small objects" (up to 256 Kb), which is far larger than the 64B cache lines. We believe that a more finely-tuned and possibly adaptive, size threshold could minimize copying overhead without hurting locality improvements. Likewise, we might consider filtering based on ages—short-lived objects have a higher likelihood of yielding a low return on investment. Based on this observation, integrating LR into *generational* ZGC is a promising next step for this work, with considerable technical challenges due to implementation details in ZGC. We intend to pursue this as future work.

## 4 Method

We build LR-ZGC, our implementation of LR, on top of ZGC in OpenJDK 15. We benchmark on an AMD Ryzen 9 5900X with 12 cores with two threads per core, 32kB data and 32kB instruction L1 per core, 512kB L2 per core, 64 MB of shared L3 and 128 GB of RAM, running Ubuntu 22.04.4 LTS with Linux kernel 6.5.0-28-generic. OpenJDK[3] was compiled with GCC 11.4.0 with gnu++11. C-states were disabled and CPU governor was set to performance for all cores.

### 4.1 Evaluated Collectors

We compare LR-ZGC with the Garbage-First (G1), Parallel, Shenandoah and ZGC collectors from OpenJDK 15 and HCSGC. The Parallel GC is a generational STW collector optimised for throughput. G1 [14] is a mostly-concurrent generational collector. G1 performs all activities concurrently with mutators except for moving objects. This reduces pause times substantially, but worst-case pause times are still proportional to the size of the live set. To support concurrent marking, G1 uses write barriers.

The Shenandoah collector is a non-generational,[4] concurrent collector delivering low pause times [16]. A GC cycle collects the whole heap by starting with concurrent marking (using write barriers to maintain tri-colour invariants), continuing with concurrent evacuation of objects in sparse regions and ending with concurrent updates of pointers for moved objects and releasing evacuated sparse regions. In OpenJDK 15, Shenandoah GC uses a Brooks' barrier [20]

[3]The source code of OpenJDK 15 we use: https://github.com/openjdk/jdk/releases/tag/jdk-15+36.

[4]An experimental version that supports generations is available in later versions of OpenJDK. We did not use it in this work.

which uses a pointer in object headers to determine the current location of an object. Objects in the EC set that are not yet moved point to themselves, otherwise they point to the new location of the object.

We ported HCSGC to OpenJDK 15. Our implementation is extracted from the artefact provided by Yang et al. [33], but only includes the logic needed for the configuration that moves live objects in all small pages and defer evacuation to the start of the next GC cycle. This has the added consequence of removing some conditional checks, so if the performance of HCSGC is affected, it should be improved.

### 4.2 Benchmarks

Akin to Yang et al. [33], we use JGraphT 1.5.0 [19], DaCapo 9.12-bach-MR1 [4] and SPECjbb2015 [27]. From JGraphT, we run maximal clique (MC), which implements the Bron-Kerbosch maximal clique enumeration algorithm [26] and connected components (CC), which implements a biconnected components algorithm [17]. The graph data, *uk-2007-05@100000* and *enwiki-2018*, are from Laboratory for Web Algorithms (LAW) [6, 7]. Using the complete data set is excessively time-consuming so we use a subset. Similar to Yang et al. [33], in DaCapo, we focus our attention on the only two applications in DaCapo that supports a *huge* input size, namely *h2* and *tradebeans*.[5] In these two we also fix the number of driver threads to four as performance does not scale by using even more threads as reported in Norlinder et al. [22] Figure 12 and Kalibera et al. [21]. As the minimal heap sizes and thus the live sets for the remaining DaCapo programs are probably small enough to fit in the L3 cache of our benchmark machine, we do not expect to see any benefits from HCSGC or LR-ZGC for these benchmarks, more likely regression due to the extra work involved in copying data which is already in cache. Nevertheless, we include these applications running them at the largest available input size.

### 4.3 GC Worker Thread Count

By default, the GCs set the number of threads using heuristics based on the number of hardware cores, ignoring the number of application threads.[6] To ensure that our results are not skewed by differences in how different GC's sizes its pool of workers, we explicitly set the number of GC. For parallel GC work (work carried out in STW pauses), we use half of the cores (12 on our machine). For concurrent GC work (work carried out in concurrent GC phases) we use one-quarter of the number of available cores (6). A comparison of our thread assignment and the GC's defaults is shown in Table 3.

### 4.4 Heap Sizes

To explore the time-space trade-off we run benchmarks using three configurations with different heap sizes, $\{1.5\times, 3\times, 6\times\}$ the minimal heap size (see below), plus a fourth (see below also). All collectors available in OpenJDK 15 are included in the evaluation, and all have different trade-offs that impose different heap requirements. Concurrent collectors are used in scenarios requiring low latency. They typically require a larger heap compared to throughput collectors, as they trade memory and throughput for lower latency. When a concurrent collector is run with a heap that is too small, the collector eventually suffers from allocation stalls, when the reclamation rate is not able to match the allocation rate. This puts the GC on the critcal path to performance (*e.g.,* mutators must wait until GC threads are able to reclaim memory). This makes frequent allocation stalls terrible for latency, and if they occur, the heap size should be increased to provide good latency. As LR-ZGC is implemented in a concurrent GC it is paramount that we also evaluate with a heap that is large enough to avoid frequent allocation stalls. Thus, we also include a fourth heap size, denoted the "stall-free heap size".

An application's minimal heap is the smallest heap that can be used without crashing due to an out of memory exception. We determined the minimal heap size for each benchmark through a binary search for the smallest heap that did not cause an out of memory exception using the Serial GC. The Serial GC was used to minimise the impact of non-determinism from the VM, application, etc. Stall-free heaps were found by running each benchmark with ZGC, gradually increasing the heap size in increments of 10 MB until the benchmark could be run in 10 consecutive VM invocations without allocation stalls. For both the minimal heap and stall-free heap the same VM paramaters were used as in our performance benchmarking except DaCapo iterations were limited to one, assuming that the greatest memory pressure occurs in the first iteration. The result is shown in Table 1.

SPECjbb2015 increases its workload progressively until the Service-level Agreement (SLA) cannot be maintained. Thus, it does not have a minimal heap size. We evaluate it with 16 and 32 GB heaps. With 16 GB, concurrent collectors cannot keep up with the allocation rate and numerous allocation stalls can be observed. With a 32 GB heap, allocation stalls are substantially reduced or fully eliminated. This demonstrates that these two heap sizes show different characteristics of the GC concerning the underlying time-space trade-off. For SPECjbb2015, which uses much larger heaps than the other benchmarks, we use large pages so that the JVM uses 2 MB OS pages instead of the default 4 kB pages. This reduces the translation lookaside buffer pressure.

[5] *tradesoap* also supports *huge* input size. However, we encountered crashes similar to the description in an open issue (https://github.com/dacapobench/dacapobench/issues/113) of DaCapo, so in the end we were not able to include it. *tomcat* also supports a *huge* input size, but we were not able to run without crashes. We believe it is due to being built for Java 5.

[6] In later JDK versions than 15, some GCs can dynamically update the number of threads with regard to an applications' footprint on resources

### 4.5 Data Collection

For each benchmark and heap size, we record execution time, number of GC cycles, number of allocation stalls, and cache misses. Cache misses are obtained using `perf` with the `cache-misses` event. (This event uses the counter for L2 data and instruction misses on our CPU.) Number of GC cycles are obtained using the built-in JVM logging. Allocation stalls is only available with the built-in JVM logging in ZGC-based GC. Except for the execution time and GC cycles in DaCapo, our metrics are for the entire JVM process.

To measure how many MB were moved, we added telemetry code in ZGC, HCSGC and LR-ZGC. While ZGC and HCSGC can utilize existing information, it comes at an extra cost for LR-ZGC, as movement of objects in the LOC set will trigger calls to an atomic add. Calls to atomic add could negatively impact performance due to cache invalidations and increased amount of loads from memory. Thus, the volume of moved data is collected in separate runs from our other results. Since HCSGC and LR-ZGC only target small pages we only include measurements for these kinds of pages. To facilitate comparability between the plots where metrics are compared on each axis, *i.e.,* Figs. 5 to 7, all of these are plotted from runs where the extra telemetry was enabled.

For JGraphT, we launch the JVM 11 times for each benchmark and discard the first execution to eliminate JVM warm-up noise (*e.g.,* loading libraries from disk). We run SPECjbb-2015 10 times and use its built-in metrics for latency performance (critical jOPS) and throughput performance (max jOPS). Since SPECjbb2015 is a long-running benchmark (each run >2 hours), there is no need to eliminate warm-up noise.

Following previous work [22, 32, 33, 36], we discard initial iterations and use repeated VM invocations to facilitate obtaining statistically significant results of execution time. In this context, statistical significance means that the confidence intervals is either sufficiently small, or does not appear to be reducing with additional VM invocations such that the null-hypothesis can be confirmed/rejected. We use 10 VM invocations, where each invocation is configured to run 10 iterations of the benchmark. The first 5 iterations are treated as warm-ups and we use the results of the remaining 5 iterations for analysis of execution time and GC cycles. The other metrics are gathered from all runs. We established sufficiently low variance by running 10 VM invocations.

Statistical significance is established using bootstrapping (using 10*k* bootstrap samples) with a 95% confidence interval akin to [32, 33, 36]; if intervals are non-overlapping, the means are different. Error bars are used to show the confidence intervals. A cross on the interval specifies the mean.

**Table 1.** Heap sizes in MB for DaCapo and JGraphT

| | | Heap sizes in evaluation | | | |
|---|---|---|---|---|---|
| Benchmark | 1× | 1.5× | 3× | 6× | stall-free |
| avrora | 8 | 12 | 24 | 48 | 117 |
| fop | 32 | 48 | 96 | 192 | 190 |
| h2 | 992 | 1488 | 2976 | 5952 | 3000 |
| jython | 20 | 30 | 60 | 120 | 2100 |
| luindex | 8 | 12 | 24 | 48 | 100 |
| lusearch-fix | 8 | 12 | 24 | 48 | 1450 |
| pmd | 50 | 75 | 150 | 300 | 480 |
| sunflow | 16 | 24 | 48 | 96 | 1020 |
| tradebeans | 236 | 354 | 708 | 1416 | 1170 |
| xalan | 16 | 24 | 48 | 96 | 1100 |
| CC uk | 312 | 468 | 936 | 1872 | 600 |
| CC enwiki | 320 | 480 | 960 | 1920 | 330 |
| MC uk | 136 | 204 | 408 | 816 | 380 |
| MC enwiki | 320 | 480 | 960 | 1920 | 400 |

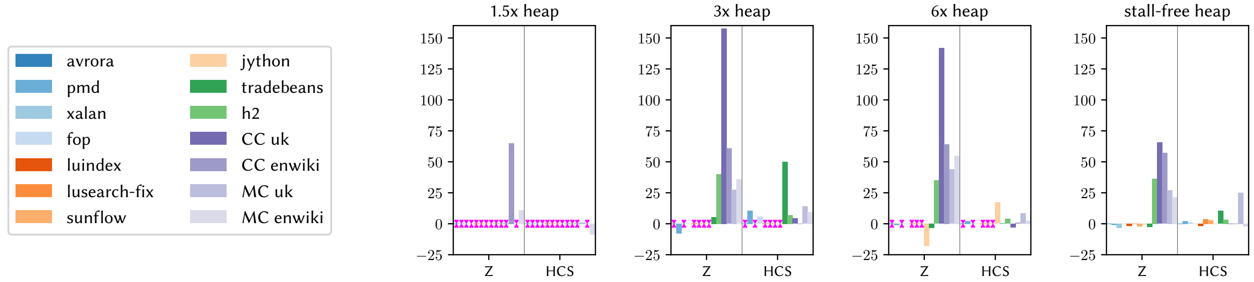
## 5 Results

We compare the results of G1, Parallel, Shenandoah, ZGC, HCSGC and LR-ZGC. When possible, we include results with and without compressed oops, as compressed oops also improve cache utilisation by avoiding storing unused address bits in the cache. We analyse 11 different applications, using 4 different configurations for JGraphT, with 4 different heap sizes. The symbol ⧗ indicates that the configuration could not run due to out of memory. Note that for Fig. 3 if neither a bar nor ⧗ is not present this means that there were no difference, and in SPECjbb2015 missing bars are due to lack of support for compressed oops for heaps ≥32 GB.

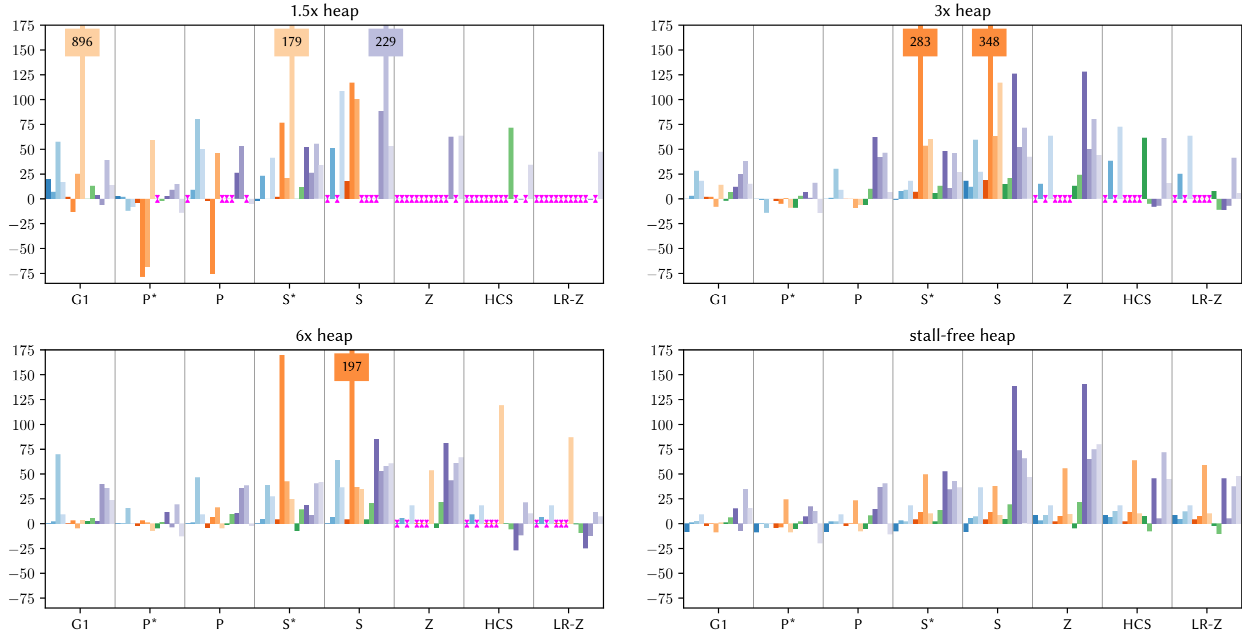
Execution time is measured as a proxy for throughput in DaCapo and JGraphT. SPECjbb2015 has a built-in metric called max jOPS. We expect a reduction in cache misses for LR-ZGC compared to HCSGC from a reduction in moved objects, a reduction in GC cycles (for each GC cycle, the heap is partially/fully traversed, leading to many cache misses), or because LR can facilitate moving hot objects by mutators to a higher degree due to its LOC set (§3). As LR does not need to inflate the EC set, we can therefore expect the number of cold objects copied in each GC cycles to be reduced. We measure how much data is copied for the entire VM invocation and expect this ranking: ZGC < LR-ZGC < HCSGC.

SPECjbb2015 tries to find the maximum workload that the machine can process for a given SLA. SPECjbb2015 has a 1% survivor rate per GC cycle [33] and that HCSGC was not able to establish an improvement in SPECjbb2015. HCSGC also showed that selecting all pages for evacuation in a benchmark that tries to saturate the machine is bad for performance as GC threads will have to compete with mutators for CPU time. Therefore, we expect that improvements over HCSGC will be driven by the reduction in copying of cold objects rather than a reduction in cache misses.

Configurations using compressed oops are marked by '*'. Absolute numbers with accompanying confidence intervals can be found in supplemental tables in the Appendix. In the plots we denote G1 GC as G1, Parallel GC as P, Shenandoah as S, HCSGC as HCS and LR-ZGC as LR-Z.




Execution time normalised to LR-ZGC. Lower is better.




Execution time normalised to G1*. Lower is better.

**Figure 3.** Throughput results for DaCapo and JGraphT. * = compressed oops was enabled. ⧗ = out of memory

### 5.1 Resource Usage

Table 2 shows that HCSGC and LR-ZGC experience stalls in ZGC's stall-free heap. LR-ZGC almost eliminates allocation stalls compared to HCSGC in the benchmarks where we expect LR-ZGC and HCSGC to perform well (JGraphT, tradebeans and h2). While there is still a few allocation stalls for tradebeans, the rate at which they occur is reduced by 95%.

Fig. 7 shows that LR-ZGC copies substantially less MB than HCSGC. The difference is the amount of cold objects that are no longer unnecessarily copied. In conclusion, LR-ZGC seems to require less resources than HCSGC.

**Table 2.** Allocation stalls per second when running DaCapo and JGraphT with stall-free heap.

| Benchmark | Z | HCS | LR-Z | Benchmark | Z | HCS | LR-Z |
|---|---|---|---|---|---|---|---|
| avrora | 0 | 0.0 | 0.0 | sunflow | 0 | 6.6 | 9.7 |
| fop | 0 | 126.7 | 117.3 | tradebeans | 0 | 124.1 | 5.9 |
| h2 | 0 | 0.0 | 0.0 | xalan | 0 | 246.0 | 249.4 |
| jython | 0 | 0.5 | 0.3 | CC uk | 0 | 0.0 | 0.1 |
| luindex | 0 | 0.0 | 0.0 | CC enwiki | 0 | 0.6 | 0.6 |
| lusearch-fix | 0 | 325.0 | 324.3 | MC uk | 0 | 7.4 | 0.5 |
| pmd | 0 | 57.4 | 52.0 | MC enwiki | 0 | 6.5 | 0.0 |

### 5.2 DaCapo

With a stall-free heap LR-ZGC outperforms ZGC and HCSGC in h2 by 36% and 3%, respectively. Recall that each GC cycle is inducing a substantial amount of cache misses as the object graph is traversed to determine reachability. LR-ZGC has a substantial amount of additional GC cycles compared to ZGC (which is natural since, they have different selection for GC work). This however, drives up the amount of cache misses and it is hard to determine if the speed-up in LR-ZGC compared to ZGC for h2 is caused by a reduction of cache misses and/or more frequent GC cycles. While a small regression is seen in LR-ZGC in tradebeans compared to ZGC

it is not statistically significant, and compared to HCSGC LR-ZGC is 10% faster. A contributing factor for the speed-up compared to HCSGC in tradebeans is the 95% reduction of allocation stalls. Additionally, in Fig. 5, the cache misses are reduced in LR-ZGC compared to HCSGC. The fact that LR-ZGC has substantially more GC cycles than ZGC and HCSGC (Fig. 6) and that LR-ZGC copies only slightly more MB than ZGC (Fig. 7). This suggests that another important factor of the improvement compared to HCSGC is an improved object layout without the need of copying cold objects.

For the remaining benchmarks, LR-ZGC had one statistically signifiant difference compared to ZGC, a regression in xalan (3.2%), and one statistically signifiant difference compared to HCSGC, an improvement in pmd (2.2%). Compared to HCSGC, LR-ZGC has only two regressions, one in MC enwiki (1.5×) and one in CC uk (6×). Comparing LR-ZGC to the rest of the collectors in OpenJDK, it provides the fastest results for h2, and for tradebeans ties with ZGC and Parallel as best GC (with and without compressed oops). For remaining applications in DaCapo, for all heap sizes, LR-ZGC, along with HCSGC, ZGC and often Shenandoah, falls behind the other collectors. It is unsurprising that concurrent collectors that generally optimise for latency do not perform as well with respect to throughput. It is therefore noteworthy that LR-ZGC or HCSGC is the best performing collector with respect to throughput in some benchmarks.

While in 1.5× heap, ZGC, HCSGC, and LR-ZGC struggle to run most benchmarks, reporting mostly out of memory, one interesting observation is that HCSGC can run h2 while ZGC and LR-ZGC cannot. We believe that this is because HCSGC defragments the entire heap every cycle. This also keeps floating garbage at a minimum allowing HCSGC to operate with a small minimum heap. At 3× all JGraphT applications, pmd, fop, tradebeans and h2 are able to run. At 6×, also jython can run. The remaining, avrora, xalan, luindex, lusearch-fix and sunflow are only able to run in the stall-free heap highlighting how ZGC, HCSGC, and LR-ZGC trade increased memory and lower throughput for low latency.

### 5.3 JGraphT

At 1.5× heap, ZGC, HCSGC, and LR-ZGC can only run CC enwiki and MC enwiki, and reports out of memory for the rest. As soon as the heaps are large enough for the GCs to run, LR-ZGC beats ZGC on all four configurations (10–158%). LR-ZGC is outperforming HCSGC in 6 configurations (1–25%), on-par in 6 configurations and is slower than HCSGC in 2 configurations (3–9%). The largest speed up for LR-ZGC compared to HCSGC is found in the stall-free heap and the largest regression compared to HCSGC is found in the 1.5× heap. That HCSGC can outperform LR-ZGC in smaller heaps is aligned with our findings for h2. The other regression compared to HCSGC is found in CC uk for the 6× heap, which is also the largest heap that is evaluated for this benchmark. Examining Fig. 5 for CC uk 6× heap, HCSGC has 4% fewer

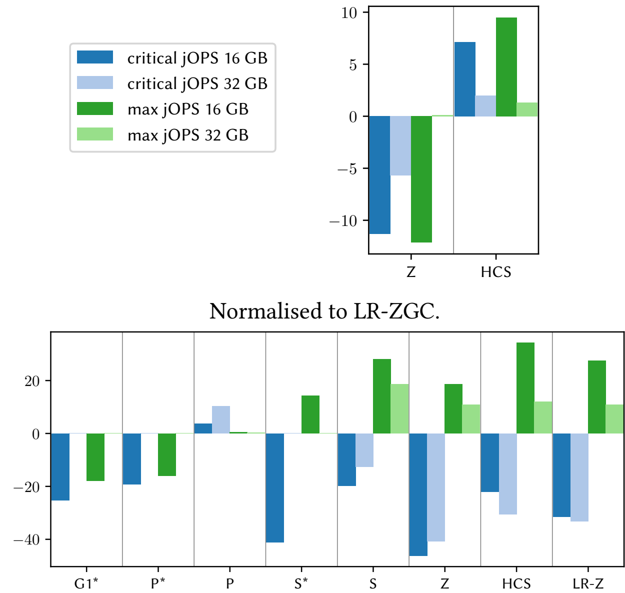



**Figure 4.** SPECjbb2015 results normalised to G1. * = compressed oops enabled. Missing bars are due to compressed oops are not available in heaps ≥ 32 GB. Lower is better.

cache misses than LR-ZGC. (Admittedly, this difference can be hard to see due to the scale of the axis, so we defer the reader to Table 58d in the Appendix.) However, Fig. 6 shows that HCSGC has the least amount of GC cycles, suggesting that defragmenting the entire heap at this heap size reduces the need for future GC work. It is a combination of improving object layout using the load-barrier and defragmenting the entire heap that gives HCSGC a slight edge over LR-ZGC in this particular case. We speculate that the heap was large enough and allocation rate was low enough to hide the extra cost of defragmenting the entire memory on each GC cycle.

Comparing LR-ZGC to the rest of the collectors in OpenJDK, LR-ZGC is outperforming ZGC and Shenandoah, but beaten by G1 and Parallel in CC uk. In, CC enwiki LR-ZGC is also outperforming Parallel, to only be beaten by G1. In MC enwiki LR-ZGC is on-par with Shenandoah without compressed oops, but beaten by G1, Parallel and Shenandoah with compressed oops. For MC uk, LR-ZGC provides a substantial boost over HCSGC, which put it at third place, in a tie with G1/Parallel (both without compressed oops).

### 5.4 SPECjbb2015

With respect to throughput, reported by SPECjbb2015 as max jOPS, G1 and Parallel are the GCs with the highest throughput in both heap sizes (Fig. 4). With respect to latency, reported as critical jOPS, ZGC is the best-performing GC. This is unsurprising. HCSGC and LR-ZGC perform substantially worse than ZGC at 16 GB, and are both outperformed by S*. HCSGC is additionally outperformed by G1*. This is in

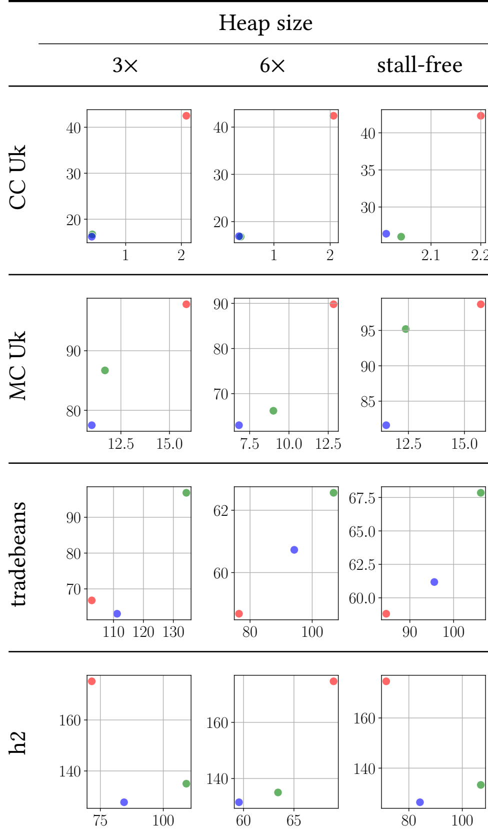



**Figure 5.** Execution time in seconds (Y axis) *vs* $10^9$ L2 misses (X axis). ● Z ● HCS ● LR-Z.

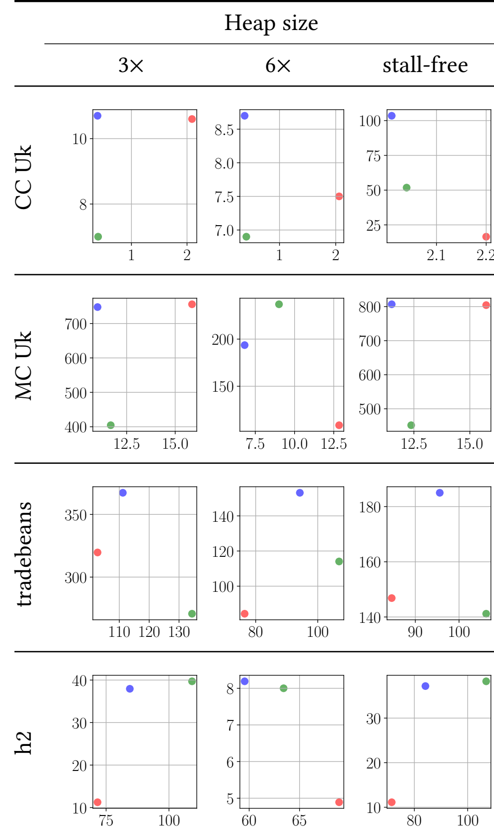



**Figure 6.** GC cycles (Y axis) *vs* $10^9$ L2 cache misses (X axis). ● Z ● HCS ● LR-Z.

line with our expectations that HCSGC and LR-ZGC are bad fits for benchmarks like SPECjbb2015, that saturates the machine and whose typical survivor rate is <1%, meaning there are few objects whose location can be improved, or that are accessed after being moved. LR-ZGC outperforms HCSGC on both scores in both heaps. HCSGC's regression compared with ZGC is likely due to mutator-driven movement of predominantly cold objects due to the low survivor rate. We believe that the results are a strong indication that LR-ZGC shows that less work (due to the LOC set) for GC threads is important in a saturated machine when GC threads and mutators compete for CPU resources.

### 5.5 Compressed Oops

Enabling compressed oops can have a substantial improvement in throughput and latency. For instance in SPECjbb2015, comparing each collector with and without compressed oops, G1, Parallel and Shenandoah improved its throughput by 17–19% and its latency by 18–25%. Most benchmarks exhibit a substantial boost in performance when enabling compressed oops for the respective GC. Despite lacking support for compressed oops, ZGC is the best-performing GC with respect to latency in SPECjbb2015 (Fig. 4).

## 6 Related Work

Courts [13] developed an adaptive memory management algorithm that increased throughput by locality improvements in Lisp. The locality was for paging from disk, although they had a 128 kB cache on the machine. This work was implemented in a generational GC. During garbage collection, mutators and GC threads competed to move objects. In the case that the mutator won, it moved the object to a specific region and placed objects in access order. Additionally, objects were rearranged during program execution for improved locality using a load barrier with customised hardware support. Live objects that were not currently used were moved to a separate region of the heap. Different configurations of the design were evaluated on 4MB and 8MB heaps respectively with performance improvements as high as 61%–108%.

Huang et al. [18] use a combination of load barriers and static analysis (to reduce the number of places instrumented by load barriers) to deliver online object reordering, OOR, in the context of the GenCopy STW copying collector in JikesRVM. OOR uses adaptive sampling driven by load barriers to find the hot fields of objects and uses this information to influence the traversal order of GC threads moving objects

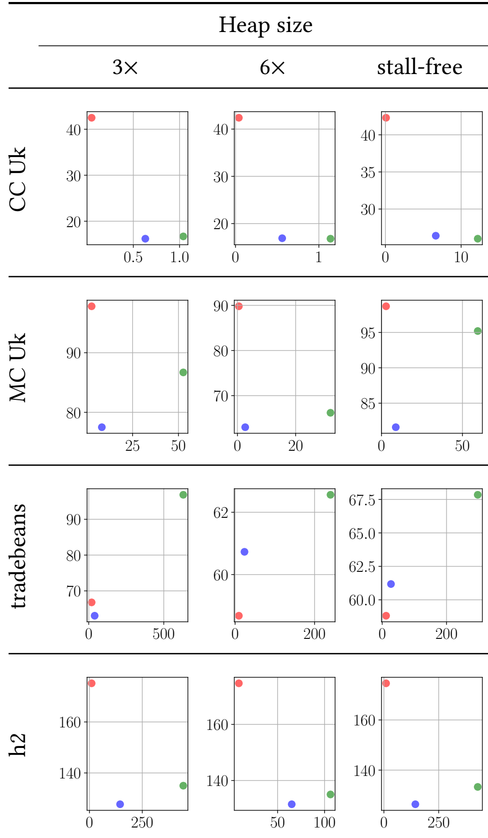



**Figure 7.** Exec time in seconds (Y axis) *vs* MB moved (X axis). ● Z ● HCS ● LR-Z.

in the heap to produce a to-space with improved cache locality. OOR's load barriers are only used to collect sampling data. A comparison with OOR (implemented on-top of ZGC) would therefore be interesting.

Chilimbi and Larus [11] use online profiling to construct an object affinity graph, where nodes are weighted by temporal affinity. They also provide a copying algorithm that uses that graph to place objects with high affinity together. The evaluation shows that such placement can reduce cache miss rates between 21% and 42% and improve application performance between 14% and 37%. In contrast, LR does not need such graph to be constructed, requiring less memory overhead and reorganise objects according to the access order, assuming temporal and spatial locality.

A compelling difference between OOR and the system of Chilimbi and Larus [11] and LR is that the former will only move objects close in memory if they are connected in the object graph through fields, whereas LR will move objects close if they are accessed close in time. In other words, the previous system used topological information to place objects suitably for fast mutator access, whereas LR draws this information from the mutator itself, like Courts [13].

Chen et al. [8] improve locality by profiling hot objects between garbage collection cycles. The hotness information is used during the garbage collection cycle to aggregate hot objects to each partition of the heap. They also explore the implications of doing layout optimisations independent of a garbage collection cycle. Furthermore, the order of the hot objects will be "according to a hierarchical decomposition order based on their inherent structural relationship". This design was evaluated on a generational GC in the Common Language Runtime, version 2.0, using six C# programs used internally at Microsoft due to a general lack of benchmarks at the time, on a single-core Pentium 4 with 1 GB RAM. For these programs, the technique saw a 17% average improvement of execution time prompting the authors to argue that garbage collection should be viewed first as an opportunity to improve the layout and second to reclaim memory. LR similarly overlays layout optimisation on-top of GC balancing improved locality with additional fragmentation.

## 7 Conclusion

We proposed LR, a design that can automatically improve the cache locality of Java programs by using mutators to moving hot objects in access order. Our design addresses the inherent tradeoff in previous work [33] between trying to ensure mutators move as many hot objects as possible and avoiding unnecessary movement of cold objects. We accomplish this by permitting mutators to move hot objects without requiring the remaining cold objects on the page to be moved. This design avoids extra GC movement of cold objects, allowing LR to achieve both goals simultaneously. To do so efficiently, LR stores hot object forwarding information on the heap, which avoids increasing the amount of forwarding storage needed for the moved hot objects.

Our implementation on-top ZGC outperforms ZGC in 18 configurations by 5–158%. Performance regression occurs in 7 configurations in the range of 3–17% and 12 is on-par. Compared to HCSGC, LR-ZGC outperforms its performance in 18 configurations by 1–50%, have 3 regressions (OOM, 3%, 9%) and 17 configurations where they are on par. These results show that LR solves the fundamental tradeoff in HCSGC's design and allows us to obtain the locality-improving properties without the need of full heap compaction on each cycle. Although, in one case this design permitted HCSGC to run with a smaller heap than both ZGC and LR-ZGC.

## Acknowledgments

We thank the anonymous reviewers whose feedback helped to greatly improve this paper.

This research was supported by the Swedish Research Council through the project Accelerating Managed Languages (2020-05346), by the Swedish Foundation for Strategic Research through the project Deploying Memory Management Research in the Mainstream (SM19-0059), and by donations from Oracle Corporation.

## References


- [1] Diab Abuaiadh, Yoav Ossia, Erez Petrank, and Uri Silbershtein. 2004. An efficient parallel heap compaction algorithm. (2004), 224–236. https://doi.org/10.1145/1028976.1028995
- [2] Andrew W. Appel. 1987. Garbage collection can be faster than stack allocation. *Inform. Process. Lett.* 25, 4 (1987), 275–279. https://doi.org/10.1016/0020-0190(87)90175-X
- [3] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. 2004. Myths and realities: the performance impact of garbage collection. *SIGMETRICS Perform. Eval. Rev.* 32, 1 (jun 2004), 25–36. https://doi.org/10.1145/1012888.1005693
- [4] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06)*. ACM, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488
- [5] Stephen M. Blackburn and Antony L. Hosking. 2004. Barriers: friend or foe?. In *Proceedings of the 4th International Symposium on Memory Management* (Vancouver, BC, Canada) *(ISMM '04)*. Association for Computing Machinery, New York, NY, USA, 143–151. https://doi.org/10.1145/1029873.1029891
- [6] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *Proceedings of the 20th International Conference on World Wide Web* (Hyderabad, India) *(WWW '11)*. Association for Computing Machinery, New York, NY, USA, 587–596. https://doi.org/10.1145/1963405.1963488
- [7] P. Boldi and S. Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proceedings of the 13th International Conference on World Wide Web* (New York, NY, USA) *(WWW '04)*. Association for Computing Machinery, New York, NY, USA, 595–602. https://doi.org/10.1145/988672.988752
- [8] Wen-ke Chen, Sanjay Bhansali, Trishul Chilimbi, Xiaofeng Gao, and Weihaw Chuang. 2006. Profile-Guided Proactive Garbage Collection for Locality Optimization. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. Association for Computing Machinery, New York, NY, USA, 332–340. https://doi.org/10.1145/1133981.1134021
- [9] C. J. Cheney. 1970. A nonrecursive list compacting algorithm. *Commun. ACM* 13, 11 (nov 1970), 677–678. https://doi.org/10.1145/362790.362798
- [10] Trishul M. Chilimbi, Bob Davidson, and James R. Larus. 1999. Cache-conscious structure definition. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, USA) *(PLDI '99)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/301618.301635
- [11] Trishul M. Chilimbi and James R. Larus. 1998. Using generational garbage collection to implement cache-conscious data placement. *SIGPLAN Not.* 34, 3 (oct 1998), 37–48. https://doi.org/10.1145/301589.286865
- [12] Cliff Click, Gil Tene, and Michael Wolf. 2005. The Pauseless GC Algorithm. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments* (Chicago, IL, USA) *(VEE '05)*. ACM, New York, NY, USA, 46–56. https://doi.org/10.1145/1064979.1064988
- [13] Robert Courts. 1988. Improving Locality of Reference in a Garbage-collecting Memory Management System. *Commun. ACM* 31, 9 (Sept. 1988), 1128–1138. https://doi.org/10.1145/48529.48536
- [14] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. 2004. Garbage-First Garbage Collection. In *Proceedings of the 4th International Symposium on Memory Management* (Vancouver, BC, Canada) *(ISMM '04)*. Association for Computing Machinery, New York, NY, USA, 37–48. https://doi.org/10.1145/1029873.1029879
- [15] Marcel Dombrowski, Konstantin Nasartschuk, Kenneth B. Kent, and Gerhard W. Dueck. 2015. A survey on object cache locality in automated memory management systems. In *2015 IEEE 28th Canadian Conference on Electrical and Computer Engineering (CCECE)*. 349–354. https://doi.org/10.1109/CCECE.2015.7129301
- [16] Christine H. Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. 2016. Shenandoah: An open-source concurrent compacting garbage collector for OpenJDK. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, Lugano, Switzerland, August 29 - September 2, 2016*. 13:1–13:9. https://doi.org/10.1145/2972206.2972210
- [17] John Hopcroft and Robert Tarjan. 1973. Algorithm 447: Efficient Algorithms for Graph Manipulation. *Commun. ACM* 16, 6 (June 1973), 372–378. https://doi.org/10.1145/362248.362272
- [18] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J Eliot B. Moss, Zhenlin Wang, and Perry Cheng. 2004. The garbage collection advantage: improving program locality. (2004), 69–80. https://doi.org/10.1145/1028976.1028983
- [19] JGraphT [n. d.]. https://jgrapht.org/
- [20] Richard Jones, Antony Hosking, and Eliot Moss. 2023. *The Garbage Collection Handbook: The Art of Automatic Memory Management* (second edition. ed.). CRC Press, Boca Raton, FL.
- [21] Tomas Kalibera, Matthew Mole, Richard Jones, and Jan Vitek. 2012. A Black-Box Approach to Understanding Concurrency in DaCapo. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Tucson, Arizona, USA) *(OOPSLA '12)*. Association for Computing Machinery, New York, NY, USA, 335–354. https://doi.org/10.1145/2384616.2384641
- [22] Jonas Norlinder, Erik Österlund, and Tobias Wrigstad. 2022. Compressed Forwarding Tables Reconsidered. In *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes* (Brussels, Belgium) *(MPLR '22)*. Association for Computing Machinery, New York, NY, USA, 45–63. https://doi.org/10.1145/3546918.3546928
- [23] Jonas Norlinder, Tobias Wrigstad, David Black-Schaffer, and Albert Mingkun Yang. 2024. *Mutator-Driven Object Placement using Load Barriers (code)*. https://doi.org/10.5281/zenodo.13119225
- [24] Erez Petrank and Dror Rawitz. 2002. The hardness of cache conscious data placement. (2002), 101–112. https://doi.org/10.1145/503272.503283
- [25] Shai Rubin and Chilimbi = Trishul = Bodí, k = Rastislav = and. 2002. An efficient profile-analysis framework for data-layout optimizations. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon) *(POPL '02)*. Association for Computing Machinery, New York, NY, USA, 140–153. https://doi.org/10.1145/503272.503287
- [26] Ram Samudrala and John Moult. 1998. A graph-theoretic algorithm for comparative modeling of protein structure11Edited by F. Cohen. *Journal of Molecular Biology* 279, 1 (1998), 287 – 302. https://doi.org/10.1006/jmbi.1998.1689
- [27] SPECjbb2015 [n. d.]. Standard Performance Evaluation Corporation. https://www.spec.org/jbb2015/
- [28] Gil Tene, Balaji Iyengar, and Michael Wolf. 2011. C4: The Continuously Concurrent Compacting Collector. In *Proceedings of the International Symposium on Memory Management (ISMM '11)*. ACM, New York, NY, USA, 79–88. https://doi.org/10.1145/1993478.1993491
- [29] Kris Venstermans, Lieven Eeckhout, and Koen De Bosschere. 2006. 64-bit versus 32-bit Virtual Machines for Java. *Software: Practice and Experience* 36, 1 (2006), 1–26. https://doi.org/10.1002/spe.679

[30] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. 1991. Effective "static-graph" reorganization to improve locality in garbage-collected systems. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '91)*. Association for Computing Machinery, New York, NY, USA, 177–191. https://doi.org/10.1145/113445.113461

[31] Wm Wulf and Sally A. McKee. 1994. *Hitting the Memory Wall: Implications of the Obvious.* Technical Report. USA.

[32] Albert Mingkun Yang, Erik Österlund, Jesper Wilhelmsson, Hanna Nyblom, and Tobias Wrigstad. 2020. ThinGC: Complete Isolation with Marginal Overhead. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management* (London, UK) *(ISMM 2020)*. Association for Computing Machinery, New York, NY, USA, 74–86. https://doi.org/10.1145/3381898.3397213

[33] Albert Mingkun Yang, Erik Österlund, and Tobias Wrigstad. 2020. Improving Program Locality in the GC Using Hotness. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 301–313. https://doi.org/10.1145/3385412.3385977

[34] Albert Mingkun Yang and Tobias Wrigstad. 2022. Deep Dive into ZGC: A Modern Garbage Collector in OpenJDK. *ACM Trans. Program. Lang. Syst.* 44, 4, Article 22 (sep 2022), 34 pages. https://doi.org/10.1145/3538532

[35] Xi Yang, Stephen M. Blackburn, Daniel Frampton, and Antony L. Hosking. 2012. Barriers reconsidered, friendlier still! (2012), 37–48. https://doi.org/10.1145/2258996.2259004

[36] Wenyu Zhao, Stephen M. Blackburn, and Kathryn S. McKinley. 2022. Low-latency, high-throughput garbage collection. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 76–91. https://doi.org/10.1145/3519939.3523440

Received 2024-05-25; accepted 2024-06-24

# Interactive Programming for Microcontrollers by Offloading Dynamic Incremental Compilation


Fumika Mochizuki
fumika.maejima@csg.ci.i.u-tokyo.ac.jp
The University of Tokyo
Tokyo, Japan

Tetsuro Yamazaki
yamazaki@csg.ci.i.u-tokyo.ac.jp
The University of Tokyo
Tokyo, Japan

Shigeru Chiba
chiba@csg.ci.i.u-tokyo.ac.jp
The University of Tokyo
Tokyo, Japan



## Abstract

Interactive execution environments are suitable for trial-and-error basis programming for microcontrollers. However, they are mostly implemented as interpreters to meet microcontrollers' limited memory size and demands for portability. Hence, their execution performance is not sufficiently high. In this paper, we propose offloading dynamic incremental compilation and linking to a host computer connected to a microcontroller. Since the computing resources of the host computer are sufficient to execute incremental dynamic compilation, they are used to enhance the relatively poor computing resources of the microcontroller. To show the feasibility of this idea, we design a small programming language named *BlueScript* and implement its interactive execution environment. Our experiment reveals that BlueScript executes a program one to two orders of magnitude faster than MicroPython, while its interactivity is comparable to that of MicroPython despite using dynamic incremental compilation.



***CCS Concepts:*** • **Software and its engineering** → *Dynamic compilers*; **Incremental compilers**; • **Computer systems organization** → Embedded software.

***Keywords:*** Interactive Programming, Dynamic Compilation, Embedded systems, Microcontrollers, TypeScript

**ACM Reference Format:**
Fumika Mochizuki, Tetsuro Yamazaki, and Shigeru Chiba. 2024. Interactive Programming for Microcontrollers by Offloading Dynamic Incremental Compilation. In *Proceedings of the 21st ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '24), September 19, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3679007.3685062



Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.



*MPLR '24, September 19, 2024, Vienna, Austria*



© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.



ACM ISBN 979-8-4007-1118-3/24/09
https://doi.org/10.1145/3679007.3685062


## 1 Introduction

An interactive execution environment of programming language, or a REPL (Read-eval-print loop), is known as being suitable for trial-and-error basis programming. For artificial intelligence and data science, Jupyter [22] is widely used as such an interactive environment in Python. For programming education, Scratch [17] is a popular interactive language and environment. A web browser such as Google Chrome provides an interactive environment in JavaScript for debugging.

In contrast, in the field of microcontroller programming, a non-interactive development environment has been mainly adopted. A program is written in the C or C++ language, statically compiled, and linked on a host machine/computer. Then, the compiled binary is written to a flash memory of a target microcontroller connected through a serial cable, and the microcontroller is rebooted to execute the program. When the behavior of the program is not satisfactory, programmers or developers repeat this linear process from the beginning. This traditional development process is still dominant, but interactive environments are becoming popular even for programming a microcontroller. Interpreter-based environments, such as MicroPython [3] and Espruino [23], have been actively developed and used. They provide a REPL, and their users can enjoy their interactivity from a host machine connected to a microcontroller.

However, such interactive environments for microcontrollers are often slow since they are interpreters. An interpreter is suitable for interactive programming. It can be implemented to run with a small amount of memory and keep portability independent of CPU architecture. This is an advantage but implies low execution performance. Dynamic incremental compilation, or dynamic compilation, would be a solution but it damages the size of an interpreter's memory footprint and portability. It would also increase response time and worsen interactivity due to the limited computing power of microcontrollers.

To address this problem, we propose to exploit the computing resources of a host machine connected to a microcontroller particularly through a wireless network such as Bluetooth. Usually, a microcontroller is connected to a host machine or computer during software development. We offload dynamic incremental compilation and linking to this

host machine. To investigate our idea in a practical interactive environment for microcontrollers, we design a small programming language named *BlueScript* and implement its interactive execution environment for Espressif Systems' ESP32 microcontroller. Although we use an off-the-shelf C compiler for ESP32 as a backend native-code generator to reduce development costs, our execution environment achieves smooth interactivity. The execution of BlueScript programs is only up to 10 times slower than C programs and one to two orders of magnitude faster than MicroPython's ones. Our contributions are twofold. One is to illustrate that offloading dynamic incremental compilation and linking still enables interactive programming for microcontrollers while significantly improving execution performance. The other is to implement a prototype of the interactive execution environment for BlueScript and design experiments to assess its interactivity.

## 2 Interactive Programming for Microcontrollers

Even when programming a microcontroller, a trial-and-error basis development is effective and useful, and it is enabled by interactive execution environments of programming languages. Suppose that we are programming a microcontroller for controlling a toy car. We first use the C language and its traditional non-interactive environment. This toy car is equipped with two motors and a front camera. Since the two motors are separately controlled and connected to two rear wheels, the car can move forward and backward and change its direction of travel. We write a program so that the car will periodically take a photo and turn to the right when it finds a red object in front of it, for example, a red plastic ball. Listing 1 is an example of this program. It is written in the C language. The `main` function registers an event handler `periodic_task`, which is invoked every 500 msec. The event handler takes a front photo by calling `take_picture`, and calls `find_red_object` to count the number of red pixels and determine whether a red object is found in the given photo image. If a red object is found, the car turns to the right to avoid collision. Otherwise, the car keeps going straight ahead.

The calls to `set_speed` in line 14, 18, and 19 change the motor speed. The second argument specifies the speed. Programmers have to carefully choose an appropriate value for the second argument so that the car will run smoothly. To choose it, programmers might want to take a trial-and-error approach. They might write a program with some initial values for that second argument, compile it, and run it to move the car. Then, they might observe the car moving and change the second argument to new values. They might iterate these steps until they find good values for the second argument and the car runs nicely. Iterating these steps is, however, time-consuming and tedious. For example, these steps take

```
1 bool find_red_object(uint16_t* img) {
2     int red_count = 0;
3     for (int row = 0; row < HEIGHT; row++) {
4         for (int col = 0; col < WIDTH; col++) {
5             int hue = get_hue(img[row*col+col]);
6             if (hue > 45)
7                 red_count++;
8         }
9     }
10    return red_count > RED_MINIMUM;
11 }
12
13 void go_straight() {
14     set_speed(BOTH, 80);
15 }
16
17 void go_right() {
18     set_speed(LEFT, 50);
19     set_speed(RIGHT, 10);
20 }
21
22 void periodic_task() {
23     uint16_t* img = take_picture();
24     if (find_red(img))
25         go_right();
26     else
27         go_straight();
28 }
29
30 void main() {
31     timer_t timer;
32     timer_create(&periodic_task, &timer);
33     timer_start_periodic(timer, 500);
34 }
```

**Listing 1.** Controlling a toy car

several seconds to a minute when we use ESP-IDF (Espressif IoT Development Framework) [7], which is the standard development environment provided by Espressif for the ESP32 microcontroller. It is slow to build executable binary code, write it into the flash memory of the microcontroller, and reboot the microcontroller for restarting a program.

An interactive execution environment mitigates this inefficiency. For example, MicroPython is available on the ESP32 microcontroller. It provides a REPL accessible from a host computer connected to the microcontroller through a serial cable or Wi-Fi network. Through this REPL, programmers can give code fragments after the prompt one by one to interactively define functions, redefine them, and run them. Listing 2 shows a log of programming a toy car through the MicroPython REPL. >>> and `...` in Listing 2 are prompts. Program texts following them are sent to the REPL. A programmer first defines four functions one by one (Line 1-12) and run them by registering `periodic_task` as a timer-event handler (Line 14-15). Then the programmer observes the move of the toy car. Since he/she thinks that the car moves

```
1  >>> def find_red_object(img):
2  ...     # omit
3  >>>
4  >>> def go_straight():
5  ...     set_speed(BOTH, 80)
6  >>>
7  >>> def go_right():
8  ...     set_speed(LEFT, 50)
9  ...     set_speed(RIGHT, 10)
10 >>>
11 >>> def periodic_task():
12 ...     # omit
13 >>>
14 >>> timer = Timer()
15 >>> timer.init(mode=Timer.PERIODIC, callback=
        periodic_task, period=500)
16 >>> # Check behavior of toy car.
17 >>>
18 >>> def go_straight(): # Redefine the function.
19 ...     set_speed(BOTH, 40)
20 >>> # Check behavior of toy car again.
```

**Listing 2.** Programming a toy car through the REPL in MicroPython

too fast, he/she redefines the `go_straight` function to pass a smaller value `40` to `set_speed` (Line 19-20). If `go_straight` did not pass a numeric literal but the value of a global variable such as `SPEED`, the programmer would not redefine a function but simply change the value of that global variable. This change is also easy through a REPL. This redefinition is immediately reflected on the car's move, and the programmer can observe its effect. If the car's move is not satisfactory, the programmer can redefine `go_straight` again. He/She can iterate these steps until the move is satisfactory. This iteration is not tedious or time-consuming.

A problem of interactive execution environments for microcontrollers is their execution speed. They often use a simple interpreter since only a limited amount of memory is equipped on a microcontroller. Their execution speed is much slower than the native machine code compiled from a program written in the C language. For example, the function `find_red_object` in the C language takes 0.45 seconds, but a MicroPython function equivalent to `find_red_object` takes 11.5 seconds on the ESP32 microcontroller, which operates at 240 MHz. The MicroPython function would be too slow to change the car's direction of travel before it collides with a red object in front of it. The travel speed of the toy car must be significantly reduced.

#### Dynamic Incremental Compilation

A promising approach to accelerating the execution speed in an interactive execution environment is dynamic incremental compilation, which can trace its origin back to the 1970s [15]. Its idea is to *incrementally* compile only a new code fragment interactively given by programmers and *dynamically* link the compiled binary to the rest of the codebase that already exists. This approach has been sophisticated [10, 25], and it is now widely adopted by a number of language virtual machines as dynamic compilation or *just-in-time* compilation, where a code fragment is dynamically selected for compilation not only when it is given by programmers but also when it is recognized as frequently executed code, so called *hotspot*, during runtime. Furthermore, modern dynamic compilers adaptively change how aggressively they optimize code to match a trade-off between compilation time and resulting speedup.

However, dynamic incremental compilation is not widely used by interactive execution environments for microcontrollers. Some execution environments like MicroPython [4] support it, but its capability is limited since the environments must contain an optimizing compiler and a linker. These components increase the memory footprint of the environment, but this increase of memory footprint is not accepted without careful consideration. A microcontroller is equipped with only a limited amount of memory. The SRAM size of the ESP32 microcontroller is 520 KB, and that of the RP2040 microcontroller for Raspberry Pi Pico is only 264 KB. These memories must be shared with application programs. Note that, from the viewpoint of product design, the total memory footprint should be reduced as much as possible to minimize product cost.

This viewpoint is also applied to the code size of execution environments. Although those microcontrollers support up to 16 MB flash memory, where the executable binary code is stored, it must be shared with applications, and hence, a smaller footprint of the execution environment is also more desirable. For example, the binary size of the V8 JavaScript engine for macOS is 1.5 times bigger than the V8 engine without dynamic compilation. Also, the Ruby virtual machine supporting the YJIT dynamic compiler is 1.3 times bigger than the original one for macOS.

Furthermore, some developers might be afraid that dynamic incremental compilation would degrade the interactivity of execution environments for microcontrollers. Since the clock speed of microcontrollers is an order of magnitude slower than high-performance processors for smartphones or server computers, it is challenging to dynamically compile code during runtime to keep a short response time.

## 3 Offloading Dynamic Incremental Compilation

To utilize incremental dynamic compilation in interactive execution environments on microcontrollers, we propose offloading it onto a host machine that is a computer used by programmers as a console to access a microcontroller. The host machine is connected to the microcontroller through a

wireless network or a serial cable, and it has larger computing resources than the microcontroller. Since these computing resources are sufficient to execute incremental dynamic compilation, they are used for enhancing relatively poor computing resources of the microcontroller.

We offload not only compilation but also linking on a host machine. This differs from a traditional approach where source code is compiled into a shared object or a dynamic library on a host machine and is dynamically loaded and linked on a target machine by `dlopen`. We offload all the steps except writing executable code on the memory of a microcontroller and reduce the memory footprint of an execution environment on a microcontroller as much as possible.

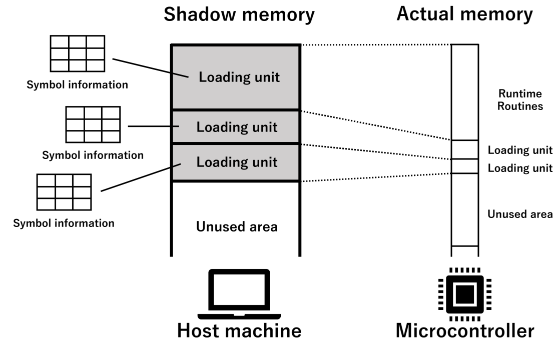



**Figure 1.** Shadow memory

### 3.1 Overview

An interactive execution environment that offloads incremental dynamic compilation consists of two components. One component runs on a host machine and provides a REPL and a compiler. The other one runs on a microcontroller, and it is a collection of runtime routines that are needed for receiving compiled native code from the host machine and executing it. These two components are connected through a network, which is a Bluetooth wireless network in the case of our prototype mentioned later. Before rebooting the component on a microcontroller, its runtime routines are compiled, built, and written onto the flash memory of the microcontroller. The compiled runtime routines are sent from the host machine through a serial cable to the microcontroller. After rebooting, the serial cable is not necessary if the two components are connected through a wireless network.

When a programmer gives a new source-code fragment through a REPL on a host machine, a compiler on the host machine compiles it into native code for a microcontroller. We call this native code *a loading unit*. The compiler performs *incremental* compilation. If necessary, it refers to code fragments previously given by the programmer. To reduce engineering efforts, we use an existing compiler tool-chain for cross-compilation to the target microcontroller. Our prototype mentioned later uses an existing C compiler as a backend. Our compiler translates source code into a C program, and a C compiler compiles it into native code in the ELF (Executable and Linkable Format). Using an existing off-the-shelf compiler indirectly improves the portability for different microcontrollers.

Then, the compiled native code, or a loading unit, is modified on the host machine to be linked to the rest of the code. After being linked, this loading unit is sent *as it is* to the microcontroller, and it is written to memory on the microcontroller. Finally, a runtime routine calls the entry point of that loading unit to execute it. After finishing the execution, the runtime routine waits until another loading unit is sent from the host computer. The REPL also waits until the programmer gives a new code fragment.

### 3.2 Shadow Memory

To execute linking on a host machine, we maintain an abstract memory image, which we call *shadow memory*, on a host machine (Figure 1). It is an abstract copy of the memory image on a microcontroller, and it also holds symbol information needed for linking compiled native code to the rest of the native code that is already running on a microcontroller.

The shadow memory consists of several regions, which correspond to memory regions on a microcontroller. The shadow memory initially consists of regions that represent unused free areas on a microcontroller's memory. Later, those unused areas will be gradually converted into regions containing compiled native code that we call *a loading unit*. The shadow memory also initially includes a region that corresponds to the memory region containing the runtime routines of the execution environment on a microcontroller. It may be a memory region on the flash memory of a microcontroller, where executable code is written before rebooting. This region in the shadow memory holds a symbol table that indicates the addresses of the entry points of the runtime routines.

A loading unit, which is native code obtained by compiling a source-code fragment interactively given by a programmer through a REPL, is linked on the shadow memory. First, a new region is allocated for this loading unit from an unused free area. A new memory region is allocated on a microcontroller, and a corresponding region is created in the shadow memory on a host machine. The loading unit is copied into this new region in the shadow memory on a host machine. The symbol information of the loading unit is attached to the region. Because a loading unit in our prototype system is executable binary in the ELF, a symbol table stored in the ELF section `.symtab` and a relocation table in the ELF section starting with `.rela` are attached to the region. A symbol table is a map that associates a symbol name with its address, which consists of a section name and an offset from the beginning of the section. A relocation table lists the locations of unresolved addresses in a section, for example,

```
1 let led: GPIO = new GPIO(23);
2 let isLit: boolean = false;
3
4 function toggleLED() {
5     led.set(isLit ? 0 : 1);
6     isLit = !isLit;
7 }
8
9 setInterval(toggleLED, 500);
```

**Listing 3.** A program in BlueScript for blinking an LED

the text section for the relocation table in the .rela.text section. Its entries are an offset, a symbol name, how a value is computed, and so on.

Then, the loading unit is modified in the shadow memory to be linked. All the entries in the relocation tables are resolved, and computed values are stored at their addresses in the region in the shadow memory on a host machine. During this symbol resolution, symbol information for other loading units in their regions are referred to. After this symbol resolution, the resulting machine code in the shadow memory is sent to a microcontroller, and it is written in the corresponding memory region on the microcontroller. When the machine code in other regions is updated, those new values are also sent to a microcontroller, and they overwrite previous values.

Note that the symbol information is never sent to a microcontroller. It is just kept in the shadow memory on a host machine so that it will be reused later when a new loading unit comes in to be linked or when an existing loading unit is modified for further optimization.

### 3.3 The BlueScript Language

As a prototype for exploring our idea of offloading, we have developed a new small language named *BlueScript*. We use this language for exploring our idea in a practical interactive environment for microcontrollers. The design goal of BlueScript is to balance sufficient flexibility for interactive programming and adequate execution performance on performance-poor microcontrollers. BlueScript borrows syntax from TypeScript, although it only uses a subset of TypeScript's syntax. Listing 3 is an example of a BlueScript program. It can be read mostly as a normal TypeScript program, but the semantics of BlueScript are more static than TypeScript. Most of the dynamic features available in TypeScript are not supported in BlueScript. For example, BlueScript does not support eval, apply, prototype-based inheritance, or structural typing. BlueScript adopts nominal typing and supports classes with simple single inheritance as well as function redefinition, but forbids redefining existing classes.

The language adopts simple gradual typing [24]. It supports primitive types such as 32-bit signed integer, 32-bit floating-point number, boolean, and a character string. Their type names are integer (or number), float, boolean, and string, respectively. Unlike TypeScript, an integer and a floating-point number are distinguished and treated as different primitive types. The language also supports an array and an instance of a class. The current class system of BlueScript is static and provides minimal functionality. It only supports single inheritance, and an interface is not supported. A class type is nominally treated, and subtype relations are determined by considering only type names. Adding a new method or field is currently not supported.

Since simple gradual typing is adopted, the language also supports any type. A value of any can be any type of value. The type checker of BlueScript performs type inference, and it infers any type when no other types are determined because, for example, a type annotation is not given by a programmer. If necessary, the type checker may treat an expression (or a variable) of any type as being compatible with any other type of expression, and vice versa. Since this implies implicit type conversion at runtime, this may raise a runtime type error.

When an element type is not explicitly given, an array object is created as an array of any type; its element type is any. The index of an array element must be integer type. When an array element is accessed, it is checked at runtime whether a given index is inside the boundaries of the array. Hence, an array access may raise a runtime error. BlueScript also provides arrays of primitive types, such as an array of integer or float. All the elements have the same type.

A function is declared by a function declaration starting with function or by the arrow notation => as in TypeScript. Currently, however, a function expression cannot capture a local variable. It cannot form a closure. The statements currently supported by BlueScript are expression statements, block statements, and const, let, if, while, for, and return statements. They also include non-labeled break and continue. for...in or for...of statements are not supported. try...catch statements or async functions are not supported either.

Unlike TypeScript, BlueScript currently does not provide a module system. Neither import nor require is available. BlueScript provides only a single global name scope where several built-in library functions and classes, such as GPIO and PWM, are available for accessing hardware components as listed in Table 1.

**Table 1.** Built-in classes and functions

| | |
|---|---|
| GPIO class | Control general-purpose input/output ports. |
| PWM class | Access a pulse width modulation controller. |
| Display class | control a display (M5Stack only). |
| Timer functions | setInterval, clearInterval, setTimeout, and clearTimeout. |
| Button function | buttonOnPressed. |

**Figure 2.** A notebook-style REPL for BlueScript

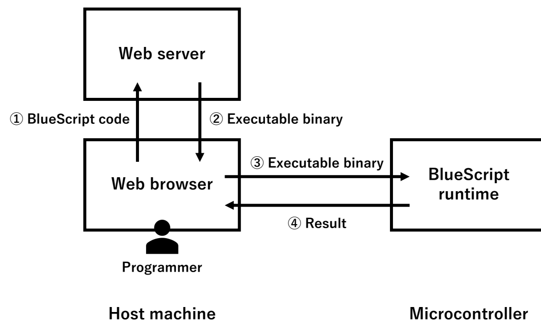



**Figure 3.** The interactive execution environment for BlueScript.

### 3.4 Implementation

We have implemented an interactive execution environment for the BlueScript language. The target microcontroller is ESP32, a 32bit microcontroller by Espressif Systems. This execution environment provides a notebook-style REPL similar to the Jupyter notebook [1]. As shown in Figure 2, a programmer can write a source-code fragment in a rectangular cell in a web page in a web browser on a host machine. Then, when the programmer clicks on the run button to the left of the cell, the code written in the cell is executed on an ESP32 microcontroller connected to the host machine through Bluetooth. The output of the execution is sent back to the web browser, and it is displayed on the right side of the web page. If the output is an error message, it is displayed under the cell. The wireless communication by Bluetooth between a host machine and a microcontroller may make it easy to program a microcontroller that controls the driving of a toy car, the flight of a drone, or the motion of a robot since we do not have to connect it to a host machine by a limited length of wire.

As already mentioned in Section 3.1, the execution environment consists of two components: one on a host machine and the other on a microcontroller. See Figure 3. The component on a host machine runs on Node.js and works as a web server that serves a web page for a notebook-style REPL. The web browser is responsible for the communication between a host machine and a microcontroller. Since it uses the Web Bluetooth API, a programmer must use a web browser supporting this API such as Google Chrome. The component for a microcontroller runs on FreeRTOS with libraries provided by ESP-IDF (Espressif IoT Development Framework).

**Compilation.** The BlueScript compiler is included in the host-machine component of the execution environment. It is implemented in TypeScript and runs on Node.js. It uses the Babel parser[2] for parsing a BlueScript program. A non-supported statement or expression is accepted by Babel but it is treated as an error after parsing. Since a BlueScript program is incrementally given to the compiler, the compiler receives a source-code fragment one by one and translates it into a program in the C language. The compiler maintains a global name table so that functions and global variables can be accessed from other code fragments given later. Then, that C program is compiled with the `-O2` option by a cross-compiler provided by ESP-IDF, which is based on the GNU C compiler. The compiler generates executable binary code in the ELF. After being linked, it is sent to ESP32 and written in an executable RAM area of the device. The data section generated by the compiler is written in a non-executable RAM area. Note that ESP32 partly uses the Harvard memory architecture.

A function in BlueScript is translated into a function in the C language. Currently, this function in the C language is indirectly invoked so that redefinitions of a function in BlueScript can be easily implemented. Only the calls to a function bound to a `const` variable are compiled into direct function calls in the C language. The top-level statements included in a given source-code fragment are collected and translated into the body of one function in the C language. This function is the entry point of the loading unit generated from the given fragment. It is invoked when the loading unit is written to the memory on a microcontroller.

The `integer` type and the `boolean` type in BlueScript are translated into the `int32_t` type in the C language. The `float` type in BlueScript is translated into the `float` type. A `string`, an array, and an instance of a class are objects in BlueScript. They are translated into a heap array in the C language. The first 32 bits of this heap array are the header. The upper 30 bits in the header are used as a pointer to the type (or class) information of that object. 2 bits in the header are used as mark bits during garbage collection. The rest of the elements of the heap array are used to store a property or an array element in BlueScript. The memory layout in the heap array is statically determined for its type or class in BlueScript.

The type system of BlueScript guarantees that a statically typed expression (or variable) results in a value of (a subtype

[1] https://jupyter.org

[2] https://github.com/babel/babel

of) that type when that expression is evaluated at runtime if no type error is reported during compilation. Thus, the C program that BlueScript code is translated into does not include runtime type checking except an expression of the any type. It runs without runtime penalties for type checking.

**Gradual Typing.** The any type is compatible with any type. An expression statically typed as the any type can be used where an expression of any other type is expected, and vice versa. For example, it can be used where an integer expression is expected. On the other hand, an integer expression can be used where an any expression is expected. Thus, when an expression of the any type is translated where an expression of other types *t* is expected, a runtime type check is inserted to assure that the value is of that type *t*. When the type of an expression is not any and it is translated where an expression of the any type is expected, that expression is wrapped in type conversion into any. For example, this BlueScript code:

```
const a: any = 3
const b: integer = a + 4
```

is translated into C code equivalent to the following pseudo code:

```
lvars[0] = int_to_any(3)
int32_t b = any_to_int(add_any(lvars[0],
                               int_to_any(4)))
```

where lvars is an array holding the values of local variables that are pointers or any-type values. int_to_any and any_to_int are functions to convert an integer value into an any value and vice versa. add_any is a function to add two any values. Since the left operand of + is statically typed as any, the right operand is converted into any.

A value of the any type is implemented by a classic technique called *pointer tagging*. It is a 32bit pointer to a heap array, but the lower 2 bits are used as a tag. An integer value and a float value are packed into the upper 30 bits with a tag for identifying their types. Thus, when integer is converted into any, its value is cast from 32 bits to 30 bits. When float is converted into any, its precision is reduced from 32 bits to 30 bits. The exponent loses 2 bits and becomes 6 bits after the conversion. Furthermore, when a value is stored in an array or an instance of a class, the value is converted into an any-type value. Only when a value is stored in an integer, float, or byte array, the value is stored as it is. No runtime overhead for type conversion implies.

**Garbage Collection.** The current execution environment for BlueScript provides a mark-and-sweep garbage collector. It runs on a microcontroller. 2 bits in an object header are used as mark bits for garbage collection. The garbage collection root is implemented by a technique called *shadow stack* [13]. During the marking phase, an object is colored black, white, or gray. A gray object is an object that is reachable from the root but not yet scanned. The reachability from this gray object is not examined. A black object is an object that is reachable from the root and scanned. The other objects are white. The collector first performs depth-first-traversal to first color objects gray and then black. If a stack overflows, the collector stops the traversal. It scans the whole heap memory from the top to the bottom, and whenever it finds a gray object, it restarts the depth-first-traversal from that gray object until no gray object is found [16].

The garbage collection for BlueScript can be interrupted. When a hardware interrupt occurs during the marking phase, the tri-color marker is interrupted, and an interrupt handler is invoked. While the interrupt handler is running, when a pointer to a white object is stored in a black object, that white object is changed to gray and pushed into the marker's stack. This is performed by a write barrier inserted before every store operation.

The garbage collector currently reclaims only the memory occupied by an object. It does not reclaim the memory occupied by executable native code. Extending the garbage collector to reclaim the memory occupied by executable native code is our future work.

**Libraries.** BlueScript provides a built-in library. To control hardware, programmers can use classes and functions listed in Table 1. The setInterval function repeatedly calls a given function at specified intervals. It adds the given function to a table, and the functions in the table are periodically called by a separate thread of FreeRTOS. Since this thread is bound to the same core as the main application thread, when that thread is executing the function passed to setInterval, the main application thread is suspended. The functions passed to setInterval and the main program are mutually exclusive in BlueScript.

## 4 Experiments

We use our prototype implementation of BlueScript language to conduct experiments. We compare the execution speed, the response time, and the code size of the language's execution environment on a microcontroller, between BlueScript, MicroPython, Espruino, and the C language. Espruino is a JavaScript virtual machine for microcontrollers. Our research question is whether or not our idea of offloading incremental dynamic compilation achieves both smooth interaction and fast execution speed on modern personal computers and microcontrollers. We use MicroPython V1.19.1 and Espruino 2V20 for experiments. The C compiler is xtensa-esp32-elf-gcc (crosstool-NG esp-2022r1) 11.2.0. As a target microcontroller, we use M5Stack Fire. It is an IoT development kit based on the ESP32-D0WDQ6 microcontroller with 520 KB of RAM, 16 MB of Flash memory, and 8 MB of PSRAM. As a host machine, we use MacBook Pro, which is equipped with Apple M1 Pro, 16 GB of memory, and 512 GB of storage.

**Table 2.** Benchmarks programs for execution times (the upper programs are from “Are we fast yet?” and the lower ones are from ProgLangComp)

| Name | LOC | Description |
|---|---|---|
| Bounce | 93 | Simulates a ball bouncing within a box. |
| List | 79 | Recursively creates and traverses lists. |
| Mandelbrot | 83 | Calculates the classic fractal. |
| NBody | 185 | Simulates the movement of planets in the solar system. |
| Permute | 42 | Generates permutations of an array. |
| Queens | 61 | Solves the eight queens problem. |
| Sieve | 38 | Finds prime numbers based on the sieve of Eratosthenes. |
| Storage | 57 | Creates and verifies a tree of arrays to stress the garbage collector. |
| Towers | 83 | Solves the Towers of Hanoi game. |
| Biquad | 43 | Converts waveform (floating point) by a digital biquad filter. |
| FIR | 55 | Converts waveform (floating point) by an FIR filter. |
| CRC | 113 | Computes CRC32-IEEE checksums using a precomputed table. |
| FFT | 97 | Computes FFT (fixed point, complex pair of int16). |
| SHA256 | 175 | Computes SHA256 hashes. |

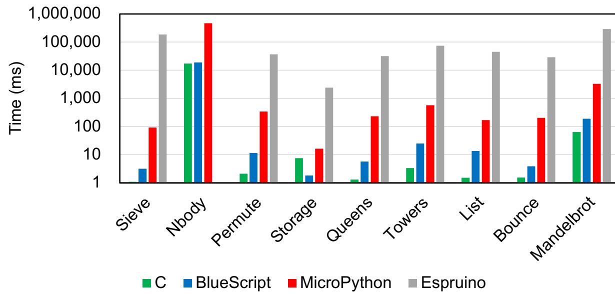



**Figure 4.** Execution time of the “Are we fast yet?” benchmark suite

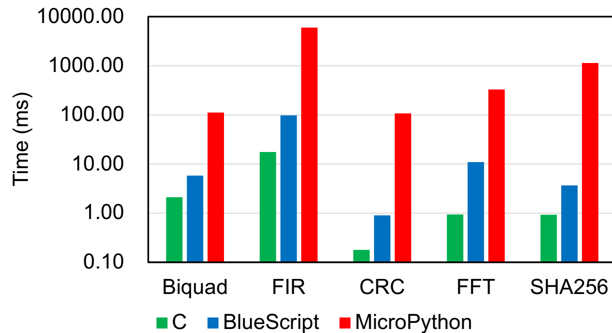



**Figure 5.** Execution time of the ProgLangComp benchmark suite

### 4.1 Execution Time

A BlueScript function equivalent to `find_red_object` in Listing 1 runs in 0.57 seconds, while the original function in the C language runs in 0.45 seconds, and the MicroPython equivalent runs in 11.5 seconds. We also measure the execution time of programs taken from two benchmark suites listed in Table 2, and compare the times among BlueScript, MicroPython, Espruino, and C. One benchmark suite is the “Are we fast yet?” benchmark suite [19][3]. This benchmark suite was developed to compare execution performance among various languages. We use its micro benchmark programs in Python and JavaScript and write equivalent BlueScript and C versions. The other is the ProgLangComp benchmark suite [21][4]. It is a collection of programs written for evaluating the performance of the ESP32 microcontroller. They perform signal processing and hash functions, which are regarded as typical computation by microcontrollers. We use MicroPython programs from this suite and write equivalent C and BlueScript programs. We fully type-annotate all the BlueScript programs used for the experiment.

For BlueScript, we compile and link a whole BlueScript program and runtime routines all at once on a host machine. The resulting executable binary is written on flash memory before a microcontroller is booted to run. For MicroPython and Espruino, we run benchmark programs according to the documents on their official pages. We download their virtual machines from the official page and install them on the microcontroller before rebooting. Then, we write each benchmark program on the host machine and send and execute it to the microcontroller. For the C language, we use the ESP-IDF build tool to compile and execute benchmark programs. We give the `-O2` option to the compiler.

Figure 4 shows the execution times of the “Are we fast yet?” benchmark suite. We compare C, BlueScript, MicroPython, and Espruino. The execution of Nbody by Espruino is timeout. Figure 5 shows the execution times of the ProgLangComp benchmark suite. For this, we compare C, BlueScript, and MicroPython. All the execution times are the means of five runs. Note that the Y axes are log scales.

[3]https://github.com/smarr/are-we-fast-yet
[4]https://github.com/ignasp/ProgLangComp_onESP32

```
1 // Code fragment #1
2 let dspl = new Display();
3 let colorWhite = dspl.color(255, 255, 255);
4 let colorRed = dspl.color(255, 0, 0);
5 dspl.showIcon(dspl.ICON_HEART, colorRed,colorWhite
      );
6 print("$$");
7
8 // Code fragment #2
9 dspl.showIcon(dspl.ICON_HEART, colorRed,colorWhite
      );
10 dspl.fill(colorWhite);
11 print("$$");
12
13 // Code fragment #3
14 let isSmall = false;
15 setInterval(() => {
16   if (isSmall) {
17     dspl.showIcon(dspl.ICON_SMALL_HEART, colorRed,
           colorWhite);
18   } else {
19     dspl.showIcon(dspl.ICON_HEART, colorRed,
          colorWhite);
20   }
21   isSmall = !isSmall;
22 }, 500);
23 print("$$");
```

**Listing 4.** The Flashing Heart program in BlueScript

The figures show that BlueScript is one to two orders of magnitude faster than MicroPython and three to four magnitudes faster than Espruino. The performance difference is much larger when a program involves a large number of arithmetic operations on primitive types. Such programs are Bounce, FIR, CRC, and SHA256. This result is because the BlueScript programs are type-annotated and thus they less frequently perform runtime type checks and boxing/unboxing. Compared to the C language, BlueScript is up to 10 times slower. The slowdown is significant when a program involves a large number of function calls since they are indirectly invoked in BlueScript. Such programs are Towers and List.

### 4.2 Interactivity

We evaluate the interactivity of BlueScript by comparing it to MicroPython and the C language. We incrementally write a program step by step according to a given scenario, and measure the response time for every step.

**Benchmark Programs.** We develop a new benchmark suite for our experiment. We take five scenarios from a tutorial course on the website of MakeCode[5], which provides programming lessons using the micro:bit [8] microcontroller for beginners. We develop three versions of programs for each scenario in BlueScript, MicroPython, and the C language. Table 3 lists the benchmark programs we develop. In each scenario, a programmer writes a code fragment, and runs it to test its behavior step by step. Each scenario consists of three or four code fragments. The goal of each scenario is to display an icon and text on the screen. The target microcontroller is M5Stack Fire for all the scenarios. We implement a library in the C language to manipulate the screen and the buttons of M5Stack Fire. We make this library accessible from BlueScript and MicroPython through a similar interface as well as the C language. The microcontroller is connected to a host machine through Bluetooth for BlueScript, WiFi for MicroPython, and a serial cable for the C language.

Listing 4 shows code fragments in BlueScript for the scenario Flashing Heart. This scenario consists of three code fragments. A programmer types and runs each code fragment step by step to finally build a small application program that shows a heart icon on the screen. The code fragment #1 initializes the display and shows a heart icon on the screen. The code fragment #2 shows a small heart icon and erase it. The code fragment #3 calls `setInterval` so that the given function will be repeatedly called and the size of the icon will change every 500 msec. Every code finally prints a text to notify the end of the execution to a host machine.

For MicroPython, the code in each code fragment is transformed into a single line since the REPL of MicroPython receives only a single line at once. We combine all the statements in the code fragment into a single line where statements are separated by a semicolon. Since the execution environment ESP-IDF for the C language, does not directly support interactive programming, we execute the previous code fragment again whenever we execute the new code fragment. For example, when we execute the code in the third fragment, we combine the first, second and third fragments into a single program, compile it, and execute it. This is because we must reboot M5Stack Fire to start a new program, and hence, all the devices must be initialized again after rebooting.

**Results.** We execute the code fragments of each scenario three times and measure the response time. It is the elapsed time since we press the button to start executing a code fragment till a string `$$` is printed as an execution result on the screen. Every code fragment prints `$$` at the end as shown in Listing 4. We also measure the execution time, excluding compilation time and communication time from the response time. For BlueScript and the C language, the times shown in figures are the means of the three runs of each scenario. For MicroPython, those are the shortest ones because of their instability and large distribution.

Figure 6 shows the execution time of every code fragment. Since every code fragment is very short and it just calls a few library functions only, the majority of the execution time is the time for executing library functions. Thus, we

[5]See https://makecode.com. It is open source under MIT license, see https://github.com/microsoft/pxt

**Table 3.** Benchmark programs for interactivity

| Scenario | Fragment | LOC | Description |
|---|---|---|---|
| Dice | 1 | 3 | Bind an empty function to button B by using `buttonOnPressed` built-in function. |
| | 2 | 8 | Initializes the display module and bind a function for displaying 0 to button B. |
| | 3 | 5 | Bind a function for displaying a random integer from 0 to 10 to button B. |
| | 4 | 5 | Bind a function for displaying a random integer from 0 to 6 to button B. |
| Flashing Heart | 1 | 5 | Initialize the display module, set global variables to color code, and then display a heart icon. |
| | 2 | 3 | Display the same heart icon again, and then erase it. Reuse the display module and color code. |
| | 3 | 10 | Use `setInterval` function to display a large heart and a small heart alternately. |
| Love Meter | 1 | 9 | Initialize the display module and bind a function for displaying a random integer 0-10 to button B. |
| | 2 | 5 | Bind a function for displaying a random integer from 0 to 100 to button B. |
| | 3 | 6 | Display “Love meter” and bind a function for displaying a random integer from 0 to 100. |
| Name Tag | 1 | 6 | Initialize the display module, set global variables to color code, and then display “My name is:”. |
| | 2 | 2 | Display “Sara!.”. |
| | 3 | 2 | Display “My age is:”. |
| | 4 | 2 | Display an integer 9. |
| Smiley Button | 1 | 3 | Bind an empty function to button B by using `buttonOnPressed` built-in function. |
| | 2 | 7 | Initialize the display module and bind a function for displaying a happy face icon to button B. |
| | 3 | 5 | Bind a function for displaying a sad face icon to button C. |

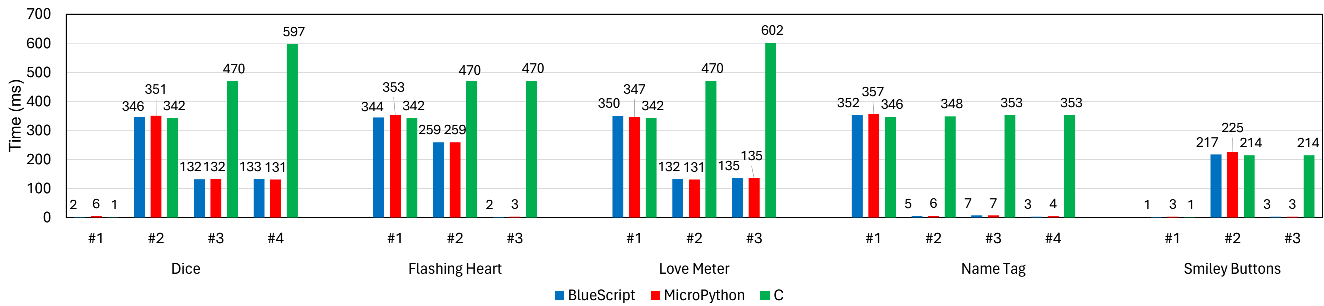



**Figure 6.** Execution times of every code fragment

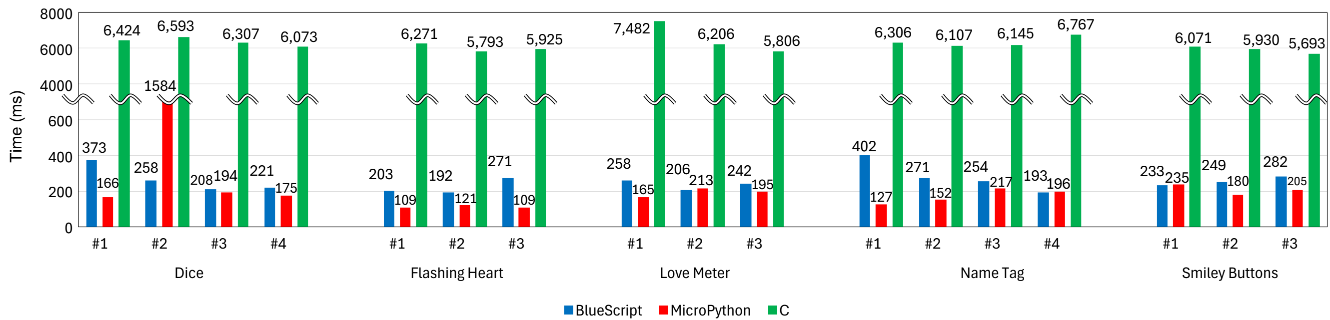



**Figure 7.** The time of compilation and others for every code fragment in BlueScript, MicroPython and C

do not observe notable differences between the three languages. The execution times of fragment #3 and #4 in the C language are significantly slow. This is because we execute the fragment #1 and #2 (and #3) again when we execute the fragment #3 (or #4). The readers might think that this is odd and unfair, but this is typical software development by using a non-interactive environment for the C language. Whenever we change a program and test it, we must reboot a microcontroller and run the program again from the beginning.

Figure 7 shows the times for compilation and others excluding execution. The times include communication between a host machine and a microcontroller. They are calculated by subtracting the net execution time from the total

**Table 4.** The size of runtime routines on a microcontroller

| Environment | Bluetooth | WiFi | Size (MB) |
|---|---|---|---|
| BlueScript | ✓ | - | 0.86 |
| | - | - | 0.27 |
| MicroPython | ✓ | ✓ | 1.64 |
| | - | ✓ | 1.40 |
| | ✓ | - | 1.26 |
| | - | - | 0.97 |

response time. The time of fragment #2 for the scenario Flashing Heart in MicroPython is extremely long for an unknown reason, which we are still investigating. The time is 1584 ms, which is significantly longer than the time for BlueScript. In other cases, BlueScript is up to 3.2 times (the mean is 1.6) slower than MicroPython since BlueScript performs incremental dynamic compilation. However, the times in BlueScript are approximately less than 300 msec. despite the use of a relatively slow off-the-shelf C compiler as a backend native-code generator. It would be slower than a dedicated highly-tuned dynamic compiler. Those times are still acceptable for interactive programming. According to the literature [20], the users feel that a response is immediate when it takes 0.5 to 1 seconds. The times for compilation and others in the C language are an order of magnitude longer than the times in the other two languages. Those times are approximately 6 seconds. They include compilation, linkage of all object code including libraries and an operating system, and writing the resulting executable binary code to flash memory of a target microcontroller. The C language is not acceptable for interactive programming.

### 4.3 Runtime Size

Table 4 lists the memory size of runtime routines on a microcontroller. For BlueScript, it is 0.86 MB but is 0.27 MB if the Bluetooth library is excluded. For MicroPython, the size of its virtual machine is 1.64 MB but is 1.40 MB if the Bluetooth library is excluded. It is 1.26 MB if the Bluetooth library is included but the WiFi library is excluded. The size of the virtual machine without Bluetooth or Wifi libraries is 0.97 MB. However, that virtual machine still includes a number of libraries for accessing hardware. Note that these runtime routines and the code of the virtual machine are stored in flash memory but not in a limited size of SRAM.

## 5 Related Work

Espruino's "`compiled`" tag [11] is most relevant to our work. The source code of a function with this tag is sent to a remote web service. It is compiled there into native code and sent back to a target microcontroller. Since this dynamic incremental compilation is supported only for official Espruino boards, it is not used for the experiment in Section 4.1. A difference from our work is that Espruino's compilation has only limited capability to link compiled native code. A global variable is not linked, and hence Espruino searches the symbol table when it accesses a global variable.

MicroPython provides two dynamic incremental compilers: the Native code emitter and the Viper code emitter [4]. They run on microcontrollers. If a MicroPython function is decorated with `native` or `viper`, it is compiled into native code when its declaration is evaluated. Since those compilers run on microcontrollers, their compilation capability is limited. They support only a subset of the language, and the compiled native code must be compatible with the byte-code interpreter for passing and returning a value beyond function boundaries. Thus, the performance improvement is limited. According to our experiment using the benchmark programs in Section 4.1, the Viper code emitter improves the execution performance of the benchmark programs only by a factor of up to 4.6. Recall that BlueScript runs one or two orders of magnitude faster than MicroPython.

LuaRTOS [14] provides an interactive shell and dynamic incremental compilation similar to the Native code emitter of MicroPython. uLisp [5] also provides an interactive shell for microcontrollers. Although it does not provide dynamic incremental compilation, it supports an inline assembler.

There have been several research activities [9, 18, 26], where the techniques for dynamic compilers are applied to virtual machines running on microcontrollers, but the design and implementation of such compilers is still a challenging topic. Our work proposes a different design of dynamic compiler targeting interactive environments for microcontrollers, rather than simply porting a dynamic compiler designed for high-performance processors to microcontrollers.

Warduino [12] is a WebAssembly (Wasm) virtual machine for microcontrollers. Like our work, Warduino utilizes the computing resources of a host machine to compensate the poor resources of microcontrollers. It offloads debugging capability to a host machine although ours offloads dynamic incremental compilation.

StaticTypeScript [2] is a subset of TypeScript, and its design is similar to our BlueScript. To achieve good execution performance on a micro:bit microcontroller, StaticTypeScript provides an offline compiler running in a web browser on a host machine. Unlike BlueScript, StaticTypeScript does not provide an interactive shell.

Contiki [6] uses a dynamic linker for reprogramming a wireless sensor node, which is run by a low-power microcontroller for energy efficiency. Contiki allows a memory-constrained sensor node to dynamically load and link native code modules. Unlike Contiki, BlueScript allows dynamic loading without dynamic linking on a microcontroller since linking is offloaded.

JTAG [1] and SWD[6] are debugging protocols for microcontrollers. They enable breakpoints, step execution, and reading/writing microcontrollers' memory. Compared to such debugging protocols, our shadow memory can be regarded as a software implementation of debugging protocol although it currently allows only reading and writing microcontroller's memory. Our shadow memory is, however, not only for debugging but also for interactive programming. Conversely, it would also be possible to implement shadow memory using JTAG.

## 6 Conclusion

We presented an interactive execution environment for our programming language *BlueScript*. To run on a microcontroller with a small amount of SRAM, this execution environment offloads dynamic incremental compilation and linking to a host machine that is a computer connected to that microcontroller and used by programmers to access it for programming. We illustrated that BlueScript programs run only up to 10 times slower than C programs and one to two orders of magnitude faster than MicroPython's ones. Our experiments showed that our execution environment responded within approximately less than 300 msec. excluding the execution time of a given BlueScript program, although it reuses an off-the-shelf C compiler to reduce its development costs and improve its portability. The source code of BlueScript is available on our website https://github.com/csg-tokyo/bluescript.

## Acknowledgments

This work was supported by JSPS KAKENHI Grant Numbers JP20H00578 and JP24H00688.

## References

[1] 2001. IEEE Standard Test Access Port and Boundary Scan Architecture. *IEEE Std 1149.1-2001* (2001), 1–212. https://doi.org/10.1109/IEEESTD.2001.92950
[2] Thomas Ball, Peli de Halleux, and Michał Moskal. 2019. Static TypeScript: An Implementation of a Static Compiler for the TypeScript Language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (Athens, Greece) *(MPLR 2019)*. Association for Computing Machinery, 105–116. https://doi.org/10.1145/3357390.3361032
[3] Damien P. George. 2014. MicroPython. https://micropython.org.
[4] Damien P. George and Paul Sokolovsky and contributors. 2014. Maximising MicroPython Speed. https://docs.micropython.org/en/v1.9.3/pyboard/reference/speed_python.html.
[5] David Johnson-Davies. 2016. uLisp. http://www.ulisp.com/.
[6] Adam Dunkels, Niclas Finne, Joakim Eriksson, and Thiemo Voigt. 2006. Run-time dynamic linking for reprogramming wireless sensor networks. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems* (Boulder, Colorado, USA) *(SenSys '06)*. Association for Computing Machinery, New York, NY, USA, 15–28. https://doi.org/10.1145/1182807.1182810
[7] Espressif Systems (Shanghai) Co., Ltd. 2016. ESP-IDF Programming Guide. https://docs.espressif.com/projects/esp-idf/en/stable/esp32/index.html.
[8] Micro:bit Educational Foundation. 2017. micro:bit. http://microbit.org/teach/.
[9] Giuseppe Di Giore, Antonella Di Stefano, Giovanni Morana, and Corrado Santoro. 2006. JIT compiler optimizations for stack-based processors in embedded platforms. In *Proceedings of the 4th International Workshop on Java Technologies for Real-Time and Embedded Systems* (Paris, France) *(JTRES '06)*. Association for Computing Machinery, New York, NY, USA, 212–217. https://doi.org/10.1145/1167999.1168034
[10] Adele Goldberg and David Robson. 1983. *Smalltalk-80: the language and its implementation.* Addison-Wesley Longman Publishing Co., Inc., USA.
[11] Gordon Williams. 2015. JavaScript Compilation. https://www.espruino.com/Compilation.
[12] Robbert Gurdeep Singh and Christophe Scholliers. 2019. WARDuino: A Dynamic WebAssembly Virtual Machine for Programming Microcontrollers. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (Athens, Greece) *(MPLR 2019)*. Association for Computing Machinery, 27–36. https://doi.org/10.1145/3357390.3361029
[13] Fergus Henderson. 2002. Accurate garbage collection in an uncooperative environment. In *Proceedings of the 3rd International Symposium on Memory Management* (Berlin, Germany) *(ISMM '02)*. Association for Computing Machinery, New York, NY, USA, 150–156. https://doi.org/10.1145/512429.512449
[14] Jaume Olivé Petrus. 2015. Lua RTOS. https://github.com/whitecatboard/Lua-RTOS-ESP32.
[15] Ronald L. Johnston. 1979. The Dynamic Incremental Compiler of APL 3000. In *Proceedings of the International Conference on APL: Part 1* (New York, New York, USA) *(APL '79)*. Association for Computing Machinery, New York, NY, USA, 82–87. https://doi.org/10.1145/800136.804442
[16] Richard Jones, Antony Hosking, and Eliot Moss. 2011. *The Garbage Collection Handbook: The Art of Automatic Memory Management* (1st ed.). Chapman & Hall/CRC.
[17] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch Programming Language and Environment. *ACM Trans. Comput. Educ.* 10, 4, Article 16 (nov 2010), 15 pages. https://doi.org/10.1145/1868358.1868363
[18] Geetha Manjunath and Venkatesh Krishnan. 2000. A small hybrid JIT for embedded systems. *SIGPLAN Not.* 35, 4 (apr 2000), 44–50. https://doi.org/10.1145/346443.346451
[19] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. 2016. Cross-Language Compiler Benchmarking: Are We Fast Yet?. In *Proceedings of the 12th Symposium on Dynamic Languages* (Amsterdam, Netherlands) *(DLS 2016)*. Association for Computing Machinery, 120–131. https://doi.org/10.1145/2989225.2989232
[20] Steven Seow Ph.D. 2008. *Designing and Engineering Time: The Psychology of Time Perception in Software* (1st ed.). Addison-Wesley Professional.
[21] Ignas Plauska, Agnius Liutkevičius, and Audronė Janavičiūtė. 2023. Performance Evaluation of C/C++, MicroPython, Rust and TinyGo Programming Languages on ESP32 Microcontroller. *Electronics* 12, 1 (2023). https://doi.org/10.3390/electronics12010143
[22] Project Jupyter. 2014. Jupyter. https://jupyter.org/.
[23] Pur3 Ltd. 2013. Espruino. https://www.espruino.com.
[24] Jeremy Siek and Walid Taha. 2006. Gradual typing for functional languages. *Scheme and Functional Programming*.
[25] David Ungar and Randall B. Smith. 2007. Self. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*

[6] https://developer.arm.com/documentation/ihi0031/a/The-Serial-Wire-Debug-Port--SW-DP-/Introduction-to-the-ARM-Serial-Wire-Debug--SWD--protocol

(San Diego, California) *(HOPL III)*. Association for Computing Machinery, New York, NY, USA, 9–1–9–50. https://doi.org/10.1145/1238844.1238853

[26] Yuan Zhang, Min Yang, Bo Zhou, Zhemin Yang, Weihua Zhang, and Binyu Zang. 2012. Swift: a register-based JIT compiler for embedded JVMs. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments* (London, England, UK) *(VEE '12)*. Association for Computing Machinery, New York, NY, USA, 63–74. https://doi.org/10.1145/2151024.2151035

# mruby on Resource-Constrained Low-Power Coprocessors of Embedded Devices

Go Suzuki
Tokyo Institute of Technology
Tokyo, Japan
gosuzuki@psg.c.titech.ac.jp

Takuo Watanabe
Tokyo Institute of Technology
Tokyo, Japan
takuo@psg.c.titech.ac.jp

Sosuke Moriguchi
Tokyo Institute of Technology
Tokyo, Japan
chiguri@psg.c.titech.ac.jp

## Abstract

As IoT devices advance, their microcontroller systems-on-a-chip (SoCs) demand higher speeds, more memory, and advanced peripherals, leading to increased power consumption. Integrating low-power (LP) coprocessors in SoCs can reduce power usage while maintaining responsiveness. However, switching application execution to and from the coprocessors generally involves complex and platform-specific procedures. We propose a JIT compilation method for managed programming languages to streamline LP coprocessor use. Our prototype for the programming language mruby includes a JIT compiler and a seamless processor-switching mechanism, enabling rapid development of IoT applications leveraging LP coprocessors. This work-in-progress paper describes the design and implementation of the extended mruby interpreter and presents preliminary evaluations of its power consumption and latency on ESP32-S3 and ESP32-C6.

***CCS Concepts:*** • **Software and its engineering** → **Interpreters**; **Just-in-time compilers**; • **Computer systems organization** → *Embedded software.*

***Keywords:*** managed languages, embedded systems, microcontrollers, low-power coprocessors, power consumption

**ACM Reference Format:**
Go Suzuki, Takuo Watanabe, and Sosuke Moriguchi. 2024. mruby on Resource-Constrained Low-Power Coprocessors of Embedded Devices. In *Proceedings of the 21st ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '24), September 19, 2024, Vienna, Austria.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3679007.3685064

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*MPLR '24, September 19, 2024, Vienna, Austria*
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1118-3/24/09
https://doi.org/10.1145/3679007.3685064

## 1 Introduction

Recent microcontroller systems-on-a-chip (SoCs) accommodate rich peripherals and large enough memory resources to meet the requirements for various Internet-of-Things (IoT) applications. For example, an ESP32-S3 SoC module incorporates WiFi, 512 KiB Static RAM, and 4 MiB or more Flash memory [7]. Despite the growing demands of complex tasks (e.g., communication via MQTT, HTTPS with JSON) performed in such devices, C, C++, and assembly language are still used as the primary implementation languages. So, memory errors and the difficulty of debugging optimized compiled binaries have plagued developers.

To help the rapid development of IoT applications, high-level programming languages with rich programming environments have been proposed for embedded devices [14, 16–18, 23, 25, 28]. Language features, such as dynamic typing and garbage collection (GC), can facilitate the development of complicated but memory-safe IoT devices. For the languages listed above, bytecode VMs running on the target devices are often used instead of native code compilers running on the development host machines. These provide rapid, interactive development. In addition, they make live code updates, such as Over-The-Air updates, much easier than with C/C++. While these are not suitable for real-time systems due to a (naïve) GC, there are many applications suitable for these languages, such as agriculture, meteorological observation, etc.

However, while low power consumption and responsiveness are essential for IoT devices, execution by a VM consumes more power than native code. Putting the processor in sleep mode reduces power consumption but at the expense of responsiveness. To solve this problem, SoCs with *low-power coprocessors* (LP coprocessors) that operate while the main processor is sleeping are gaining popularity. For this purpose, LP coprocessors operate with limited resources. The amount of available memory and accessible peripherals should be limited. Moreover, the address space and processor architecture may also differ from those of the main processors. Thus, developing applications that take advantage of LP coprocessors usually requires writing complex procedures in C/C++ that directly access the hardware.

This work aims to facilitate the development of applications that utilize LP coprocessors using a dynamically typed

**Table 1.** The Specifications of the Targets

| | ESP32-S3 | ESP32-C6 |
|---|---|---|
| Main Processor ISA | Xtensa LX7 | RV32IMAC |
| LP Coprocessor ISA | RV32IMC | RV32IMAC |
| Main SRAM [KiB] | 512 | 512 |
| RTC Slow Memory [KiB] | 8 | 16 |

managed language. Toward this goal, we introduce a JIT compiler and an execution migration mechanism into mruby/c[1] to execute code on LP coprocessors. These mechanisms allow, within an mruby script, seamless switching between the main processor and coprocessor execution in a SoC. Our current contributions are implementations and preliminary evaluation of the mechanisms.

We target two microcontroller SoCs, ESP32-S3 [6] and ESP32-C6 [8]. The main processor and the LP coprocessor are connected via the interconnect. They interact via the RTC Slow Memory, a memory for the LP coprocessor. During the light-sleep state, the LP coprocessor can access only the RTC Slow Memory. This memory can be accessed via the interconnect and is memory-mapped. In ARM-based microcontrollers, a coprocessor, processors, and memories are connected via the AXI interconnect bus [12, 21, 22]. We guess that ESP SoCs are implemented like ARM-based microcontrollers. Table 1 shows the specifications of these targets. The LP coprocessor has limited accessible memory space, accessible peripherals, and processor performance, compared to the main processor. Thanks to these limitations, the LP coprocessor of ESP32-S3 consumes 200 µA, while the main processor consumes 13.2 mA at the lowest frequency (40 MHz) [6].

The rest of this work-in-progress paper is organized as follows. The next section describes related works. Section 3 provides two simple examples to illustrate how to switch execution between the main processor and the LP co-processor. Section 4 describes our proposed method for JIT compilation. Then, Section 5 presents the preliminary evaluation results. Finally, Section 6 discusses the future work and Section 7 concludes the paper.

## 2 Related Work

### 2.1 Ahead-of-Time Compilation

Static TypeScript [1] is a subset of TypeScript with an Ahead-of-Time (AoT) compiler. It targets small-scale embedded devices, which have only 256–512 KiB ROM and 16–256 KiB RAM. Although AoT compilation gives better performance, it can result in larger compiled binaries occasionally. Our target can communicate over WiFi and potentially manipulate JSON data (an example of the highly dynamic data structures). Thus, conservative type checking may generate larger codes. However, AoT compilation can generate better codes for well-typed programs and is suitable for real-time systems. Depending on the application, it is important to employ AoT or JIT compilation properly.

### 2.2 Tiny Interpreters

The Ribbit system [30] is a compact Scheme interpreter with a footprint of 4 KiB, and the following work [24] implements the $R^4RS$ standard in 7 KiB by using LZ compression for bytecodes. Without compression, it exceeds 8 KiB. This suggests that compiling or transferring functions on demand is necessary. Even though bytecodes tend to be smaller than machine code [3], the small memory of the LP coprocessor cannot accommodate all of the standard libraries. We must declare functions used in LP coprocessors, or functions must be compiled/transferred on demand. Moreover, while Scheme is simple, popular high-level programming languages in embedded systems such as Ruby and Python are more complicated. Hence, we think that it is difficult to implement an interpreter for such languages within 8 KiB.

### 2.3 Just-in-Time Compilation

**2.3.1 On Embedded Devices.** Some works [10, 19, 27] developed just-in-time (JIT) compilers for resource-constrained devices. These works imply that ESP32 can do JIT compilation. With the fact that programs on LP coprocessors are usually small, it is possible to run a JIT compiler on main processors to generate programs for LP coprocessors. The conventional main purpose of JIT compilers is the code speed, but our purpose is a small memory resource. An efficient code is not always minimum, for example, excessive code duplication (code specialization) should be avoided.

**2.3.2 For Dynamic Languages.** *Lazy basic block versioning* [4] is a JIT compilation technique suitable for dynamic programming languages and is used in the Ruby compiler YJIT [5]. The compiler incrementally compiles one basic block at a time. Compiled basic blocks have branch stubs for branches to uncompiled basic blocks. When execution reaches such stubs, the compiler resumes the code generation. The header of each basic block has a typing context for local variables, and basic blocks are specialized according to the typing context. The destination of compiled code of branches is determined not only by the program location but also by the typing context. This technique avoids heavy and complex implementations such as type analysis. However, it still requires its own partial evaluator for compiler optimizations such as constant folding and devirtualization.

*Trace-based JIT* [10] is also a JIT compilation technique to detect and compile frequently executed program paths. It gathers constant-foldable values and types of local variables

[1] mruby [20, 29] is a lightweight implementation of Ruby, and mruby/c [23] is an implementation of the mruby runtime that runs on resource-constrained devices. We chose mruby for this study because information on RiteVM (the mruby VM) is relatively easy for us to obtain compared to MicroPython [25] or other languages.

by executing programs by the interpreter. It reduces the code size of the compiler, which is suitable for resource-constrained systems. However, generated codes by trace-based JIT may occur too many code duplications. As a result, the generated code may exceed the memory resource of the LP coprocessor.

### 2.4 Execution Migration

*Execution migration* is to migrate a running program across heterogeneous-ISA (Instruction Set Architecture). One of the purposes of execution migration is to run programs on suitable processors for performance or power consumption. Some works [2, 11] compile statically and generate bare binaries, unlike our approach. In embedded systems, a big issue of execution migration is code size overheads for state transformation (transforming stack frames and the heap) at migration points. It is possible that the compiled binary is twice as big. UNIFICO [15] solves this problem by adjusting the stack and limiting the number of registers. These works allow execution migrations at any call site. We believe that a carefully designed migration granularity can reduce such overheads in applications of LP coprocessors.

Emerald [13] is a managed programming language designed for distributed computing. Objects in Emerald can freely move within the distributed system, and developers can explicitly move objects. Unlike Emerald, in our work, the LP coprocessor and the main processor work exclusively. Thus, developers do not need to think about the concurrency so we will not provide primitives such as monitor. Moreover, we try to move objects automatically on demand without modifying the object system in Ruby.

## 3 Motivating Example

Mainly, (general purpose) coprocessors of microcontrollers have two purposes: for lower power consumption and for real-time tasks. Our approach is not suitable for real-time systems because we employ JIT compilation that results in a long pause time when execution reaches uncompiled program locations. Therefore, we focus on the lower power consumption purpose. In this section, we show two motivating examples. The former is used in the preliminary evaluation.

### 3.1 LED Blinking

LED blinking is a popular test case of embedded systems. Listing 1 is an example using the LP coprocessor, written in Ruby. In this example, the GPIO4 and GPIO5 pins are connected to an LED and a tactile switch, respectively. When the `Copro#run` method is called, the JIT compiler starts to compile the given block. Eventually, the LP coprocessor executes the compiled code and the main processor sleeps (explained in Section 4.1). When the given block is finished (*i.e.*, after the tactile switch is pressed), the main processor wakes up and executes the following program.

**Listing 1.** LED blinking in Ruby

```
1  Copro.run do
2    prevPress = false
3    press = false
4    # While the tactile switch is not pushed.
5    while (!prevPress ||
6      press) do # Negative Edge
7      Copro.gpio(4, true)  # Turn on the LED.
8      Copro.delayMs(30)    # Sleep for 30 ms.
9      Copro.gpio(4, false) # Turn off the LED.
10     Copro.delayMs(30)
11     prevPress = press
12     press = Copro.gpio?(5) # Check the switch.
13   end
14 end
15 # LP coprocessor never executes here.
```

**Listing 2.** IoT Sensor in Ruby

```
1  sensor = SHT3xSensor.new(I2C.new(5,4))
2     # May be set by the JSON config.
3  buffer = Array.new(60)
4  Copro.run do
5    (0...60).each do |i|
6      Copro.delayMs(1000*60)    # Sleep for 1 min.
7      buffer[i] = sensor.read() # Read from sensor.
8    end
9  end
10 Network.send(buffer) # Send buffered data.
```

In this way, we can save the number of migration points discussed in Section 2.4, if we use a static compilation approach. Migration points are only before/after the call sites of `Copro#run`. Stack frame transformations are unnecessary; it only transfers the closure (`Proc` object in Ruby) to the LP coprocessor because the LP coprocessor never executes outside the `Copro#run`.

### 3.2 IoT Sensor

IoT sensors are popular applications to sense humidity, motion, pressure, etc. They gather environmental information from sensors and send gathered information over the network to a central server. Some sensors consume lower current than the main processor, *e.g.*, a humidity and temperature sensor consumes 600 μA typically [26]. The LP coprocessors of our targets can interact with sensors connected over communication methods such as $I^2C$, 1-Wire, and analog-to-digital converters. If the IoT sensor sends to the server infrequently, waking the main processor for measurements affects the battery life.

Listing 2 is an example of an IoT sensor in Ruby. Ruby's subtyping mechanism allows the creation of interfaces that are independent of specific sensors or communication methods. In our implementation, this mechanism can also be used on the LP coprocessors. Thus, if the production of components, such as sensors, becomes discontinued, we can prepare a new code for the replacement components with small changes.

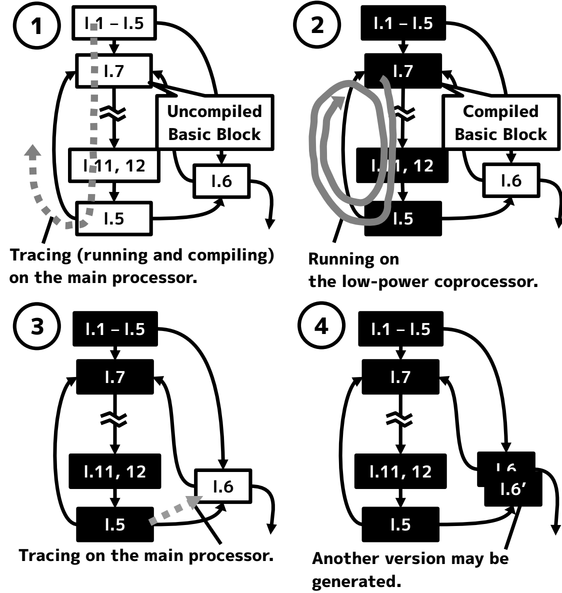



**Figure 1.** An Overview of Just-in-Time Compilation

# 4 Summary of Proposed Method

However, the LP coprocessors of our targets do not have enough memory to run the VM, so we introduce a JIT compiler running on the main processor to generate code for the coprocessor dynamically. This section briefly describes our JIT compiler and discusses object management between the main processor and coprocessor.

## 4.1 Just-in-Time Compilation

Our method is based on lazy basic block versioning [4]. It avoids traditional type analyses. Type analyses can be complex and heavy to support richer type representations. The generated programs for the LP coprocessor also may become inefficient and large due to the conservative type checking. We consider that gathering types by running the program is a cheap way for the footprint and the computation complexity.

However, we use the interpreter for the first execution like trace-based JIT [10] to reduce the processor wake-ups and the execution migrations. Unlike trace-based JIT, the generated codes are divided into basic block versions. This allows compiled basic blocks (including functions) to be reused in the different code paths. Similar to trace-based JIT, our method reuses the original mruby/c interpreter. It reduces the runtime footprint[2]. In addition, if an executing basic block is not appropriate to compile and execute on the LP coprocessor (*e.g.*, low frequently executed, or using not supported features (*e.g.*, too dynamic features) on the LP coprocessor), it allows to run on the main processor seamlessly.

Figure 1 shows the control flow graph corresponding to Listing 1. Each node is a basic block split by method callings and branches, whose label represents the line number on Listing 1. First, uncompiled basic blocks are executed and traced on the main processor (1). The traced basic blocks are compiled while applying optimizations such as type specialization. Then, when it reaches a compiled basic block, the LP coprocessor executes the compiled basic blocks (1 → 2). During execution on the LP coprocessor, the main processor sleeps. After that, when the LP coprocessor reaches the uncompiled basic block, the main processor wakes up and executes and compiles the uncompiled basic block (3). In the LED blinking example, this is happened when the tactile switch is pushed. Like lazy basic block versioning, new basic block versions may be generated (4). If the `Copro#gpio?` (*l*.12) does not return a boolean value at the second time, a new basic block version for *l*.6 is created.

## 4.2 Objects

During execution on the LP coprocessor, shapes of objects are fixed. Unlike Python, Ruby does not allow to access instance variables outside instance methods. We assume that programs on the LP coprocessor do not use too dynamic features. On the LP coprocessor, too dynamic features such as `Object#extend` and class definitions are disallowed. As a result, objects can be realized without hash tables. We note that dynamic features still can be used on the main processor (outside `Copro#run`).

A `Method` object on the main processor can contain a pointer to a C function on the LP coprocessor, in addition to a C function on the main processor. When it is called, it executes the C function on the main processor; instead, the compiled code calls the C function on the LP coprocessor. `Copro#gpio` and `Copro#delayMs` use this to execute on both processors.

# 5 Preliminary Evaluation

## 5.1 Evaluation Detail

We evaluate the code size and the wake-up/compile overheads by the LED blinking example (described in Listing 1) with the initial implementation. Currently, it supports:

- Integers, Booleans and `nil`
- Arithmetic operators
- A single call frame migration
- Calling methods defined in C
- Garbage Collection (however, not used)

and does not support:

- Objects including Arrays, Strings and Hashes
- Calling methods defined in Ruby
- Global variables
- Closures

Table 2 shows the evaluation environment. We evaluated under following configurations:

[2] Currently, we copied the original one.

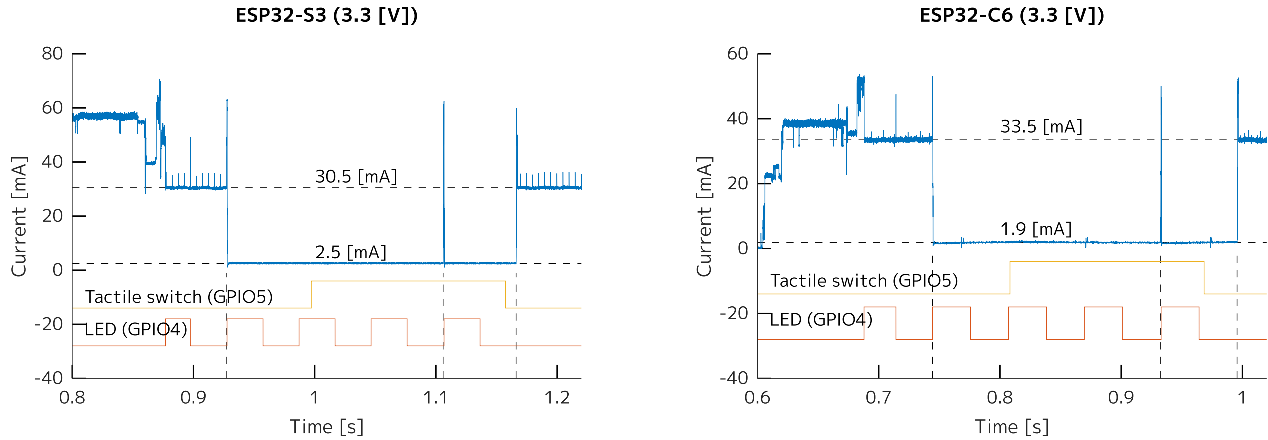



**Figure 2.** Current Consumption Results

**Table 2.** The Evaluation Environment

| Role | Name and Revision |
|---|---|
| Evaluation boards | ESP32-S3-DevKitC-1 N8, v1.0<br>ESP32-C6-DevKitC-1 N8, v1.3<br>(Main processors freq. : 160 [MHz]) |
| SDK | ESP-IDF v5.2.1 |
| Ammeter | Nordic Power Profiler Kit 2 (PPK2) |
| Signal Generator | Nodemcu ESP8266 Ver 0.1 |

- The signal generator simulates the tactile switch input. It makes GPIO5 (tactile switch) high for 160 [ms], 120 [ms] after GPIO4 (LED) is high at the first time.
- The sample rate of PPK2 is 100 [kHz]. It delivers the 3.3 [V] power to the targets and captures GPIO signals.
- On the ESP32-C6-DevkitC-1, the jumper is removed to disconnect the power-on indicator LED and the UART/USB controller [9]. However, on the ESP32-S3-DevkitC-1, there is no jumper so they are connected.
- The commit hash of mruby/c that we use is `73c1324f93`.
- The watchdog timers are disabled.
- We used the light-sleep state because the deep-sleep state cannot retain the main memory.

We measured the code size of the mruby runtime by `$ idf.py size-components` provided by ESP-IDF SDK. We also measured the wake-up time of the processors. GPIO1 becomes high before the running processor wakes the other processor and becomes low after the other processor is ready.

### 5.2 Result and Discussion

Table 3 shows the code size of the mruby runtime. The code of the LP coprocessor contains the garbage collector, the implementations of `Copro#delayMs`, `Copro#gpio`, and the bootstrap. The code size changes are about 20 [KiB]. Since we currently copy the original interpreter and modify it, these changes are not small compared to the interpreter code size (about 55 or 65 [KiB]). This suggests that the code should be shared between the interpreter and the JIT compiler when the ROM size is limited.

**Table 3.** The Runtime Code Size [B]

| Target | | Main Processor .data+.bss | Main Processor .text | Main Processor .rodata | Copro. |
|---|---|---|---|---|---|
| ESP32-S3 | Orig. | 2582 | 48928 | 6511 | 0 |
| | Ours | 2614 | 68411 | 6864 | 1336 |
| ESP32-C6 | Orig. | 2574 | 57456 | 6615 | 0 |
| | Ours | 2614 | 80822 | 6968 | 3072 |

The memory overheads on both targets are:

- Profiling data on the main processor: 776 [B]
- Generated codes on the LP coprocessor: 220 [B]

Profiling data manages the register allocation and the typings. The generated code has many move instructions for constant values. This can be reduced if the copy-on-write register allocation is implemented (discussed in Section 6.1).

Figure 2 shows the current consumptions. In the figures, the horizontal dashed lines represent approximate values under steady-states. 30.5 [mA] and 33.5 [mA] are the power consumption of the main processor, and 2.5 [mA] and 1.9 [mA] are the power consumption of the LP coprocessor. By using the LP coprocessors, we can see that the power consumption is reduced by about 10 times. The vertical dashed lines in the figures represent when the main processors wake up or enter the sleep state. At the first vertical dashed line, the main processors enter the sleep state. At the second vertical dashed line, the main processors wake up and compile the following program for the LP coprocessors, then enter the sleep states. Since The tactile switch input changes its value, the taken branch is changed, and the uncompiled basic block is compiled. After the third dashed line, execution of `Copro#run`

**Table 4.** Wake-Up Time Overheads [ms]

| | ESP32-S3 | ESP32-C6 |
|---|---|---|
| The Main Processor | 0.51 | 0.58 |
| The LP Coprocessor | 0.18 | 0.02 |

finishes, and the main processors continue to work. Comparing before/after the first dashed line, we observed that the power consumption is reduced by the LP coprocessor. In this example, ten basic block versions are generated, but the main processor has only woken up twice. This is due to the tracing of the first execution by our method.

Table 4 shows the elapsed time of processor wake-ups. These overheads are inevitable when a processor wakes up, and are expected to be a problem for some applications. For example, $I^2C$ standard mode operates at 100 [kHz] (*i.e.* 1-bit per 0.01 [ms]). If the processor wakes up during the $I^2C$ communication by software, the timing is not met. We believe that entering the sleep state with a delay can alleviate this problem.

# 6 Future Work

## 6.1 Code Generation

Because the mruby bytecode is a register machine, the register allocation is simply done with one-pass algorithms. However, the instruction format is represented as (`R_i` is the *i*th register.):

```
ADD i # R_i = R_i + R_(i+1)
```

This frequently introduces move instructions despite the three-address code of RISC-V, the target (as follows).

```
add i, j, k # R_i = R_j + R_k
```

To reduce the code size, we should implement a copy-on-write register allocation algorithm. However, using a simple one-pass algorithm, each basic block version must have allocated register numbers for local variables because the allocated registers may be different with the code path. Using a two-pass algorithm, the generated code is minimal, but the code size of the compiler and the compilation time will be larger (The instruction format of the mruby bytecode is variable-sized). We must design the register allocation while considering the trade-off between the generated code size and the compilation overhead.

## 6.2 Object Management

Because objects are transferred on demand, read barriers are required, *e.g.*, before reading an instance variable. Before a pointer outside the memory region available for the LP coprocessor is copied into a register, the object pointed at must be transferred and the pointer must be translated. Instead of barriers, we consider that the hardware interrupts can be used. The handler for the bus error interrupts[3] searches an Least-Recently-Used (LRU) table. The table has combinations of addresses translated from and to. If the pointer is not found in that table, it may not have been transferred. The main processor wakes up to transfer the object. The main processor also has a complete (non-LRU) table. If the pointer is found in that table, the main processor tells the LP coprocessor the translated address. If it is not found, the object is transferred and then is told to the LP coprocessor. Fortunately, popular microcontrollers do not overlap memory regions among the LP and main processors.

## 6.3 I/O

In modern microcontrollers, the control registers of peripherals are memory-mapped. The problem is how to implement I/O functions. When the developer implements I/O functions in C language, additional machine codes have to be placed on the LP coprocessor. Even when the I/O functions are actually not used, these codes still have to be placed beforehand. It is a problem if live code updates happen. To avoid this, a relocation infrastructure for C functions is required.

To implement I/O functions in Ruby, we consider the differences in the memory-mapped addresses of the control registers between the processors. To solve this, we need to define an address translation function in C language, separately. Another solution is to perform the address translation on the bus error interrupts. However, guarding in the Ruby program cannot solve this problem because programs executed on the LP coprocessor must be executed on the main processor during compilation.

# 7 Concluding Remark

In this work-in-progress paper, we propose a method for utilizing low-power (LP) coprocessors in microcontroller SoCs using a dynamically typed managed language. The proposal introduces a JIT compiler for LP coprocessors that do not have sufficient memory and an inter-processor object management method. Our proposal enables seamless use of LP coprocessors in mruby scripts. We implemented a prototype based on mruby/c running on two microcontroller SoCs ESP32-S3 and ESP32-C6. The evaluation of their power consumption and the wake-up time of each processor shows that the proposed method has sufficient practical use.

# Acknowledgments

This work was supported in part by JSPS KAKENHI Grant Numbers JP22K11967 and JP24K14892.

[3]It is thrown on the invalid memory accesses

## References


[1] Thomas Ball, Peli de Halleux, and Michał Moskal. 2019. Static TypeScript: an implementation of a static compiler for the TypeScript language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (Athens, Greece) *(MPLR 2019)*. Association for Computing Machinery, New York, NY, USA, 105–116. https://doi.org/10.1145/3357390.3361032

[2] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. 2015. Popcorn: bridging the programmability gap in heterogeneous-ISA platforms. In *Proceedings of the Tenth European Conference on Computer Systems* (Bordeaux, France) *(EuroSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 29, 16 pages. https://doi.org/10.1145/2741948.2741962

[3] Bernd Burgstaller, Bernhard Scholz, and Anton Ertl. 2006. An Embedded Systems Programming Environment for C. In *Euro-Par 2006 Parallel Processing*, Wolfgang E. Nagel, Wolfgang V. Walter, and Wolfgang Lehner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1204–1216.

[4] Maxime Chevalier-Boisvert and Marc Feeley. 2015. Simple and Effective Type Check Removal through Lazy Basic Block Versioning. In *29th European Conference on Object-Oriented Programming (ECOOP 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 37)*, John Tang Boyland (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 101–123. https://doi.org/10.4230/LIPIcs.ECOOP.2015.101

[5] Maxime Chevalier-Boisvert, Takashi Kokubun, Noah Gibbs, Si Xing (Alan) Wu, Aaron Patterson, and Jemma Issroff. 2023. Evaluating YJIT's Performance in a Production Context: A Pragmatic Approach. In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (Cascais, Portugal) *(MPLR 2023)*. Association for Computing Machinery, New York, NY, USA, 20–33. https://doi.org/10.1145/3617651.3622982

[6] Espressif Systems 2023. *ESP32-S3 Technical Reference Manual*. Espressif Systems.

[7] Espressif Systems 2023. *ESP32-S3-WROOM-1, ESP32-S3-WROOM-1U Datasheet*. Espressif Systems.

[8] Espressif Systems 2024. *ESP32-C6 Technical Reference Manual*. Espressif Systems.

[9] Espressif Systems Accessed May 2024. *ESP32-C6-DevKitC-1 v1.2*. Espressif Systems. https://docs.espressif.com/projects/espressif-esp-dev-kits/en/latest/esp32c6/esp32-c6-devkitc-1/user_guide.html#current-measurement

[10] Andreas Gal, Christian W. Probst, and Michael Franz. 2006. HotpathVM: an effective JIT compiler for resource-constrained devices. In *Proceedings of the 2nd International Conference on Virtual Execution Environments* (Ottawa, Ontario, Canada) *(VEE '06)*. Association for Computing Machinery, New York, NY, USA, 144–153. https://doi.org/10.1145/1134760.1134780

[11] Edson Horta, Ho-Ren Chuang, Naarayanan Rao VSathish, Cesar Philippidis, Antonio Barbalace, Pierre Olivier, and Binoy Ravindran. 2021. Xar-trek: run-time execution migration among FPGAs and heterogeneous-ISA CPUs. In *Proceedings of the 22nd International Middleware Conference* (Québec city, Canada) *(Middleware '21)*. Association for Computing Machinery, New York, NY, USA, 104–118. https://doi.org/10.1145/3464298.3493388

[12] Infenion Technologies AG 2023. *PSoC 6 MCU: CY8C61x4, CY8C62x4 Architecture Technical Reference Manual (TRM)*. Infenion Technologies AG.

[13] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. 1988. Fine-grained mobility in the Emerald system. *ACM Trans. Comput. Syst.* 6, 1 (feb 1988), 109–133. https://doi.org/10.1145/35037.42182

[14] M5Stack. Accessed May 2024. UIFlow 2.0. https://uiflow2.m5stack.com/

[15] Nikolaos Mavrogeorgis, Christos Vasiladiotis, Pei Mu, Amir Khordadi, Björn Franke, and Antonio Barbalace. 2024. UNIFICO: Thread Migration in Heterogeneous-ISA CPUs without State Transformation. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction* (Edinburgh, United Kingdom) *(CC 2024)*. Association for Computing Machinery, New York, NY, USA, 86–99. https://doi.org/10.1145/3640537.3641565

[16] Microsoft. Accessed May 2024. DeviceScript. https://microsoft.github.io/devicescript/

[17] Moddable Tech, Inc. Accessed May 2024. Node-RED MCU Edition. https://github.com/phoddie/node-red-mcu

[18] Moddable Tech, Inc. Accessed May 2024. XS JavaScript. https://github.com/Moddable-OpenSource/moddable

[19] Konrad Moron and Stefan Wallentowitz. 2023. Support for Just-in-Time Compilation of WebAssembly for Embedded Systems. In *2023 12th Mediterranean Conference on Embedded Computing (MECO)*. 1–4. https://doi.org/10.1109/MECO58584.2023.10155088

[20] mruby developers. Accessed May 2024. mruby. https://github.com/mruby/mruby

[21] NXP Semiconductors N.V. 2019. *UM10503 LPC43xx/LPC43Sxx ARM Cortex(R)-M4/M0 multi-core microcontroller*. NXP Semiconductors N.V.

[22] NXP Semiconductors N.V. 2021. *i.MX RT1160 Processor Reference Manual*. NXP Semiconductors N.V.

[23] Kyushu Institute of Technology and Shimane IT Open-Innovation Center. Accessed May 2024. mruby/c. https://github.com/mrubyc/mrubyc

[24] Léonard Oest O'Leary, Mathis Laroche, and Marc Feeley. 2023. A R4RS Compliant REPL in 7 KB. arXiv:2310.13589 [cs.PL]

[25] George Robotics. Accessed Jan. 2024. MicroPython - Python for microcontrollers. https://micropython.org/

[26] SENSIRION 2019. *Datasheet SHT3x-DIS Humidity and Temperature Sensor*. SENSIRION.

[27] Nik Shaylor. 2002. A Just-in-Time Compiler for Memory-Constrained Low-Power Devices. In *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium*. USENIX Association, USA, 119–126.

[28] Ruby Programming Shounendan. Accessed May 2024. Smalruby for mruby/c (SmT). https://github.com/gfd-dennou-club/smt-gui

[29] Kazuaki Tanaka, Avinash Dev Nagumanthri, and Yukihiro Matsumoto. 2015. mruby: Rapid Software Development for Embedded Systems. In *15th International Conference on Computational Science and Its Applications*. IEEE. https://doi.org/10.1109/ICCSA.2015.22

[30] Samuel Yvon and Marc Feeley. 2021. A small scheme VM, compiler, and REPL in 4k. In *Proceedings of the 13th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages* (Chicago, IL, USA) *(VMIL 2021)*. Association for Computing Machinery, New York, NY, USA, 14–24. https://doi.org/10.1145/3486606.3486783


Received 2024-05-25; accepted 2024-06-24

# Imagine There's No Source Code

## Replay Diagnostic Location Information in Dynamic EDSL Meta-programming

Baltasar Trancón y Widemann
Technische Hochschule Brandenburg
Brandenburg, DE
baltasar@trancon.de

Markus Lepper
semantics gGmbH
Berlin, DE

## Abstract

Programs in embedded domain-specific languages are realized as graphs of objects of the host language rather than as static input texts. This property enables dynamic meta-programming, but also makes it harder to attach location information to diagnostic messages that arise at a later stage, after the program graph construction. Thus, EDSL-generating expressions and algorithms can be difficult to debug. Here, we present a technique for transparently capturing and replaying location information about the origin of EDSL program objects. It has been implemented in the context of the LLJava-live EDSL-to-bytecode compiler framework on the JVM. The basic idea can be generalized to other contexts, and to any managed runtime environment with reified stack traces.

***CCS Concepts:*** • **Software and its engineering → Domain specific languages**; *Software prototyping*; *Extensible languages*; **Dynamic compilers**.

***Keywords:*** embedded domain-specific languages, diagnostic information, dynamic meta-programming, Java virtual machine

**ACM Reference Format:**
Baltasar Trancón y Widemann and Markus Lepper. 2024. Imagine There's No Source Code: Replay Diagnostic Location Information in Dynamic EDSL Meta-programming. In *Proceedings of the 21st ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '24), September 19, 2024, Vienna, Austria.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3679007.3685061

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*MPLR '24, September 19, 2024, Vienna, Austria*
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1118-3/24/09
https://doi.org/10.1145/3679007.3685061

## 1 Introduction

Embedded domain-specific languages (EDSLs) are a convenient way to enhance the expressive power of a general-purpose programming language [4, 6]. Instead of defining a syntax of their own, and a toolchain for the processing of dedicated source files, they manifest as an API in the host language [10]. The embedding is called *flat* if calls to the API cause the intended domain-specific behavior directly, or *deep* otherwise [5]. In the remainder of this paper, only deep embedding will be considered. Popular deeply embedded DSLs include the streams framework for data processing hosted in Java, LINQ for queries hosted on .NET, and dask for parallel computing hosted in Python.

Deeply embedded DSLs work in two, or more, stages [9]: In the first stage, a graph of *program objects* (POs) is constructed. These provide a distinct API that is able to cause the intended behavior in a second stage. Hence the actual EDSL program, i.e., the PO graph, is *not* explicit in the host-language source code. Thus, if a second-stage error is detected, a location where the offending fragment is written down may not exist, and hence cannot take the blame; rather, blame must be assigned to a location where the PO graph has been constructed by the hosting meta-program.

There are several levels of implicit expression:

1. PO constructor expressions can be used as a canonical notation for EDSL syntax trees, in a static and homomorphic way. This also functions as a foundation for the higher levels:
2. Factory functions can implement idioms and syntactic sugar. These may be shipped as an additional API of the EDSL, or even as the only public one, for reasons of data abstraction.
3. Meta-programming can involve arbitrarily complex host-language algorithms that dynamically construct an EDSL graph. These can be an integral part of the EDSL, provided by third-party meta-programmers, or devised as proprietary abstractions by the EDSL user.

For instance, in an EDSL with a multiplication operator, the squaring of some data x could be expressed on the three levels as **new** Mult(x, x), or as square(x), or as pow(x, 2), respectively, where the latter more generally implements fast exponentiation by recursive squaring.

As a toy example, consider a Java-hosted EDSL that realizes a process algebra of "multi-media" application behavior.

```
flash (). andAlso(beep().later ()). orElse(flash (). later ())
// Problem here: ---------------^^^^
```

**Figure 1.** Statically embedded example program

The implementing classes and their constructors are private, such that the canonical expression level is not available. Instead, program construction is based on a factory API: There are two distinct primitive events, flash() and beep(). The binary operators andAlso() and orElse() denote parallel composition and nondeterministic choice, respectively. The unary operator later () defers a behavior for some amount of time. See Figure 1 for an example.

Furthermore, the language shall define some semantic restrictions:

1. No two beeps may occur at the same time.
2. Any flash must be followed by a beep later.

These properties should of course be checked, and diagnostic measures taken if a violation is detected. Contrary to optimistic intuition, however, the subtly different temporal-logical nature of the two properties makes a big difference in practice:

Property 1 is a *safety* property [1] that can be enforced locally and bottom-up, since a valid program may only have valid subprograms. Violations need to be checked only, and can be reported immediately, for andAlso operators. The obvious countermeasure is to throw an exception instead of returning the constructed illegal program. Thus, errors are detected early, and the blame is located precisely by means of call-stack information.

By contrast, Property 2 is a *liveness* property [1] that can only be enforced globally, since for any invalid subprogram there is a context that restores validity. Violations must be reported only when the construction is known to be complete, at which point the call sites into the EDSL API are necessarily no longer on the stack.

For simple static EDSL program expressions, it is quite feasible to locate errors by code inspection. For instance, in the example given above, the second occurrence of flash is easily spotted as the offender. Unfortunately, this debugging technique does not scale well to more complex, let alone dynamic program construction. In the worst case, when the meta-program is a full-fledged compiler that targets the EDSL, we have found unaided debugging utterly infeasible in practice.

This paper discusses a novel solution to the problem that

- captures and replays diagnostic location information for EDSL POs,
- works automatically and transparently without explicit effort on behalf of the EDSL user, and
- can be realized with standard facilities of managed runtime environments, including but not limited to the JVM.

It appears that this particular set of requirements has not been addressed before. The situation is different where an actual source text and compiler are involved: Intermediate representations used for hosting compiled DSLs on general-purpose platforms, such as the Java or .NET language models or LLVM of course do support location information that a DSL compiler may collect. However, the two-stage nature of EDSLs, with the first stage being potentially both dynamic and light-weight, poses the additional challenges that we investigate and solve in the present work.

## 2 Design

The proposed technique requires that the hosting runtime environment provides *reified* caller information, i.e., there is an operation that can capture the source code location of pending calls, at any point of the execution of some application thread, as data objects. Additionally, EDSL POs must be able to store (a relevant excerpt of) such data.

Finally, it is required that the EDSL-using application is *stratified*, i.e., there is an objectively decidable criterion whether any given method in the running application belongs to either

1. the EDSL service, namely
   a. the public API of the EDSL,
   b. the private implementation of the EDSL,
   c. some other library that is (transitively) used by the EDSL, independently of the application, and makes dynamic callbacks to the EDSL,

   or otherwise
2. the client application,

and that the application may only call the private implementation of the EDSL through its public API. The recognition of the former three categories can be settled once and for all by the EDSL provider. If application code respects the public API as an abstraction barrier, it is safe to lump all other code under the last one.

A call to the EDSL API is called a *client call* if the caller belongs to the application, or otherwise an *internal call*.

Every client call provides a source of information about the location of its caller, the most specific data that is directly relevant to the client application programmer. The constructors of EDSL POs act as the sinks for such information, storing the data for some checks to be performed later.

There are important reasons *not* to collect location information greedily on each public API call:

- The public API may be reentrant, leading to redundant copies of location data.

- Explicitly passing location data down the call tree to the constructor sinks is inconvenient, and clutters both method signatures and code.

A more elegant technique relies on the fact that the client call remains (somewhere) on the stack until concerned constructors are finished. Hence the collection of call stack data can be deferred until requested in a constructor, and the client identified by scanning down the stack, skipping all intervening internal calls. Thus, location data is properly encapsulated, and clutter greatly reduced.

# 3 Implementation

The implementation of a technique for capturing diagnostic location information for later replay splits into three modular tasks: Call stack data needs to be *supplied* by the runtime environment, *selected* by a generic library, and *stored* by the EDSL.

## 3.1 Suppliers of Reified Stack Data

The JVM has, for historical reasons, not one but two mechanisms for stack reification with overlapping but distinct features [3].

The older mechanism, present in all Java versions, powers the stack trace information stored in exceptions, and is available publicly via constructing and querying an exception, or since Java 5 via the method java.lang.Thread.getStackTrace(). In either case, an array of StackTraceElement objects is returned, which has been eagerly populated with information about all pending callers. This approach has two downsides [3]:

1. The overhead increases at least linearly with the depth of the call stack, much of which may be irrelevant for the present purpose.
2. The available information is purely symbolic; the caller class is specified only by name (and the class name of its class loader), which may be hard to resolve correctly, or even actually ambiguous, in the presence of multiple class loaders.

The newer and more sophisticated mechanism, introduced in Java 9, uses the API of the class java.lang.StackWalker. The call stack can be accessed via a stream of StackFrame objects. This approach remedies both problems discussed above:

1. The stream of stack frames is constructed lazily; thus the overhead depends only on the depth of the actually observed part of the call stack.
2. Stack walkers can (optionally) specify the identity of the calling class directly as a Class object; no manual class name resolution is required.

We have found the differences in the technical capabilities to be of modest practical importance, in particular since location information is often boiled down to a pathless file name and a line number; hence both choices are generally feasible.

## 3.2 Selection of Client Calls

In order to keep the choice between supplier mechanisms transparent, we introduce a simple EDSL for predicates on stack frames that expresses the logics abstractly, and can be compiled to either back-end supplier; see Figure 2.

We have found that purely string-based predicates are often cumbersome and of poor expressive power in practice. For instance, consider that an EDSL X exposes an API class XProvider, and allows clients to inject "trusted" code by overriding methods in a subclass. Thus the internal classes comprise all classes in the same package as XProvider, and its subclasses in other packages, as well as inner classes of such classes. This semantic predicate can be expressed concisely; see Figure 3.

## 3.3 Storage of Location Information

Caller information is stored as an abstract datatype of location information, which can be collected by means of a static factory; see Figure 4 for a minimal API. The walker-based approach allows for a particularly elegant, declarative implementation; see Figure 5.

The actual location datatype that we have used in the implementation is richer. In particular, it supports parameterized semantic document identifier types rather than simple strings, column numbers, and robust interval logics that can deal with partial information and "include" directives.

Location data storage functionality must be provided explicitly by the EDSL program classes. The strategy specified above for querying the call stack ensures that it can be implemented as a private side effect in constructors. Hence, if there is a single common base class for all EDSL POs, only a self-contained code injection into that class is necessary; see Figure 6. Public APIs and the rest of the EDSL implementation are not affected; client code does not have to cooperate, nor even be aware that location data are collected.

# 4 Case Study

A worked-out didactical case study can be found in the supplementary appendix.

# 5 Real-World Evaluation

The use of replay diagnostic location information, captured by the technique described above, has been implemented and validated in practice in the context of LLJava(-live).

LLJava [12] is a low-level JVM programming language. In its standalone, textual form it serves similar purposes as "JVM assembly" representations such as javap or Jasmin, only at a somewhat higher level of abstraction. The

```
public interface StackFilter {

    <S> Predicate<S> compile(StackFilterCompiler<S>);

    static StackFilter any();
    static StackFilter none();
    StackFilter and(StackFilter);
    StackFilter or(StackFilter);
    StackFilter negate();

    static StackFilter classIs(Predicate<Class<?>>);
    static StackFilter classNameIs(Predicate<String>);
    static StackFilter methodNameIs(Predicate<String>);

    static Predicate<Class<?>> inToplevel(Predicate<Class<?>>);
    static Predicate<Class<?>> inPackageOf(Predicate<Class<?>>);
    static Predicate<Class<?>> subclassOf(Predicate<Class<?>>);
}

class TracingStackFilterCompiler implements StackFilterCompiler<StackTraceElement> {...}

class WalkingStackFilterCompiler implements StackFilterCompiler<StackWalker.StackFrame> {...}
```

**Figure 2.** Stack Filter EDSL

```
Class<?> api = XProvider.class;
StackFilter internal =
    classIs(inToplevel(inPackageOf(api)
                        .or(subclassOf(api))));
```

**Figure 3.** Example Filter Predicate

```
public interface Location {
    String getDocumentId();
    int getLineNumber();
    static Optional<Location> live(StackFilter);
}
```

**Figure 4.** Location Data API

```
Predicate<StackWalker.StackFrame> dropping =
    internal.compile(getWalkingCompiler());
return walker.walk(stream → stream.skip(overhead)
                            .dropWhile(dropping)
                            .map(frame2location)
                            .findFirst ());
```

**Figure 5.** Body of Method Location.live (simplified)

```
public abstract class MyAbstractSyntax {
    // ...
    private static final StackFilter MY_INTERNAL =
        /* recognize category 1(a–c) */;
    private final Optional<Location> location =
        Location.live(MY_INTERNAL);
    public Optional<Location> getLocation() {
        return location;
    }
}
```

**Figure 6.** Location Data Injection

LLJava toolchain, comprising a parser, an intermediate representation, and a compiler and a decompiler back-end, is implemented in Java.

LLJava-live [11] is a front-end builder API for the LLJava intermediate representation. Together, they form expression levels 1 and 2, respectively, of an EDSL for JVM bytecode generation. The design of LLJava-live is geared particularly towards staged meta-programming, i.e., the dynamic generation of code for subsequent direct use in the same application. This capability is particularly useful for accelerating other EDSLs, by just-in-time-compiling their loose PO graphs to compact specialized bytecode.

```
class IntAdd extends IntBinaryOperator {

  /** Evaluate the expression right away. */
  @Override
  public int evaluate(Environment env) {
    return getLeftOperand().evaluate(env)
         + getRightOperand().evaluate(env);
  }

  /** Emit JVM code to evaluate the expression,
   *  pushing the result on the operand stack. */
  @Override
  public void compile(CompilationContext cc) {
    getLeftOperand().compile(cc);
    getRightOperand().compile(cc);
    cc.add();
  }

}
```

**Figure 7.** Example LLJava-Compilable PO Class

Since LLJava POs put the full complexity of JVM bytecode at the disposal of the client application, numerous complex checks have to be performed in order to ensure that verifiable code is emitted. Some of these are liveness-like and global in nature, e.g., the integrity (types and bounds) check for the JVM operand stack, or the requirement that no control path may fall off the end of a method body. The LLJava back-end performs the analysis as described in the JVM specification [7, §4.10], as a whole-method check.

By contrast, interpreters for EDSLs tend to be highly modular; every PO class can bring their own means of execution. This approach yields a clean and extensible design. However, it implies that a compiler for such a language must also be modular. Thus, a typical client of LLJava-live comes with a distributed code generator: Small fragments of bytecode are constructed in various places, brought together by aggressive inlining, and connected by implicit data flow via the JVM operand stack. The correctness of each code generator fragment is heavily context-dependent because of implicit cross-boundary data flow.

See Figure 7 for an example, namely a PO class that defines a binary integer addition operation. The first method implements the interpreter fragment for this language construct, the second the compiler. Whereas the former can be checked statically to a high degree of certainty by the Java compiler, the latter relies on a lot of implicit information: It is assumed that compiling the left and right operands will each emit code to the effect that exactly one value of type **int** is pushed onto the operand stack, such that the finally emitted "iadd" instruction meets with the correct context. Hence the compliance of all POs is required to ensure overall consistency.

We have found this kind of fragility to interact with liveness-like global constraints in complex ways, and to be a likely source of subtle errors. For debugging such a code generator, it is crucial to have precise blame assignment; the alternative outcome without replay diagnostic location information, namely a post-mortem JVM VerifyError, would not be helpful.

The practical use of replay diagnostic information concerning modular code generators has been tried with Whilst, a didactic Pascal-like toy EDSL specifically designed for showcasing LLJava-live [11]. In particular, Whilst is by design compilable, modular and open for extension. All language constructs with operational semantics, namely statements and expressions, are endowed with their own interpreter and compiler. A new language construct can be added at any time, just by loading a new subclass of Statement or Expression.

The qualitative benefit from having accurate diagnostic location information is significant; the task of debugging a code generator fragment is reduced in complexity from "daunting" to "routine".

### 5.1 Time and Space Overhead

Although no code changes to the client are required in order to reap the benefits of replay location information, there is a dynamic price to pay; the processing of call stack data increases the running time of the compiler. As discussed above, the simple, robust default strategy is to store location data eagerly for every PO. In this case, we have measured an overhead of 30–60% for code generation with LLJava-live, using the stack walker implementation. Mileage for other EDSLs and usage contexts may vary significantly, for various reasons:

- In particular, the relative overhead for EDSLs with trivial semantics and hence very fast interpreters is expected to be much bigger, whereas for EDSLs that execute on a larger timescale, e.g., remote database queries or intensive numerical calculations, it is likely negligible.
- The cost of creating and scavenging stack traces varies greatly both with the choice of supplier, and the dynamic nesting depth of EDSL generators in the application. In our experiments, where EDSL generators run on rather shallow call stacks, we have found stack trace costs roughly on par with the allocation and initialization costs for location data. We would expect the former to dominate in large real-world applications with deeper call stacks.

This work-in-progress report does not provide conclusive experimental results; in the future a more thorough investigation of different realistic scenarios is required.

The assessment of space costs for storing captured location information likewise needs to take the complexity of the EDSL POs into account. A useful Location object stores at least a reference to a shareable file name string and a line number. Thus, for very simple PO classes, the space overhead can rise to about 100%. However, we have found that EDSL programs tend to be fairly small in terms of the PO graph size; hence this might not be too much of an issue.

### 5.2 Tactical Considerations

To mitigate both time and space costs, location data can be associated only with certain types of POs, or restricted by some other criteria, e.g. the semantic categories selected for checkpointing.

Location information capture behaves rather analogous to assertions: On the one hand, the costs vary wildly with the usage profile. On the other hand, for production configurations of the software, they may be switched off, per code unit or wholesale. Even the mechanisms used by the Java platform for controlling assertions can be re-used for this purpose, and hence the same negligible runtime overhead applies when the feature is not used. Only the space overhead of POs carrying a null reference as location (non-)information remains.

Note that virtually all costs are incurred during the first stage when the EDSL POs are constructed, the only exception being increased cache pressure in the execution stage due to extra attributes. Hence they are naturally amortized by both long-running and often-reused EDSL programs.

### 5.3 Spatial Resolution

The technique discussed here, while requiring little effort from the EDSL designer, and hardly any from the user, inherits the resolution of location data from the host execution platform, in this case the JVM. As usual for traditional imperative languages, *statements* and sequences of lines are the basic structures, and thus a source file name and a single line number are the coordinates.

This paradigm may or may not be adequate for EDSLs: In particular, declarative languages tend to put more complexity into *expressions* that are inherently tree-shaped, and thus require at least column numbers, and ideally intervals, to specify a location with useful precision.

For "non-linear" EDSLs, the same precautions apply for captured location information as they would for execution-time exception stack traces: Where the distinction between two contructs is of interest, they should be placed on different lines. This has the ironic implication that EDSL code generators written in imperative style are easier to equip with precise location information than highly declarative ones.

## 6 Conclusion

Deep embedding of DSLs works in stages, where the client application first constructs a graph of program objects that define some domain-specific behavior, which can be checked, transformed and executed later. The PO graph functions as the intermediate representation of a DSL program that need not have any directly corresponding source code. Error reports from the later stages need to point to the meta-program that constructs the POs instead. This location information can be captured from the call stack at PO construction time, requiring strictly no effort on behalf of the DSL user for single-point diagnostics. Debugging of recursive or context-sensitive bits of code generators may be aided by user intervention, which can be supported with checkpointing facilities.

We have discussed strategies for capturing location information, as well as for filtering the stack trace to account for complex stratified EDSL implementations.

Note that the approach presented in this paper is only concerned with the location part of EDSL error messages, not their content; see [8] for a survey of the state of the art of domain-specific errors, and [2] for compiler errors in general.

## Acknowledgments

Anonymous reviewers have helped to shape and improve this paper.

## References

[1] Bowen Alpern and Fred B. Schneider. 1987. Recognizing Safety and Liveness. *Distrib. Comput.* 2, 3 (sep 1987), 117–126. https://doi.org/10.1007/BF01782772

[2] Brett A. Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J. Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, Janice L. Pearce, and James Prather. 2019. Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education* (Aberdeen, Scotland Uk) *(ITiCSE-WGR '19)*. Association for Computing Machinery, New York, NY, USA, 177–210. https://doi.org/10.1145/3344429.3372508

[3] Mandy Chung. 2014. *Stack-Walking API.* OpenJDK. https://openjdk.org/jeps/259

[4] Matthias Felleisen. 1991. On the Expressive Power of Programming Languages. *Sci. Comput. Program.* 17, 1-3 (1991), 35–75. https://doi.org/10.1016/0167-6423(91)90036-W

[5] Jeremy Gibbons and Nicolas Wu. 2014. Folding Domain-Specific Languages: Deep and Shallow Embeddings (Functional Pearl). *SIGPLAN Not.* 49, 9 (aug 2014), 339–347. https://doi.org/10.1145/2692915.2628138

[6] Paul Hudak. 1996. Building Domain-Specific Embedded Languages. *ACM Comput. Surv.* 28, 4es (dec 1996), 196–es. https://doi.org/10.1145/242224.242477

[7] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, and Daniel Smith. 2018. *The Java Virtual Machine Specification* (Java SE 11 ed.). JSR, Vol. 384. Oracle. https://docs.oracle.com/javase/specs/jvms/se11/jvms11.pdf

[8] Alejandro Serrano and Jurriaan Hage. 2019. A compiler architecture for domain-specific type error diagnosis. *Open Computer Science* 9, 1

(2019), 33–51. https://doi.org/doi:10.1515/comp-2019-0002

[9] Tim Sheard, Zine-el-abidine Benaissa, and Emir Pasalic. 2000. DSL Implementation Using Staging and Monads. *SIGPLAN Not.* 35, 1 (dec 2000), 81–94. https://doi.org/10.1145/331963.331975

[10] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. 2011. Languages as Libraries. *SIGPLAN Not.* 46, 6 (June 2011), 132–141. https://doi.org/10.1145/1993316.1993514

[11] Baltasar Trancón y Widemann and Markus Lepper. 2021. LLJava Live at the Loop: A Case for Heteroiconic Staged Meta-Programming. In *Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (Münster, Germany) *(MPLR 2021)*. Association for Computing Machinery, New York, NY, USA, 113–126. https://doi.org/10.1145/3475738.3480942

[12] Baltasar Trancón y Widemann and Markus Lepper. 2016. LLJava: Minimalist Structured Programming on the Java Virtual Machine. In *Proc. Principles and Practices of Programming on the Java Platform (PPPJ 2016)*. ACM, 1–6. https://doi.org/10.1145/2972206.2972218

# Existential Containers in Scala

Dimitri Racordon
dimitri.racordon@epfl.ch
EPFL, LAMP
Lausanne, Switzerland

Eugene Flesselle
eugene.flesselle@epfl.ch
EPFL, LAMP
Lausanne, Switzerland

Matthieu Bovel
matthieu.bovel@epfl.ch
EPFL, LAMP
Lausanne, Switzerland

## Abstract

Type classes have been well-established as a powerful tool to write generic algorithms and data structures while escaping vexing limitations of subtyping with respect to extensibility, binary methods, and partial abstractions. Unfortunately, type classes are typically inadequate to express run-time polymorphism and dynamic dispatch, two features considered central to object-oriented systems. This paper explains how to alleviate this problem in Scala. We present existential containers, a form of existential types bounded by type classes rather than types, and explain how to implement them using Scala's existing features.

***CCS Concepts:*** • **Software and its engineering** → **Polymorphism**; *Inheritance*; **Abstract data types**; **Object oriented languages**.

***Keywords:*** polymorphism, dynamic dispatch, type class, dependent types

**ACM Reference Format:**
Dimitri Racordon, Eugene Flesselle, and Matthieu Bovel. 2024. Existential Containers in Scala. In *Proceedings of the 21st ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '24), September 19, 2024, Vienna, Austria.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3679007.3685056

## 1 Introduction

Generic programming consists of lifting general abstractions from their concrete implementations [11]. This process requires a mechanism to write polymorphic functions and data structures, that is to write code capable of operating on different types of values. In a statically typed setting, it further requires a way of describing a common interface to these values so that a type checker may guarantee that a particular operation is sensible, regardless of the run-time type of its operands. Subtype polymorphism is one of the most popular approaches to that goal, especially in the context of object-oriented languages.

In a system with subtyping, a type $\tau$ can be defined to be a *subtype* of type $\sigma$—typically written $\tau <: \sigma$—so that an instance of $\tau$ may be used in all places where values of type $\sigma$ are expected. The validity of the substitution hinges on the fact that $\tau$ is known to implement *at least* the interface of $\sigma$. This concept lays down solid foundations for generic programming: abstract data structures are expressed as interfaces meant to be extended by more specific definitions. For example, one may write a type `Shape` representing the concept of a polygon and have a subtype `Triangle` represent those composed of three edges. Presumably, any operation that accepts arbitrary shapes also accepts triangles.

Despite its significance for object-orientation and near ubiquitousness in contemporary programming languages, subtyping suffers from well-known limitations that hinder one's ability to write modular yet extensible abstractions [13, 18]. Borrowing a page from functional programming, most of these shortcomings are addressed by *type classes*, such as those found in Haskell [16]. A type class describes the interface of a general concept, say a polygon, as a set of top-level functions rather than methods bound to a particular receiver. These top-level functions may be overloaded for a specific type $\tau$, effectively specifying how $\tau$ *models* the abstract concept. Alas, while type classes elegantly fulfill the essential requirements of generic programming [15], they offer limited support for type erasure—the eliding of some type information at compile-time. Hence, it is difficult to manipulate heterogeneous collections or write procedures returning arbitrary values known to model a particular concept. Doing the same with subtyping is routine.

Existential types have been used to hide data representations since at least the 80s [7, 10] and play a pivotal role in the support of modular programs in languages like OCaml [5]. More recently, both Swift and Rust have described the signature of existential types and their implementations relying on type classes. Intuitively, given a type class instance $p$ defining an operation $f : \sigma \to \tau$ for a value $v$ of type $\sigma$, one can form an existential pair $\langle \sigma, \{v, p.f\} \rangle$ of type $\langle \exists X, \{X, X \to \tau\} \rangle$. This paper leverages this intuition in the context of Scala. We implement *existential containers*, a form of existentials bounded by a type class. It turns out that Scala's type system supports all the ingredients necessary for implementing this extension efficiently. In particular, we rely on extension methods, path dependent types [14], and implicit scopes.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*MPLR '24, September 19, 2024, Vienna, Austria*
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1118-3/24/09
https://doi.org/10.1145/3679007.3685056

Neither type classes nor existential containers are new ideas. Our contribution is to equip Scala with an ergonomic way to use dynamic dispatch in settings relying on type classes rather than inheritance hierarchies. Section 2 motivates this goal. Section 3 introduces key features of the Scala language that are used to express type classes and can be re-used to implement existential containers, as then described in Section 4. Section 5 discusses the expressiveness of our encoding and examines its performance on different benchmarks. Section 6 positions our work in the state of the art and Section 7 concludes.

## 2 Dynamic Dispatch and Type Classes

Subtyping is a popular approach to support polymorphism in object-oriented programming languages. Details vary from one language to the next but in general, one defines classes to denote abstractions that are extended to model refinements, either by adding properties and operations or by specializing them. A class is *abstract* if it declares operations without an implementation and *concrete* otherwise. Abstract classes hide details irrelevant to generic algorithms.

The left-hand side of Figure 1 shows an example. Line 3 declares an abstract class `Shape` denoting arbitrary polygons with two operations. The first returns the surface area of a shape and has no implementation. Concrete types extending `Shape` must provide one. The second compares shapes and has an implementation referring to their areas. Line 9 declares a concrete subclass of `Shape` denoting squares—i.e., quadrilaterals having edges of equal lengths and equal angles—and provides an implementation for computing their area.[1] Likewise, line 14 declares a concrete subclass denoting rhombuses—i.e., quadrilaterals having edges of equal lengths with no constraint on their angles–whose implementation is omitted for conciseness. Notice that the infix operator `<` of `Shape` is a generic algorithm. It can be applied to any pair of shapes, irrespective of their representations, as demonstrated by the main function. Line 21 asserts that a square whose edges have a length of 2 is smaller than a rhombus whose diagonals have lengths of 3 and 4, respectively.

### 2.1 Shortcomings of Subtyping

Considering only simple examples such as the one we just discussed, we may convince ourselves that subtype polymorphism is a panacea. Abstract classes define general concepts like `Shape`, methods of such classes define generic algorithms such as `<`, and concrete subclasses model concepts such as `SquareShape`. Unfortunately, more complex situations reveal crippling limitations of this approach. While universal bounded quantification [7]—i.e., generic parameters with type bounds—solve many of them, two flaws remain.

[1] We elaborate on the reason to represent the dimensions of the square as a separate object later in this section.

#### 2.1.1 Retroactive Modeling.

Looking at Figure 1, one may wonder why we defined a `SquareShape` as a class *wrapping* another data structure specifying the dimensions of a square. Why did we not define `Square` as a subclass of `Shape` directly? On closer inspection, note that we imported `Square` from a package. If we imagine that we are not the authors of that package—we may not even have access to its sources—then our choice of representation makes sense.

There is no way to *retroactively* add supertypes to a type. In other words, once a class like `Square` has been declared, its set of supertypes cannot change without also modifying the original declaration. This limitation is frustrating here because the authors of the `quadrilaterals` package have omitted the definition of an operation computing areas. While we can write it ourselves, we must use the adapter pattern to make `Square` notionally extend `Shape`.

This workaround is not ideal because it involves tedious boilerplate. Not only must we declare an additional class for each type we want to extend, we must also wrap and unwrap the contents of these classes to use methods from the original package, as line 21 demonstrates. We could improve the situation by writing a `scale` method for `RhombusShape` that forwards calls to the wrapped instance, but that would generate even more boilerplate. In a real setting, we would also have to duplicate documentation and test suites. This problem grows with the number of types we import and the number of new concepts we define. For example, we may import a `Drawable` concept from a package and a `Serializable` concept from another. Unless all shapes are drawable and serializable, we would eventually find ourselves implementing a `DrawableSerializableRhombusShape`!

#### 2.1.2 Binary Methods.

Figure 1 suggests that shapes have a total order. It may be tempting, therefore, to capture that observation in another concept that applies to shapes:

```
1 abstract class Comparable:
2   def <(s: Comparable): Boolean
```

Unfortunately, `Comparable` is a poor abstraction. To understand why, let us examine the signature of its unique operation. The operator requires that the second parameter be another instance of `Comparable`, meaning any value whose type is subtype of `Comparable` will do. However, it might not necessarily make sense to compare a shape to a character string. While there exist perfectly sensible orderings for these data types, meaning that both model a comparable concept, they do not imply the existence of a relation between a shape and a character string.

This issue is known as the *binary method problem* [6]. It states that subtyping struggles to safely express concepts having an operation of the form $\sigma \times \sigma \rightarrow \tau$. In a system where operations are written as methods of a single receiver, that means abstract concepts should avoid referring to themselves in contravariant positions. One solution is to check

```scala
import quadrilaterals.{Square, Rhombus}

abstract class Shape:
  def area: Double
  def <(s: Shape): Boolean =
    area < s.area


final case class SquareShape(
  dimensions: Square
) extends Shape:
  def area: Double = dimensions.side.squared

final case class RhombusShape(
  dim: Rhombus
) extends Shape: ...

@main def Main =
  val s = SquareShape(Square(2))
  val r = RhombusShape(Rhombus(1.5, 2))
  assert(s < RhombusShape(r.dim.scale(2)))
```

```scala
import quadrilaterals.{Square, Rhombus}

trait Shape extends TypeClass:
  extension (self: Self)
    def area: Double
    def <[S](s: S)(using S is Shape): Boolean =
      area < s.area

given Square is Shape as SquareIsShape:
  extension (self: Square)
    def area: Double = self.side.squared


given Rhombus is Shape as RhombusIsShape: ...



@main def Main =
  val s = Square(2)
  val r = Rhombus(3, 4)
  assert(s < r.scale(2))
```

**Figure 1.** Definitions of shapes using subtype polymorphism (left) and type classes (right) in Scala.

at run-time that the type of the method's argument is compatible with that of its receiver. However, that comes at the expense of static type safety as some invalid applications of the operator may no longer be caught at compile-time. Another approach consists of defining an auxiliary `Comparator` class parameterized by the type of the elements:

```scala
abstract class Comparator[T]:
  def inIncreasingOrder(l: T, r: T): Boolean
```

### 2.2 Type Classes

Type classes can be understood as a generalization of the trick we have just discussed. A type class describes the set of operations necessary to establish for any particular type to model a concept, thereby forming that concept's interface. For instance, `Comparator` describes types having a strict total order. Unlike a regular abstract base type, however, a type class defines its interface using free functions rather than members of the conforming type. In our example, that means `Comparator` declares a method operating on two instances of `T` rather than assuming the receiver is one of the two operands. The receiver of that method is merely a *witness* of the conformance of `T` to the concept.

In Scala, a type class is written as a trait defining the interface of another type, similar to `Comparator`. The latter is represented by a type member—i.e., a type that depends on instances of the trait [3]—named `Self`.

```scala
trait TypeClass:
  type Self
```

The right-hand side of figure 1 shows an example involving type classes. The trait at line 3 is a type class describing the same concept as the abstract class on the left-hand side using method extensions. Line 9 declares an instance of the type class for `Square`. Being defined as a *given*[2], this instance is made implicitly available to name lookup when a method is selected on an instance of `Square`. The details of this process are discussed in the following section. For now, simply observe that the compiler considers the extensions when it resolves the entity referred to by the infix operator < at line 21. Line 14 similarly declares another instance for `Rhombus`.

The main function evidences the advantage of having defined the concept of shapes as a type class. We can now manipulate instances of types imported from `quadrilaterals` directly, without wrapping or unwrapping them. Further, Scala offers a pleasingly familiar programming style that supports infix application and dot notation.

### 2.3 Dynamic Dispatch

Our example so far has painted a rather unfavorable picture of subtype polymorphism. There is nothing, it would appear, that type classes cannot express better than inheritance. We have, however, overlooked a key feature of most object-oriented systems: *dynamic dispatch*.

Dynamic dispatch delays the selection of the definition implementing a particular generic operation until run-time, when the operation is applied. Unlike static dispatch, which makes this choice at compile-time, this technique supports type erasure and lets variables hold different types of values at run-time. For example, one can write the following using the definitions on the left-hand side of Figure 1:

[2] *givens* were known as *implicits* in previous versions of Scala.

```
def smallest(xs: List[Shape]): Option[Shape] =
  xs match
    case Nil => None
    case a :: ys =>
      smallest(ys) match
        case None => Some(a)
        case Some(b) =>
          Some(if b < a then b else a)
val s = smallest(List(
  SquareShape(Square(2)),
  RhombusShape(Rhombus(1.5, 2))
))
```

This program defines an algorithm operating on a heterogeneous list of shapes, returning the smallest of its elements. Writing such a function is natural and unsurprising with subtyping. We use `Shape` to abstract over the specific types of the shapes populating the list and rely on dynamic dispatch to figure out at run-time which implementation of `area` must be called to compare instances. Writing the same function with type classes is not obvious, however, due to the lack of dynamic dispatch support. A close translation uses a generic parameter to abstract over the type of the list's contents but this approach does not accept heterogeneous lists. The fundamental issue is that we now need a witness of each element's conformance to the shape concept *and* a way to associate each element to the right witness statically. Returning an arbitrary shape is also problematic for the same reason. Not only must we return an element of the list—unless it is empty—we must also return the appropriate witness of its conformance to `Shape`. In summary, the solution conceptually looks like the following sketch:

```
def smallest(xs: List[(Any, AnyShapeWitness)])
    : Option[(Any, AnyShapeWitness)] =
  xs match
    case Nil => None
    case (va, wa) :: ys =>
      smallest(ys) match
        case None => Some((va, wa))
        case Some((vb, wb)) =>
          Some(
            if wb.<(vb)(va) then (vb, wb)
            else (va, wa)
          )
val s = smallest(List(
  (Square(2), SquareIsShape),
  (Rhombus(1.5, 2), RhombusIsShape)
))
```

The pair `(Any, AnyShapeWitness)` represents an arbitrary value along with a witness of a conformance to the shape concept. Together, these two elements let us perform dynamic dispatch, as line 10 illustrates. We call such a pair an *existential container* and spend the remainder of the paper discussing a safe and ergonomic implementation in Scala.

## 3 Contextual Abstractions in Scala

We ought to introduce a few features of Scala before we can delve into our implementation of existential containers, starting with context parameters. Code written with a functional flavor tends to require certain values to be passed around from one function to the next. These values represent the "context" in which a function runs, which may capture configuration settings or additional properties of other parameters. For example, consider the following:

```
class Tree(
  val id: String,
  val children: List[Tree]
):
  def depth(
    using parent: ParentRelationships
  ): Int =
    parent.get(this) match
      case None => 0
      case Some(p) => 1 + parent.depth

given ParentRelationships =
  computeBackwardLinks(root)

println(someChild.depth)
```

The class represents a directed tree: we can go from a node to its children but not to its parent. The method takes a context parameter denoting a table mapping each node of the tree to its innermost parent. We can use this parameter like any other but, additionally, notice it gets passed *implicitly* in recursive calls to `depth`, thus improving legibility. The value of `parent` is merely context in `depth`; it would be just a regular local binding if we had written the method with an imperative loop rather than a tail recursive call. Repeating this argument would therefore only introduce noise.

The compiler goes through a process called *implicit resolution* to identify which argument to pass to context parameters. A detailed description of this process is beyond the scope of this paper but in a nutshell, expressions live in an implicit scope populated by various constructs of the language. Those notably include context parameters and given instances as the one declared at line 12. Both may include extension methods.

Extension methods are another interesting feature, serving to retroactively "add" methods to a class without modifying its definition. They are not really members of the extended class, though. Rather, they are free functions, allowing selection with dot syntax and infix application. In the following, for instance, `encode` is added to the local scope of `Tree` so that it can be applied like any other method, as line 3 shows.

```
extension (t: Tree) def encode(): String =
  val s = t.children
    .map(_.encode()).mkString(" ")
  s"${t.id} ${s}"
```

Moreover, extension methods can be made available through the implicit scope. By using a contextual parameter containing a member which is an extension method, the latter becomes implicitly accessible within the body of the definition. Having combined contextual abstractions and extension methods, we can now better explain the right-hand side of Figure 1, discussed in Section 2.2. Recall the definition of the type class instance for `Square`, repeated below.

```
1  trait Shape extends TypeClass:
2    extension (self: Self)
3      def area: Double
4      def <[S](s: S)(using S is Shape)
5          : Boolean =
6        area < s.area
7
8  given Square is Shape:
9    extension (self: Square)
10     def area: Double = self.side.squared
11
12 assert(Square(1) < Square(2))
```

In this example, the type alias "`Square is Shape`" expands to "`Shape { type Self = Square }`"[3], which denotes a *refinement* of the type class restricting the type of `Self`, the conforming type. The given declaration introduces an instance of this refinement into the implicit scope as a witness of `Square`'s conformance to `Shape`, meaning that the compiler can now use the extension methods in `Shape` as long as the given instance is part of the implicit scope. This witness is passed implicitly as an argument to the operator's context parameter at line 12, allowing the compiler to resolve `s.area` at line 6.

## 4 Implementation

An existential container is a pair containing a value and a witness of its conformance to some type class. Expressing such a value in Scala is easy: just write "`(Square(1): Any, summon[Square is Shape] : Any)`", using `summon` to obtain a value of the given type from the implicit scope. This simple encoding, however, only gets us so far. Indeed, the erasure of the pair's elements—notice the widening to `Any`—makes it impossible to select any method. Even if we could, we would still have to convince the type checker that the first element is a valid argument for these methods. One workaround could be to widen the pair's elements to a tighter bound but this approach would rely heavily on subtyping and thus be subject to the limitations we discussed in Section 2.1. Fortunately, Scala has other tools to address these issues.

The key idea is to store the wrapped value as an instance of some *path dependent type* [14], which is already well-established as a way to implement existential types in Scala.[4]

[3]The `is` infix type operator is merely a standard library definition providing syntax sugaring.

[4]Note that this approach differs from Scala 2's Existential Types, which were dropped in Scala 3[1].

```
1  /** A value together with an evidence of its
2   *  type conforming to some type class. */
3  trait Container[Concept <: TypeClass]:
4    /** The type of the contained value. */
5    type Value : Concept as witness
6    /** The contained value. */
7    val value: Value
8
9  object Container:
10   /** Wraps a value of type `V` into a
11    *  `Container[C]` provided a witness that
12    *  `V is C`. */
13   def apply[C <: TypeClass](v: Any)
14            [V >: v.type](using V is C) =
15     new Container[C]:
16       type Value >: V <: V
17       val value: Value = v
```

**Figure 2.** Encoding of an existential container in Scala.

For example, the following trait notionally represents the type $\langle \exists X, \{\} \rangle$—i.e., a an existential type with no interface:

```
1 trait Existential:
2   type X
3   val value: X
4 val e: Existential = new Existential:
5   type X = Int
6   val value = 1
```

As the ascription on the declaration at line 4 suggests, `e` is an instance of `Existential`. Since `X` is a type member of that trait, we know that there exists a type `e.X` and that it is the type of `e.value`. We know nothing more. Even so, we can extract the existential value with "`val v: e.X =e.value`" and retain its type simultaneously. Crucially, we know the type of `v` *depends* on `e`, meaning that the compiler can use information stored in `e` to conclude facts about `e.X`. The last piece of the puzzle is to store a witness of `e.X`'s conformance to a type class.

### 4.1 Encoding

Our encoding is shown in Figure 2. The trait essentially composes two types and two values: a type `Value` with a corresponding term `value` of that type, and a type `Concept` with a witness of the conformance to that concept for the contained value. `Concept` is a simple parameter of the trait and is associated only to the type of the container being defined. `Value`, on the other end, is a type member since it *depends* on each container instance, similar to `Existential .X` in our previous example. Additionally, `Value` is context-bounded by `Concept`, meaning that instances of `Container` must provide a term of type "`Value is Concept`" named `witness`—i.e. an evidence of the conformance.

Suppose we have a value `c` of type `Container[Shape]`. We could access its parts using explicit selections and combine

the two to obtain the area of the wrapped value. Since the witness is declared as a context bound, however, it is in the implicit scope of the container's value and its selection can be elided. In effect, a selection of an extension method defined by the interface of the container—e.g., c.value.area—is resolved by the compiler to use the witness given within c implicitly. The selection of value can also be elided with a modest change in the compiler. Calls of the form c.m are transformed to c.value.m when c is a container and the selection does not type-check without being adapted. This fallback mechanism ensures that explicit selections of a container's members are still possible.

Building a container comes down to the following:

```
1 new Container[Shape]:
2   type Value = Square
3   val value = Square(2)
4   val witness = SquareIsShape
```

Here too, the witness definition can be omitted since it is declared as a context bound and SquareIsShape is in the context. The remaining boilerplate can be replaced by "Container[Shape](Square(2))" or, if Shape can be inferred from the expected type, "Container(Square(2))". These expressions call the apply method of Container's companion object and are equivalent to the "wrap" or "pack" primitive featured in formal definitions of existential types. This expansion is enabled by the design of the method's signature. The first type parameter list is [C <: TypeClass] for consistency with the trait declaration. The witness is not in the first term parameter list so that its Self can be constrained by the value being wrapped before looking for it in the context, thereby avoiding ambiguities. Putting everything together, we can write a well-typed version of the program we sketched at the end of Section 2:

```
1  def smallest(xs: List[Container[Shape]])
2      : Option[Container[Shape]] =
3    xs match
4      case Nil => None
5      case a :: ys =>
6        smallest(ys) match
7          case None => Some(a)
8          case Some(b) =>
9            Some(if b < a.value then b else a)
10 val s = smallest(List(
11   Container(Square(2)),
12   Container(Rhombus(1.5, 2))
13 ))
```

The method accepts a heterogeneous list of shapes represented and returns the smallest unless the list is empty. It is called at line 10, where its argument is constructed by wrapping different shapes along with their respective conformances to Shape. The body of the method reads almost the same as if we had used subtyping with one difference. We must write a.value rather than just a for the argument of the operator < at line 9. The next section explains why.

### 4.2 Opening Containers

Perhaps unintuitively, Container[C] is not itself a type conforming to C since an existential container does not conform to its own bound. Recall the binary method problem to understand why. Imagine we define a type class C modeling comparable values and consider two values of types Int and String, respectively, each conforming to C. We could then create two different existential containers wrapping 42 and "forty-two", respectively. Although both instances would have type Container[C], they should not be comparable, and therefore Container[C] cannot conform to C.

Coming back to the above example, we can now observe that a call of the form b < a would not type check. Indeed, the operator requires the witness of a conformance to Shape for its right operand (see Figure 1) but such a witness does not exist for Container[C]. By passing a.value however, we pass an argument of type a.Value, of which the implicit scope includes the witness given in a. More formally, accessing a.value *opens* the existential while retaining the type a.Value, effectively keeping an anchor—the path to the scope of the witness—to the interface of the type class. This example showcases the power of path dependent types. The ability to use an opened existential without escaping static type safety—i.e., without unchecked type coercions—does not require any amendment to the type system.

### 4.3 Using Containers over Generic Parameters

While the ability to open an existential grants us enough expressiveness to call a generic function with a context bound, we could instead define a non-parametric version of our operator, using existential containers, to apply b < a without any explicit opening. Taking another look at the signature, we notice that we actually care neither about which specific type of shape we receive nor about the accompanying type class instance, as long as get to call area on an object representing a shape. An existential container gives the same guarantees with a shorter spelling:

```
1 // Existential version
2 def <(s: Container[Shape]): Boolean =
3   area < s.area
4 // Parameteric version
5 def <[S](s: S)(using S is Shape): Boolean =
6   area < s.area
```

This existential definition is arguably simpler than its parametric counterpart. The ascription on s in the former completely defines the type of the arguments that the method expects. In contrast, the type of s in the latter is the composition of three elements: the generic parameter declaration, the context parameter, and the ascription itself. While a generic parameter may be sometimes required, e.g., to address binary methods, an existential container is more appropriate when a method simply expects a value of *some* type with a compatible type class instance.

# 5 Evaluation

This section discusses the applicability of our implementation. We start by studying the expressiveness of existential containers in a use case more sophisticated than the examples we have seen so far. We then measure the performance of our implementation in a realistic setup and compare it with subtyping before discussing some limitations.

## 5.1 Expressiveness

So far we have only examined type classes with a single parameter: the conforming type. It is likely, however, to encounter type classes having more than one parameter. We call those *associated types*, following the steps of Järvi et. al on generic programming [9]. To illustrate, consider the following example.

```
1 trait Archivable extends TypeClass:
2   type Encoder
3   type Decoder
4   extension (self: Self)
5     def encode()(using encoder: Encoder): Unit
6     def decode()(using decoder: Decoder): Self
```

`Archivable` defines an interface for serializable objects by means of two methods; one for serialization and the other for deserialization. The signature of these methods refers to `Encoder` and `Decoder`, which are associated types declared as path dependent types.

We can certainly create a heterogeneous list of archivable objects, as we have already demonstrated. To write a function serializing the contents of this list, however, we need a more precise type since each element could conform to `Archivable` for a different type of encoder. In other words, we must constrain each element's witness to have the same associated type without constraining the type of their value. Fortunately, a simple refinement suffices:

```
1 def encodeAll[E](
2   xs: List[
3     Container[Archivable { type Encoder = E }]
4   ]
5 )(using E): Unit =
6   xs.forEach((x) => x.encode())
```

Two important observations can be made on this example. First, it shows that our encoding of implementation containers fits perfectly into Scala's existing support for type refinements, letting us attach equality and/or conformance constraints to associated types. That is an essential requirement for generic programming [15]. Second, notice that the signature of `encodeAll` does not have to mention anything about the *decoder* of its arguments. Only the relevant parameters of the interface need to be specified; the others can remain abstract. That is in contrast with approaches based on universal bounded quantification which typically require all parameters to appear in type constructors, along with a sensible bound. In some instances, finding such a bound is, in fact, impossible and one must resort to Java-like wildcards, which come with well-known soundness issues [4].

## 5.2 Performance

The main question concerning the performance of our implementation is whether calling methods on existential containers incurs significant overhead compared to traditional dynamic dispatch. We ran two benchmarks to test this overhead. The first is a micro benchmark focusing only on the dispatch time of an operation. The second is a macro benchmark aiming at measuring the overhead in a realistic scenario. We implemented two versions of each benchmark: one uses traditional subtyping and the other uses existential containers. We also implemented similar benchmarks in Swift to provide an inter-language perspective. The source code for these programs is available as a companion artifact. Hereafter, we present and discuss measures from 10,000 iterations across 4 runs for each version[5].

### 5.2.1 Micro Benchmark.

The results of the micro benchmark are presented in Figure 3. We observe that dispatching an operation through existential containers in Scala is about twice as slow as traditional virtual method dispatch. This can be explained by the extra pointer indirection. On the JVM, the runtime increases with the number of classes. For the Existential version, it ranges from approximately 5 ms for 2 classes to around 8 ms for 10 classes, and stays stable for a higher number of classes. For the Inheritance version, we observe similar curves, ranging from about 2.5 ms to 5 ms. The Swift measures are stable across the number of distinct classes, which is expected due to the lack of just-in-time compilation. Using inheritance in Swift is especially slow due to the reference counting overhead for classes.

### 5.2.2 Macro Benchmark.

To test the performance of existential containers in a real-world scenario, we have implemented a small (≈1500 lines of code) 3D collision engine, which we use to detect object occlusion from a particular point of view. The shapes of different 3D objects are approximated by idealized mathematical volumes, such as spheres and cuboids, positioned randomly within an imaginary bounding box. Occlusion is then detected by shooting a ray in the direction of each object and checking whether another object lies in the way. Each collision shape has its own data representation—e.g., a sphere is merely a radius—and implements its own collision detection algorithm. To select

[5]We ran both the *micro benchmark* and the *macro benchmark* 4 times, alternating versions. Each run comprised 2,500 warm-up iterations and 2,500 measurement iterations, totaling 10,000 measurements. The benchmarks were executed on a MacBook Pro with a 2.6 GHz 6-Core Intel Core i7 processor, using Swift 5.10.0.13, Swift Benchmark 1.23.1, a nightly version of Scala 3.5.0 and JMH 1.37 through sbt-jmh 0.4. The "openjdk" JVM is Adoptium 1.21.0.3, and "graal" is GraalVM 21.0.2. The benchmark results were processed and visualized using the Seaborn, Matplotlib, and Pandas Python libraries.

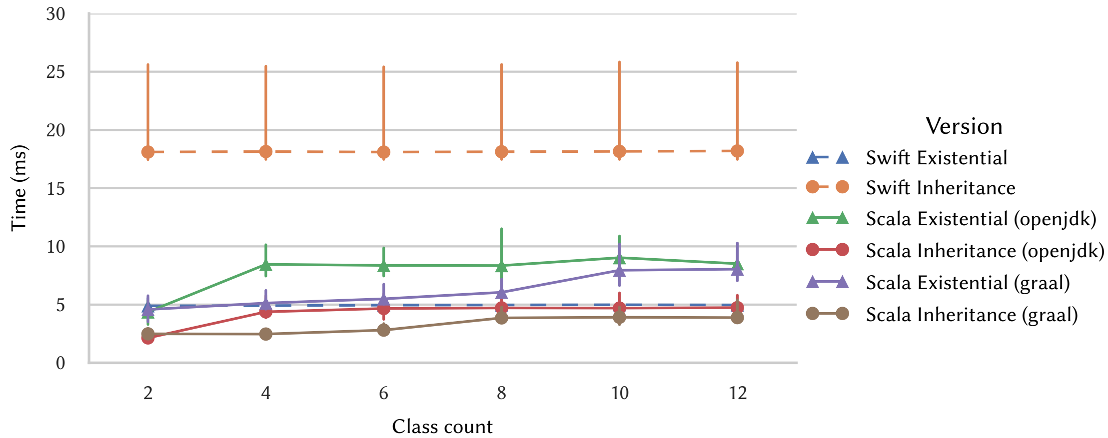



**Figure 3.** Micro benchmark results. This benchmark measures the time to apply a simple operation (adding the value of a class field to a constant) to 10 million objects in a contiguous array. The operation is modeled using a type class and existential containers in the Existential versions, while it is modeled using a virtual method in the Inheritance version. The graph shows the runtime in milliseconds for the 10 million operations, as a function of the number of different object types present in the array. The error bars show the 99th percentile, while the markers represent the measurement means. Lower is better.

the right implementation at run-time for each element of the heterogeneous collection of shapes, we use existential containers in one version and classes with inheritance in the other. The program terminates after a ray has been shot at every object. Looking at the results, we observe that the overhead of the existential version in Scala is 6% for OpenJDK and 2% for GraalVM.

### 5.3 Limitations

An important limitation of our encoding is that an existential container can only have exactly one bound; that is, it can only wrap a single evidence. As a consequence, one cannot easily express a type representing, for instance, any shape that is also archivable. One workaround is to define a new type class that extends both concepts, which in Scala can be done with an intersection type. This approach has one problem, however. It induces significant boilerplate to provide and manipulate witnesses, as it is not obvious that a type conforming to both $T$ and $U$ automatically conforms to a type class $T\&U$ without manually aggregating witnesses. Further, that has to be done for all different pairs of type classes for which an aggregate is needed.

## 6 Related Work

JavaGI [17] is an extension of Java that supports *interface types*, a generalization of Java's interface supporting retroactive modeling, type-safe binary methods, implementation inheritance, and dynamic dispatch. These features closely match the use cases of type classes in Scala.[6] A significant difference is that, unlike a type class, an interface type defines its operations in terms of a receiver—i.e., the "`this`" parameter in Java-family languages—rather than methods of an ad-hoc object. This approach is more in line with classical interfaces: whereas a type class instance *witnesses* the conformance of a type to some concept, an interface type instance *is* a model of some concept. Hence, interface type can be used naturally to express polymorphic functions and data structures, whereas one must use existential containers to achieve the same with type classes. In contrast to existential containers, however, JavaGI's interfaces cannot be used as types if they define associated types.

Similar efforts include Genus [19], another extension of Java that borrows from early designs of C++ concepts [8, 15] to express type classes as sets of type constraints. Unlike JavaGI, Genus supports non-uniquely witnessed conformances along with language mechanisms to resolve ambiguities. Similar expressiveness can be achieved naturally with our encoding in Scala by passing arguments to context parameters explicitly.

As we have already mentioned, Swift advocates for a type class oriented programming style and uses existential containers for dynamic polymorphism. While our approach requires minimal change to the Scala language and its compiler, Swift's built-in support for existential types offers advantages with respect to ergonomics. Specifically, the subtyping

[6] The intersection between Scala's type classes and JavaGI's interfaces is even larger and covers features we have not discussed in this paper, notably including static methods and conditional conformance.

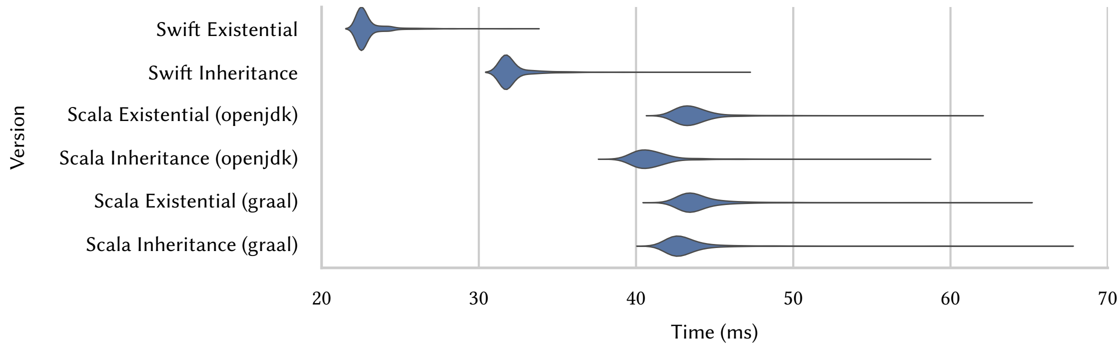



**Figure 4.** Graph of the *macro benchmark* results. This benchmark measures the time to shoot a ray at every shape in a contiguous array of 1000 shapes. As in the micro benchmark, operations are modeled using a type class and existential containers in the Existential versions, while they are modeled using a virtual method in the Inheritance version. The graph uses a violin plot to compare the distributions of the runtimes (in milliseconds) for each version. Closer to the left is better.

relationship can be extended so that a type `T` can be considered subtype of `any P`, where `P` is a type class to which `T` conforms. As a result, the wrapping/unwrapping of values into/from existential containers interacts more organically with the language's syntax. Further, Swift can express compositions of type classes, therefore providing built-in support for existential containers bound by more than one type class. This system is formally described by Racordon and Buchs [12], though their work does not consider constraints on associated types.

Rust also supports existential containers as a built-in feature, where `dyn P` is similar to Swift's `any P`. However, it imposes heavy restrictions on the definition of the traits that can be used with `dyn`. In particular, these cannot contain static or generic methods [2]. Our approach does not suffer from any of these limitations.

## 7 Conclusion

We have presented an extension of the Scala programming language to support *existential containers*, a form of existential types bounded by type classes rather than types. These containers are represented as a dependent pair $\langle v, w \rangle$ where $w$ is an instance *witnessing* the conformance of $v$'s type to some type class. Our approach exploits Scala's existing features to not only express such a dependent pair but also use the operations defined by the type class in a natural way, which has two advantages. First, the soundness of the theoretical foundations upon which these features are built [4] extends to our work, which requires no change in the metatheory of the language. Second, the fact that our encoding only involves a modest desugaring transformation in the compiler gives us confidence that extending Scala with existential containers does not introduce bugs due to unforeseen interactions with other features.

The two essential features we have used, namely path dependent types and context parameters, are not exclusive to Scala. Hence, our approach could apply to a number of other programming languages.

We have motivated the need for existential containers, discussed their practicality, and studied their efficiency. While our examples demonstrate the gain in expressiveness, performance results reveal that dynamic dispatch through existential containers incurs noticeable overhead over traditional virtual dispatch. Our experiments show that such an overhead might be affordable in applications not bottlenecked by method dispatch. Nonetheless, these results suggest that more engineering effort should be spent on optimizing calls to extension methods. Other future works include an extension of our encoding to support existential containers bound by more than a single type class. We believe that existential containers open the door to alternative ways to design libraries that may suffer less from the extensibility limitations of subtyping and plan on proposing our encoding and compiler changes to be included in future versions of Scala.

## Acknowledgments

This project has been funded by the Swiss National Science Foundation project entitled "Capabilities for Typing Resources and Effect" (TMAG-2_209506/1).

## References

[1] [n. d.]. Dropped Scala 2 Existential Types. https://docs.scala-lang.org/scala3/reference/dropped-features/existential-types.html Accessed 16 July 2024.
[2] [n. d.]. Object Safety–The Rust Reference. https://doc.rust-lang.org/reference/items/traits.html#object-safety Accessed 16 July 2024.
[3] Nada Amin, Tiark Rompf, and Martin Odersky. 2014. Foundations of path-dependent types. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Andrew P. Black and Todd D. Millstein (Eds.). ACM, 233–249. https://doi.org/10.1145/2660193.2660216

[4] Nada Amin and Ross Tate. 2016. Java and scala's type systems are unsound: the existential crisis of null pointers. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 838–848. https://doi.org/10.1145/2983990.2984004

[5] Clément Blaudeau, Didier Rémy, and Gabriel Radanne. 2023. Retrofitting OCaml modules. In *Journées Francophones des Langages Applicatifs (JFLA)*. 59–100.

[6] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, Jonathan Eifrig, Scott F. Smith, Valery Trifonov, Gary T. Leavens, and Benjamin C. Pierce. 1995. On Binary Methods. *Theory and Practice of Object Systems (TAPOS)* 1, 3 (1995), 221–242.

[7] Luca Cardelli and Peter Wegner. 1985. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computer Surveys* 17, 4 (1985), 471–522. https://doi.org/10.1145/6041.6042

[8] Douglas P. Gregor, Jaakko Järvi, Jeremy G. Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. 2006. Concepts: linguistic support for generic programming in C++. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Peri L. Tarr and William R. Cook (Eds.). ACM, 291–310. https://doi.org/10.1145/1167473.1167499

[9] Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine. 2005. Associated types and constraint propagation for mainstream object-oriented generics. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Ralph E. Johnson and Richard P. Gabriel (Eds.). ACM, 1–19. https://doi.org/10.1145/1094811.1094813

[10] John C. Mitchell and Gordon D. Plotkin. 1988. Abstract Types Have Existential Type. *ACM Trans. Program. Lang. Syst.* 10, 3 (1988), 470–502. https://doi.org/10.1145/44501.45065

[11] David R. Musser and Alexander A. Stepanov. 1988. Generic Programming. In *Symbolic and Algebraic Computation, International Symposium (ISSAC) (Lecture Notes in Computer Science, Vol. 358)*, Patrizia M. Gianni (Ed.). Springer, 13–25. https://doi.org/10.1007/3-540-51084-2_2

[12] Dimi Racordon and Didier Buchs. 2020. Featherweight Swift: a Core calculus for Swift's type system. In *ACM SIGPLAN International Conference on Software Language Engineering (SLE)*, Ralf Lämmel, Laurence Tratt, and Juan de Lara (Eds.). ACM, 140–154. https://doi.org/10.1145/3426425.3426939

[13] John C. Reynolds. 1978. *User-Defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction*. Springer New York, New York, NY, 309–317. https://doi.org/10.1007/978-1-4612-6315-9_22

[14] Tiark Rompf and Nada Amin. 2016. Type soundness for dependent object types (DOT). In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 624–641. https://doi.org/10.1145/2983990.2984008

[15] Jeremy G. Siek and Andrew Lumsdaine. 2005. Essential language support for generic programming. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Vivek Sarkar and Mary W. Hall (Eds.). ACM, 73–84. https://doi.org/10.1145/1065010.1065021

[16] Philip Wadler and Stephen Blott. 1989. How to Make ad-hoc Polymorphism Less ad-hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*. ACM Press, 60–76. https://doi.org/10.1145/75277.75283

[17] Stefan Wehr. 2010. *JavaGI: a language with generalized interfaces*. Ph. D. Dissertation. University of Freiburg. http://www.freidok.uni-freiburg.de/volltexte/7678/

[18] Stefan Wehr and Peter Thiemann. 2011. JavaGI: The Interaction of Type Classes with Interfaces and Inheritance. *ACM Transactions on Programming Languages and Systems* 33, 4 (2011), 12:1–12:83. https://doi.org/10.1145/1985342.1985343

[19] Yizhou Zhang, Matthew C. Loring, Guido Salvaneschi, Barbara Liskov, and Andrew C. Myers. 2015. Lightweight, flexible object-oriented generics. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, David Grove and Stephen M. Blackburn (Eds.). ACM, 436–445. https://doi.org/10.1145/2737924.2738008

Received 2024-05-25; accepted 2024-06-24

# Quff: A Dynamically Typed Hybrid Quantum-Classical Programming Language

Christopher John Wright
University of Manchester
Manchester, United Kingdom
christopher.wright-5@postgrad.manchester.ac.uk

Mikel Luján
University of Manchester
Manchester, United Kingdom
mikel.lujan@manchester.ac.uk

Pavlos Petoumenos
University of Manchester
Manchester, United Kingdom
pavlos.petoumenos@manchester.ac.uk

John Goodacre
University of Manchester
Manchester, United Kingdom
john.goodacre@manchester.ac.uk

## Abstract

Current strategies for quantum software development still exhibit complexity on top of the already-intricate nature of quantum mechanics. Quantum programming languages are either restricted to low-level, gate-based operations appended to classical objects for circuit generation, or require modelling of quantum state transformations in Hilbert space through algebraic representation.

This paper presents the Quff language which is a high-level, dynamically typed quantum-classical programming language. The Quff compiler and runtime system facilitates quantum software development with high-level expression abstracted across the quantum-classical paradigms. Quff is constructed on top of the Truffle framework which aids the implementation and efficiency of the stack, while reusing the JVM infrastructure. The presented comparisons display that Quff lends itself as an effective, easy-to-use solution for the development of executable quantum programs with automatic circuit generation and efficient computation.

***CCS Concepts:*** • **Computer systems organization** → **Quantum computing**; • **Software and its engineering** → **Formal language definitions**; **Domain specific languages**; **Compilers**; **Runtime environments**; **Object oriented frameworks**.

***Keywords:*** Quantum Computing, Quantum Programming, Compilation, Runtime Systems, Dynamically-typed Programming, Intermediate Representation, Java on Truffle, GraalVM

This work is licensed under a Creative Commons Attribution 4.0 International License.

*MPLR '24, September 19, 2024, Vienna, Austria*
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1118-3/24/09
https://doi.org/10.1145/3679007.3685063

**ACM Reference Format:**
Christopher John Wright, Mikel Luján, Pavlos Petoumenos, and John Goodacre. 2024. Quff: A Dynamically Typed Hybrid Quantum-Classical Programming Language. In *Proceedings of the 21st ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '24), September 19, 2024, Vienna, Austria.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3679007.3685063

## 1 Introduction

Quantum computing offers theoretical improvements over classical systems in a multitude of areas [21, 28, 29, 36, 40, 42, 43, 60]. Leveraging the principles of quantum mechanics to process and store information, the field fundamentally differs from classical computing - posing novel challenges on current mainstays of software development practices [1, 12, 15, 32, 50].

Describing intricate high-level quantum algorithms with primitive quantum operations does not lend itself to effective software development practices [65], yet such a strategy remains the focal point of many quantum software development practises [4, 22, 27, 31, 34, 39, 70]. Various attempts have been made to alleviate this quandary with some form of abstraction. For example, through raising low-level operations to perform over larger data constructs [22, 31, 56], or enabling specific algorithmic calls via API functions [5, 72]. Others have tried to elaborate type systems for compile-time quantum safety [8, 65, 80].

The contemporary quantum computing can be characterised as being in a Noisy Intermediate-Scale Quantum (NISQ) era, exemplified by quantum devices that have highly limited, error-prone qubits and imperfect gates [41, 45, 61, 69]. This leaves computations which are strictly quantum, such as Shor's Factoring algorithm [20, 68], out of the picture. Thus, most present-day quantum applications are hybrid in nature by bringing together classical systems with quantum devices as small-scale accelerators [26, 40, 46, 58, 71, 74]. As a result, there is a need for hybrid programming languages that seamlessly merge the classical and quantum realms, ensuring efficient data exchange and task distribution between

the two, while being able to harness quantum accelerators [59, 66].

This paper describes Quff (*"cuff"*); a high-level dynamically typed programming language and runtime system for hybrid quantum-classical software development. Using a novel form of abstraction over the classical and quantum paradigm, Quff provides a reliable bridge between the two. Allowing a developer to focus on algorithmic logic and program correctness, rather than remaining perplexed by the underlying quantum mechanics and convoluted syntax of current quantum programming. This paper delves deep into the philosophy and design of Quff for furthering the adoption of hybrid quantum software development.

The main contributions of Quff can be summarised as:

- An intuitive high-level, dynamically-typed, quantum-classical programming language enabling the description of all quantum computations with only a single extra quantum keyword, using a syntax familiar to most classical programmers;
- Novel strictly-quantum subset of language denotional semantics and run-time type inference rules which dictate the flow of quantum states through a Quff program;
- A novel compiler and runtime system which leverages the Quff language to provide quantum abstraction, enabling more succinct expression of quantum computations to improve software development; and
- An expressive quantum intermediate representation for high-level computations written in Quff, allowing swift automatic *uncomputation* (definition in Section 2) and decomposition to low-level quantum circuits.

Section 2 gives a brief introduction to quantum computing and quantum programming, along with the GraalVM [53, 54] and the Truffle framework [55] used for language creation, execution and optimization. Section 3 details the Quff the language design, syntax, semantics & quantum-specific typing rules in 3. Section 4 describes the compiler & runtime system. Here, the novel internal quantum intermediate representation enables new lazy evaluations with optimizations for *uncomputation*, as well as the simulation process, is described. Due to space limitations we do not present the full language syntax and semantics, only the parts specific to quantum computations, and the dynamic transition between classical and quantum data. Section 5 uses the quantum Grover's algorithm to illustrate Quff. Section 6 then provides a side-by-side analysis between Quff and two quantum languages; the most popular quantum development toolkit by IBM, Qiskit [5], and a recent type-safe quantum language Silq [8]. As an example, Figure 9 illustrates for a well-known quantum computation the succinctness of Quff (one single line of code), and compares it against Qiskit A.1.1 (more than 20 lines of code). Section 7 then explores related work, and finally the summary in section 8.

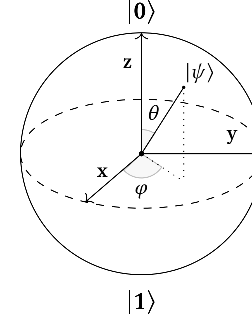



**Figure 1.** The Bloch Sphere: a 3D representation of a qubit, where $\alpha = \cos\left(\frac{\theta}{2}\right)$ and $\beta = e^{i\varphi}\sin\left(\frac{\theta}{2}\right)$ in $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$.

## 2 Background

### 2.1 Quantum Fundamentals

**2.1.1 Qubits.** As the quantum counterpart to the classical bit, qubits form the fundamental basis of quantum computing and are the reason for the distinct contrast between the two paradigms. Instead of taking the exact state **0** or **1**, its state is instead represented by a linear combination over two states, $|0\rangle$ and $|1\rangle$;

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \qquad \alpha, \beta \in \mathbb{C} \qquad |0\rangle = \begin{pmatrix}1\\0\end{pmatrix} \quad |1\rangle = \begin{pmatrix}0\\1\end{pmatrix}$$

The complex nature of the coefficients leads to a further degree of freedom as the global phase, which is typically normalized for a real $\alpha$ and a complex $\beta$. Thus lending to a graphical representation of the state known as the Bloch sphere, as seen in Figure 1.

**2.1.2 Measurement.** One can then assess the state of a qubit along any plane within this sphere, but is typically done along the $z$-basis - commonly known as the computational basis. Since in the classical realm we may only have a state **0** or **1**, measurement is said to collapse the qubit, resulting in one of the two states, with a probability given by the square magnitude of the corresponding coefficient. This leads to the constraint $|\alpha|^2 + |\beta|^2 = p_z(0) + p_z(1) = 1$.

**2.1.3 Operations.** Following this, single qubit gates correspond to rotations around any of the three axes, leading to the common **X**, **Y** and **Z** Pauli gates, which are 180° rotations around the corresponding axis. The **H** gate (known as the Hadamard gate) consists of a 90° rotation about the $x$-axis, followed by another 90° rotation about the $z$-axis - thus taking a qubit in the state $|0\rangle$ into a state halfway between $|0\rangle$ and $|1\rangle$ in the positive (right) direction;

$$H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = |+\rangle$$

Furthermore, gates may be applied to some qubit, conditional to the state of another (typically when it is in the state $|1\rangle$). This does not result in measurement of the control

qubit, but may be seen as being executed in the realm where the qubit is in the necessary state. Further understanding of this can be gained through through the use of Feynman diagrams [16].

These two qubits are then said to be *entangled*, meaning the state of each cannot be described independent to the other. Therefore, when one of these qubits is measured, we instantly know the state of the other; the state of both qubits will have collapsed according to the path the computation took.

#### 2.1.4 Clifford+T Gate Set.

This gate set is a universal subset of quantum gates with which all quantum computation can be described [81]. It is typically used for decomposition down to an abstract circuit which has no detail about the specific quantum hardware to be used - a circumstance dealt with typically by a later, hardware-specific quantum compiler. The set consists of the gates $X, Y, Z, CX, T, T^\dagger, S, S^\dagger$.

#### 2.1.5 Circuits.

The composition of gates over a variety of qubits is a circuit, which describes the steps involved in a quantum computation at a fundamental level. These are diagrammatically represented with an $\boldsymbol{x}$-axis representing time, qubit lines as rows, and gate operations placed on the rows at discrete points. Such computations often consist of three main basic stages: *state preparation*, in which a qubit system's state is pushed from $|0\ldots0\rangle$ to some required initial state $|\psi\rangle$; *state driving*, in which the state is then operated on to produce some desired solution state; and *measurement*, in which specific qubits are measured to reveal the required solution to the problem with some probability.

### 2.2 Constraints

**No-cloning theorem** The no-cloning theorem states that the state of a qubit may not be duplicated exactly, creating a direct contrast to the classical paradigm. However, one can perform *weak* quantum copying; $|\psi\rangle \otimes |0\rangle^{\otimes n} \mapsto |\psi\rangle \otimes |\psi'\rangle$, where $|\psi'\rangle$ is in the same computational state as $|\psi\rangle$, but with a different phase.

**Reversibility** Due to the nature of quantum systems, quantum computations must be reversible, and so operations consist of unitary matrices with a one-to-one relation between inputs and outputs. As such, ancillary qubits are often required to performed some operations, for example a logical `AND` gate.

**Uncomputation** Due to the phenomenon of entanglement, quantum variables can not simply be reset back to **0**, nor can they be measured and set, as both would affect any other entangled quantum systems. They must instead be uncomputed, a process in which a set of operations is performed which corresponds to the inverse of its previous operations, taking its state back to $|0\rangle$.

### 2.3 Further Information

Further information on the fundamentals of quantum computation, such as quantum information theory, Hilbert spaces, quantum hardware etc. are out of the scope of this paper, and so the reader is kindly directed to resources such as [7, 35, 52].

### 2.4 Quantum-Classical Computation

#### 2.4.1 Hybrid Circuits.

Computations which perform quantum operations based on classical conditions are known as hybrid circuits. Such circuits pose a further challenge in both high- and low-level program representation, as they require close control and compatibility with both the classical computer and quantum system, leading to a paradigm known as *classical control, quantum data* [65]. This may be expanded further to involve conditions based on measurement results, known as measurement-based quantum computation.

#### 2.4.2 Hybrid Programs.

At a higher level, hybrid quantum-classical programs consist of some quantum circuit and a parenting classical program. The classical program utilises the circuit in part of its computation, and may repeat such circuit, often changing parameters each time, as much as it requires. This process is one of the main applications for current-day NISQ systems, as it typically requires smaller circuits - both in terms of *depth* (number of operations) and *width* (number of qubits required).

### 2.5 Quantum Software Engineering

#### 2.5.1 Key Features.

Current quantum programming languages can be split into two categories [19, 27, 44]: circuit-based or state-based. The former typically consists of building up circuit representations consisting of numerous quantum gates over multiple qubits, often performed using an API for circuit objects, gate functions and simulation. On the other hand, state-based languages require mathematical modelling of state transformations through a system's Hilbert space to perform computation, often using a functional paradigm.

#### 2.5.2 Program Correctness.

Ensuring quantum program correctness is an in-depth and complicated task [18, 47, 48, 62, 63, 79]. Even with understanding of the quantum algorithm to implement, a programmer requires knowledge on low-level gates, quantum states, API usage, optimisation strategies and platform capabilities to develop programs [6]. Furthermore, debugging large, complex circuits quickly gets out of hand due to the enormous amount of low-level operations needed when using current quantum development platforms, with additional complication when viewed through the lens of efficient quantum resource usage [3, 49].

### 2.6 GraalVM, Truffle & TornadoQSim

GraalVM [53, 54, 77] is a virtual machine and compiler infrastructure providing a shared platform for language implementations. It allows for dynamic optimizations where nodes in an AST structure may rewrite themselves during interpretation, alongside speculative optimization and deoptimization [78].

Truffle [55] is the language implementation framework which lies on top of GraalVM and is implemented in Java. Utilization of this framework provides a high-performance interpreter alongside a DSL annotation processor [30] for efficient language creation, with a focus on dynamically-typed, imperative programming languages. Its shared structure allows guest languages to inter-operate with one another, allowing for multi-faceted program design [23].

TornadoQSim [38] allows for the description and simulation of quantum circuits with GPU acceleration; simulation of quantum computations is performed using complex matrix multiplication, and so naturally lends itself to speedup using GPUs.

## 3 Quff Design

### 3.1 Philosophy

In order to aid the adoption of quantum programming and ease the learning curve required to jump from classical software engineering to quantum, efforts need to be made in the creation of user-friendly modalities for the expression of quantum computation, while not compromising on the power of the topic. Furthermore, due to the highly constrained nature of present NISQ devices, quantum programs must focus on minimal usage of quantum resources, and preferably utilise a hybrid programming approach to solving some problem. This leads to the main governing philosophy of the Quff language: *simplistic abstraction* alongside *complete functionality.*

### 3.2 Structure

The layout of a program written in Quff is similar to that of current, common classical programming languages, such as Python [17], JavaScript [10] and Kotlin [2], where programs define classes and functions, each containing high-level statements and expressions in an object-oriented, imperative structure. Program execution then starts from a defined `main()` method. The language revolves around a dynamic type system and the principle of *duck typing* (cross-hierarchy polymorphism) akin to Python.

For readability sake, code written in the Quff language in this paper uses colors to highlight language constructs:

- Blue represents keywords, e.g. `fun` and `if`;
- Green represents types, e.g. `UInt` and `QReg`;
- Teal represents built-in classical functions, e.g. `main` and `println`;
- Red represents the quantum keywords `q:` and `H:`;
- Violet represents built-in quantum functions representing low-level operations, e.g. `Z` and `CX`.

### 3.3 Syntax

The language's syntax resembles common classical dynamic languages. Statements may consist of typical constructs, such as constant and variable declaration with `val` and `var` ; assignment; function application with `f(a)`; conditional branching with `if-else` and `when` (i.e. switch-case) statements; looping with `for` and `while` loops; and `try-catch` `-finally` statements. Functions can be declared with `fun` *name(args)* – where arguments are laid out identical to in Python – while classes can be declared with `class` *name(args)* – where arguments create primary constructors in a manner identical to in Kotlin.

As a brief aside, due to the frequent necessity for the creation of explicitly sized quantum registers for representation as binary states in quantum computation, Quff provides syntactic sugar for the creation of unsigned integers. $V$`u`$N$ produces an unsigned integer with value $V$ and size $N$, while $V$`u{e}` produces an unsigned integer with value $V$ and size equal to the expression `e`.

### 3.4 Quantum Semantics

The distinct difference regarding language constructs between Quff and classical languages with a similar syntax and classical semantics, is that the language's dynamic nature involves conversion to and from **all** types, including quantum types.

This section defines the quantum-related semantics of Quff, namely the denotational semantics of quantum variable creation, followed by the typing rules for quantum expression to showcase the propagation of quantum types through high-level expression. The evaluation of quantum expressions is performed in a lazy manner; upon execution of a quantum expression at run time, pertinent information of the corresponding quantum operation is stored in a quantum intermediate representation (shortened to IR throughout this paper). This IR is later optimised, decomposed and simulated when measurement/simulation is called for. More detail on this can be found in section 4.

In denotational semantics, semantics functions are used to represent the evaluation of certain types of statements, such as expressions with $\mathbb{E}$ or declarations with $\mathbb{D}$. Such functions are applied to sections of code within $[\![\ ]\!]$ and a program state $\sigma$, which results in some change of program state.

We describe the symbols used in Figure 2. At run time, the program state $\sigma$ contains a set **IR** which captures all quantum operations performed. For readability sake, we include the definition of an additional semantic function $\mathbb{Q}$ which wraps the result of an expression within a new `QReg` – effectively setting it to be of a quantum type (2). Furthermore, we use short-hand notation to represent multiple elements from the same set, such as $\vec{e}$ to denote $e_0, \ldots, e_n$. Multiple lines

| Syntactic Element | Symbol | Semantic Function |
|---|---|---|
| Identifier | $x$ | - |
| Declaration | $d$ | $\mathbb{D}$ |
| Expression | $e$ | $\mathbb{E}$ |
| Statement | $s$ | $\mathbb{S}$ |
| Function | - | $\mathbb{F}$ |

| Notation | Description |
|---|---|
| $\mathbf{IR}$ | Stateful Quantum IR |
| $\mathbf{IR} \longleftarrow \{\vec{\mathbf{op}}\}$ | Appending $\{\mathbf{op}_0, \ldots, \mathbf{op}_n\}$ to IR |
| $\mathbf{IR} \longrightarrow \mathbf{IR}_{\vec{e}}$ | Changing to a controlled state on $\vec{e}$ |
| $\mathbf{IR} \longrightarrow \mathbf{IR}^{\dagger}$ | Changing to an inverted state |

**Figure 2.** Notation detail

$$\mathbb{D}[\![\ \texttt{var}\ x = e\ ]\!]\sigma = \mathbb{D}[\![d]\!]\sigma = \sigma[x \mapsto \mathbb{E}[\![e]\!]\sigma] \tag{1}$$

$$\mathbb{Q}[\![e]\!]\sigma \equiv\ \mathbb{F}[\![\ \texttt{QReg}(v)\ ]\!]\sigma, \qquad \textit{where } v = \mathbb{E}[\![e]\!]\sigma \tag{2}$$

$$\mathbb{F}[\![\ \texttt{H}(x)\ ]\!]\sigma = \sigma[\mathbf{IR} \longleftarrow \{\texttt{gate}_x\ \texttt{H}\ \}] \tag{3}$$

$$\mathbb{D}_q[\![\ \texttt{var}\ x = e\ ]\!]\sigma_q = \sigma[x \mapsto \mathbb{Q}[\![e]\!]\sigma,\ \mathbf{IR} \longleftarrow \{\texttt{init}_x,\ \texttt{assign}_x\ e\ \}] \tag{4}$$

$$\mathbb{S}[\![\ \texttt{q:}\{\ \vec{d}\ ;\ \vec{s}\ \}]\!]\sigma\ = \mathbb{S}[\![\ \vec{s}\ ]\!]\sigma \circ \mathbb{D}_q[\![\ \vec{d}\ ]\!]\sigma_q \tag{5}$$

$$\mathbb{S}[\![\ \texttt{H:}\{\ \vec{d}\ ;\ \vec{s}\ \}\ ]\!]\sigma = \mathbb{F}[\![\ \texttt{H}(\ \vec{x}\ )\ ]\!]\sigma \circ \mathbb{S}[\![\ \vec{s}\ ]\!]\sigma \circ \mathbb{F}[\![\ \texttt{H}(\ \vec{x}\ )\ ]\!]\sigma \circ \mathbb{D}_q[\![\ \vec{d}\ ]\!]\sigma_q, \tag{6}$$
*where $\vec{x}$ represents all symbols used in $\vec{d}$.*

$$\mathbb{S}[\![\ \texttt{H:}(\vec{d}\ ;\ \vec{e})\{\ \vec{s}\ \}\ ]\!]\sigma = \mathbb{F}[\![\ \texttt{H}(\ \vec{x}\ )\ ]\!]\sigma \circ \mathbb{S}[\![\ \vec{s}\ ]\!]\sigma \circ \mathbb{F}[\![\ \texttt{H}(\ \vec{x}\ )\ ]\!]\sigma \circ \mathbb{Q}[\![\ \vec{e}\ ]\!]\sigma \circ \mathbb{D}_q[\![\ \vec{d}\ ]\!]\sigma_q, \tag{7}$$
*where $\vec{x}$ represents all symbols used in $\vec{d}$ and $\vec{e}$.*

**Figure 3.** Quantum keyword semantics.

of code are written with a semi-colon as a delimiter, while $\circ$ represents the composition of semantics, i.e. $\mathbb{S}[\![s_0; s_1]\!]\sigma = \mathbb{S}[\![s_1]\!]\sigma \circ \mathbb{S}[\![s_0]\!]\sigma$.

### 3.5 Quantum Keywords

As eluded to earlier, the language provides two quantum keywords – namely `q:` and `H:`. These keywords allow for quick creation of – and conversion to – quantum variables, either in a basis (`q:`) or superpositional (`H:`) quantum state. Figure 3 denotes the semantics of this explicit quantum declaration. When placed within a block preceeded by `q:` or `H:`, the use of `var` or `val` declares quantum variables. This is implemented by wrapping a classical value within a `QReg` type (2), and adding **`init`** and **`assign`** operations to the **IR**. This can be seen by the semantic function $\mathbb{D}_q$ (4) which uses a program state $\sigma_q$. Here, $\sigma_q$ only demonstrates that execution of a declaration is within the boundary of `q:` or `H:`, showcasing the contextual difference between equations (1) & (4).

Equations 5 & 6 detail the full usage of `q:` and `H:` to change how declarations are evaluated. Equation 5 can be seen as first creating quantum variables for each declaration within $\vec{d}$, and then executing all the statements $\vec{s}$ within the block given by {}. Equation 6 shows that `H:` does the same, however also puts these variables into a superpostitional state, by adding `H` gates before and after the execution of the statements $\vec{s}$. Finally, equation 7 shows the other manner of `H:` usage, which allows for the passing in of arguments to this keyword. These arguments may take the form of declarations for quantum variable creation, or expressions for conversion from classical to quantum variables. These quantum variables are put into a superpositional state in a similar manner to before, i.e. via `H` gates.

### 3.6 Quantum Expressions

The evaluation of an expression is governed by the type of the sub-expressions within; child expressions which are of a quantum type, cause the parenting expression to also be of a quantum type. In the following, we use typing rules to showcase this effect at run time. Here, the typing context $\Gamma$ represents the mapping of all expressions to either a classical set of types $C$ – such as `Int` or `Bool` – or a quantum set of types $Q$ – such as `QReg` or `Operation`.

This can be seen in Figure 4. Rule UNARY_E shows that when an expression $e$ has a type that belongs to the set of quantum types $Q$, then any unary operation it is used in also

$$\frac{\Gamma \vdash e \ : \ Q}{\Gamma \vdash !e \ : \ Q} \ \text{Unary_E} \qquad \frac{(\Gamma \vdash e_0 \ : \ Q) \ \vee \ (\Gamma \vdash e_1 \ : \ Q)}{\Gamma \vdash e_0 \ + \ e_1 \ : \ Q} \ \text{Binary_E}$$

$$\frac{\Gamma \vdash e_0 \ : \ Q \qquad \Gamma \vdash e_1 \ : \ Q}{\Gamma \vdash e_0 \ \&\& \ e_1 \ : \ Q} \ \text{L_And} \qquad \frac{\Gamma \vdash e_0 \ : \ Q \qquad \Gamma \vdash e_1 \ : \ Q}{\Gamma \vdash e_0 \ || \ e_1 \ : \ Q} \ \text{L_Or}$$

**Figure 4.** Quantum expression typing rules.

$$\frac{\dfrac{\Gamma \vdash e \ : \ Q}{\Gamma \vdash \texttt{if}(e)\{ \ s_0 \ \} \ \texttt{else} \ \{ \ s_1 \ \} \ : \ Q \qquad \Gamma \vdash s_0 \ : \ Q \qquad \Gamma \vdash s_1 \ : \ Q} \ \text{If_Type}}{\mathbb{S}[\![\texttt{if}(e)\{ \ s_0 \ \} \ \texttt{else} \ \{ \ s_1 \ \}]\!]\sigma = \sigma[\mathbf{IR}_{\ldots,!e} \rightarrow \mathbf{IR}_{\ldots}] \circ \mathbb{S}[\![s_1]\!]\sigma_q \circ \sigma[\mathbf{IR}_{\ldots,e} \rightarrow \mathbf{IR}_{\ldots,!e}] \circ \mathbb{S}[\![s_0]\!]\sigma_q \circ \sigma[\mathbf{IR}_{\ldots} \rightarrow \mathbf{IR}_{\ldots,e}]} \ \text{If_DSem}$$

$$\mathbb{F}[\![\texttt{control}(e)]\!]\sigma = \sigma[\mathbf{IR}_{\ldots} \rightarrow \mathbf{IR}_{\ldots,e}] \quad (8)$$

$$\mathbb{F}[\![\texttt{uncontrol}()]\!]\sigma = \sigma[\mathbf{IR}_{\ldots,e} \rightarrow \mathbf{IR}_{\ldots}] \quad (9)$$

$$\mathbb{F}[\![\texttt{invert}()]\!]\sigma = \sigma[\mathbf{IR} \rightarrow \mathbf{IR}^{\dagger}] \quad (10)$$

$$\mathbb{F}[\![\texttt{invertF}(f,e)]\!]\sigma = \sigma[\mathbf{IR}^{\dagger} \rightarrow \mathbf{IR}] \circ \mathbb{F}[\![f(e)]\!]\sigma \circ \sigma[\mathbf{IR} \rightarrow \mathbf{IR}^{\dagger}] \quad (11)$$

$$\mathbb{S}[\![\texttt{invert}() \ ; \ s_{q,0} \ ; \ s_{q,1} \ ; \ s_{q,2} \ ; \ \texttt{invert}()]\!]\sigma \equiv \mathbb{S}[\![s^{\dagger}_{q,2} \ ; \ s^{\dagger}_{q,1} \ ; \ s^{\dagger}_{q,0}]\!]\sigma \quad (12)$$

**Figure 5.** Quantum control flow and IR inverting.

has a type that belongs to the set of quantum types $Q$. This extends in the same manner to binary operations as can be seen in rule Binary_E, where if any sub-expression $e_0$ or $e_1$ has a type that belongs to the set of quantum types $Q$, then the overall expression $e_0 + e_1$ also has a type that belongs to the set of quantum types $Q$. Here, the operators ! and + represent any unary or binary operations, except for logical operators && and ||.

For logical operators consisting of classical and quantum sub-expressions, the system evaluates classical sub-expressions first. If all classical sub-expressions hold `true` for &&, or `false` for ||, then quantum sub-expressions must be evaluated. These are then concatenated together to produce a purely-quantum logical && or || expression – as is shown in rules L_And and L_Or.

### 3.7 Quantum Control & Inversion

Figure 5 details the typing rules and denotational semantics for quantum control using an `if` statement in Quff, as well as the denotational semantics of the 4 built-in functions for quantum IR manipulation at run time.

The typing rule and denotational semantics for an `if` statement using a quantum conditional expression $e$ are described together. This can be read as when the expression $e$ has a type which belongs to the set of quantum types $Q$, then an `if` statement it is used in, is also of a type in $Q$, along with both $s_0$ and $s_1$ (If_Type). Therefore, both $s_0$ and $s_1$ must be evaluated, where the statement(s) inside must now perform quantum operations, controlled and anti-controlled on $e$ (If_DSem).

Controlling states within the IR are implemented as a stack, where all the controlling states in this stack are used to control quantum operations. This can be seen in equations 8 & 9, which effectively push and pop controlling operations on/off this stack.

Inversion of the IR can be done using the functions `invert()` and `invertF`$(f, e)$, as shown in 10 & 11. When in an inverted state, the IR collects all successive operations in a separate set to normal. Later, when the IR is put back into a normal state via another call to `invert()`, all operations that have been collected in this separate set are inverted, and added to the IR's normal operation set in reverse order. The IR is then set back to a non-inverted state; `invert()` is involutive, in that it toggles the IR between a normal and inverted state. As a clarifying example, we provide equation 12. Here, the application of quantum operations $s_{q,0}$, $s_{q,1}$ & $s_{q,2}$ in between two `invert()` statements, is equivalent to just applying the inverse of the operations in reverse order, i.e. $s^{\dagger}_{q,2}$ ; $s^{\dagger}_{q,1}$ ; $s^{\dagger}_{q,0}$.

### 3.8 *qfree* Expressions

Expressions which do not strictly impose quantum effects are known as being *qfree*, as they do not introduce or destroy superposition. An example is the `X` gate. When applied to a

qubit in a computational basis state, such as $|0\rangle$ or $|1\rangle$, the application of this gate does not make the qubit unable to be represented classically. In Quff, such gates in this context do not impose quantum operation – `X(3u2)` simply equals the classical value `0u2`.

Most high-level operations are not *qfree*, as if a subexpression is of a quantum type, then it typically leads to a quantum operation. However, there is an exception to this: bit-shifting. When bit-shifting a quantum variable by a classical amount, one can simply insert or remove qubits in the relevant quantum variable.

### 3.9 Quantum Variables and Qubits

As shown in equation 2, quantum variables are implemented as a `QReg` type which wraps around its classical value. However, these `QReg` types are also implemented as a subclass of a `List` of `Qubit`s, and so individual `Qubit`s can be operated on as elements within.

### 3.10 Quantum Variable Pointers

It is of note that there can be two separate reasons for assigning one quantum variable to another; *weak* quantum copying (subsection 2.2) or pointing a quantum variable at another. By default, assignment of one quantum variable to another works as the former, but Quff allows the use of pointers on quantum variables to enable the latter.

### 3.11 Quantum Low-level Operation

As eluded to in earlier sections, typical quantum gates common to quantum circuits are available in Quff as simple built-in functions. When executed at run time, these functions correspond to adding the relevant gates to the IR as a **gate** operation (3). Gate functions may operate over individual `Qubit`s, whole `QReg`s, or classical values, where the latter converts the classical value to a temporary quantum variable (except for *qfree* gates). Quff also enables **phase** constructs, which applies a given phase rotation across the quantum system. When controlled, these constructs instead apply the given phase rotation to the controlling state.

### 3.12 Measurement

Measurement is used to simulate the current quantum program stored in the quantum IR, invoked at run time via the `measure`($x$) function. Subsections 4.6 – 4.8 detail the implementation of this process. All operations within the quantum IR which act on $x$, as well as quantum variables entangled with $x$, are simulated. The simulated quantum state of the IR is then collapsed, resulting in a now-classical state for each variable. These variables within the program state are then updated to the classical values, and the evaluation of Quff code resumes.

## 4 Compilation, Runtime & IR Generation

### 4.1 Pipeline

Figure 6 portrays the flow of compilation and execution of a file containing a Quff program. After parsing and transforming to Truffle nodes (subsection 4.2), the Truffle executor is used to execute such Truffle nodes. Here, quantum operations re-specialise to call methods within an internal IR generator, storing pertinent information regarding quantum computation (subsection 4.3). Upon the execution of a `measure()` or `toQASM()` function call, execution passes to the Decomposer, which transforms the current quantum IR into a low-level circuit representation (subsection 4.6), after which simulation (subsection 4.8) or OpenQASM 3.0 translation occurs (subsection 4.9). Following the simulation path, measured variables are updated within the Truffle execution frame, and execution of the Quff code resumes.

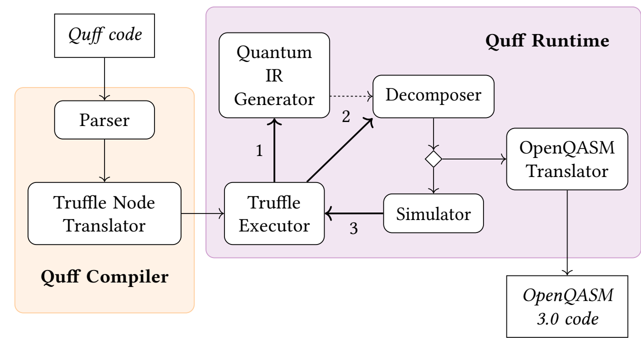



**Figure 6.** The Quff pipeline, showcasing how code flows through the system at compile time and run time. The edge labelled 1 indicates repeated internal calls to the IR Generator; the edge labelled 2 indicates a `measure()` or `toQASM()` call; and the edge labelled 3 indicates the passing back of measured values. The dashed ⇢ arrow represents passing of the IR to the Decomposer when required.

### 4.2 Parsing and Truffle Tree Compilation

Leveraging the dynamic structure of the language, the Quff compiler aims to produce abstract syntax tree (AST) representations of programs consisting solely of classical nodes, with quantum-classical data available inside of the corresponding scope. The result is a detailed AST which will be executed by the Truffle runtime system [55].

Leveraging this system, nodes re-specialise based on the data they execute over. This lends to a simple implementation for operating over classical data vs the generation of a quantum IR when operating on quantum data.

### 4.3 Quantum Intermediate Representation

The wrapping of classical values within a quantum type is implemented using two language objects: the `QReg` object for individual quantum variables, and the `Operation` object

for quantum operations across various quantum or classical variables.

During evaluation at run time, assigning an `Operation` to a variable corresponds to *(a)* potentially creating a new `QReg` object for the variable to be stored as, and *(b)* adding the `Operation` to the IR on said `QReg`.

The IR itself is made up of multiple sets of nodes. Each set corresponds to an ordered list of `Operation`s directly affecting a specific `QReg`, termed a `QReg`'s critical path/chain. Each node in this chain will represent the state of the `QReg` at a specific point in time. Nodes may link between chains to indicate usage of other quantum states, such as for controlled application. This is demonstrated in Figure 7, showcasing the critical path for each `QReg`, along with the control and usage dependencies for any relevant operation.

Pointers to the start and end of each set in this IR are stored for efficient access and manipulation of the IR, implemented using two Java HashMaps. Of important note in the structure of this IR, is the fact that specific ordering of operations on time-steps is not captured, only the dependencies for each operation. Therefore, it is ensured that resulting quantum computation optimises for parallel operation whenever possible, intrinsically reducing circuit depth.

This can be seen in Figure 7, which displays a short Quff program and the corresponding quantum IR created. Standard ⟶ arrows denote a quantum variable's critical path. Arrows with a circle •⟶/○⟶ indicate a controlled/anti-controlled operation. Dotted lines ---› indicate that an operation uses the value of another operation. Finally, the circles ● / ● represent the start and end `HashMaps` and corresponding pointers.

### 4.4 Feedback

The compiler and runtime system may produce basic messages to the user upon encountering quantum effects which cannot be achieved. One example of this is a quantum-controlled while loop; it is not possible to implement a termination condition when repeatedly controlling operations on a quantum system. Further feedback with respect to complex quantum debugging practices and quantum correctness tooling is not covered.

### 4.5 Automatic Uncomputation

Whenever a quantum variable goes out of scope at run time, a call is sent to the IR generator to try and uncompute it automatically. The algorithm for this – as seen in algorithm 1 – is similar in nature to that of `unqomp` [57], with the difference that it acts over high-level operations and quantum variables with distinct control and usage dependencies. This may fail in certain cases as highlighted by [57]. In this case, the relevant qubits are kept in the system and a warning message is produced. This does not affect program correctness, but may lead to inefficient qubit usage.

```
1 q: var one = 1u1, walk = 0u2
2 var coin = H(0u1)
3 if (coin) walk += one
4 else walk -= one
```

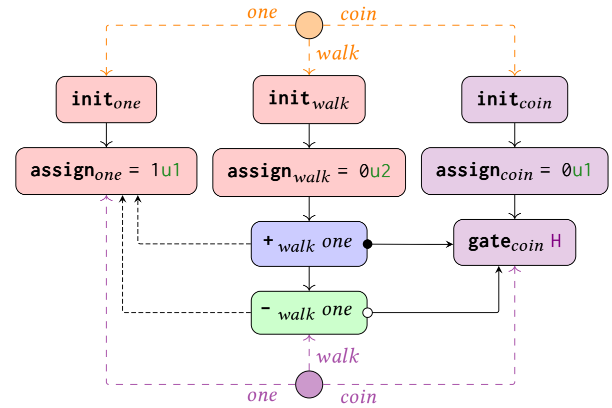



**Figure 7.** An example Quff program for one step of a quantum random walk program [33], with the corresponding quantum intermediate representation generated during run time. In this example, nodes in the IR are colour-coded based on the line of code which adds it to the IR.

### 4.6 Low-level Transformation

Upon the execution of a call to the built-in `measure()` or `toQASM()` function, the transformation process begins. A pass is applied over the generated quantum IR, gathering all nodes which interact with the quantum variable(s) to be measured (passed as arguments to the function), generating a sub-graph of the IR consisting solely of relevant operations to the measurement. From there, nodes are visited from start-to-finish, resolving dependencies for each time-step, and transforming into a corresponding low-level quantum circuit. This modular decomposition inductively ensures correctness is achieved, as long as each operation is individually transformed correctly.

The corresponding circuit for high-level operations is, by default, the application of the relevant high-level arithmetic, logical or bit-wise operation over signed or unsigned quantum integers, and quantum booleans as single qubits. Operations are decomposed to correct qubit-efficient circuits, as simulator complexity scales exponentially on qubit count. Arithmetic operations on both unsigned and signed integers are decomposed to the circuits described in [82], while unsigned integer division uses the circuit given in [73] for variable preservation. Logical operations correspond to controlled usage of applicable operations, potentially using temporary qubits to store results, where comparisons are performed using the circuits described in [82].

**Algorithm 1:** Algorithm for automatic uncomputation

```
1  Function uncompute(qvar):
2      curr ← last node on qvar
3      last ← curr
       // while there is a current node
4      while curr not null do
           // skip if it doesn't modify qvar
5          if op does not alter qvar then
6              curr ← node on qvar before curr
7              continue
           // check if uncomputable
8          if hasCycle(curr) then
9              return false
10         inv ← inverse operation of curr
11         make last point to inv
12         last ← inv
13         curr ← node on qvar before curr
14         if curr is an init node then
15             free ← create free node
16             make last point to free
17             store free in lasts map
18             return true // uncompute finished
19     return true // nothing to uncompute
   // recursively check usages of nodes for
      dependency cycles
20 Function hasCycle(curr):
21     usages ← all variables that are used by curr
22     return recur(curr, usages)
23 Function recur(curr, usages):
24     for c in nodes which use curr do
25         if c.qvar is in usages then
26             return recur(c, usages) // dependency
                  cycle
27     return false
```

Bit-wise operations are implemented with qubit array manipulation when coupled with classical values (i.e. not requiring quantum operation), or shifted application of quantum operators controlled on their quantum operand.

### 4.7 Final Decomposition

A final decomposition pass then occurs on the circuit, converting multi-controlled gates and operations down to the Clifford+T universal gate set [81]. This is done via the ZYZ-decomposition on single qubit gates [67], the optimised Quantum Shannon Decomposition method for multi-qubit operations [37], and the Gray code method described in [75] for multi-controlled operations.

### 4.8 Simulation

Following decomposition, the circuit is transferred to TornadoQSim [38] for parallel simulation. Following successful simulation, a `State` object is produced, containing pertinent details regarding the current quantum state of the variables measured. This state is then collapsed, updating all the relevant quantum variables with their classical values; ensuring these variables are used as classical going forward. Thus allowing effective, abstracted measurement-based hybrid quantum programming.

The section of the quantum IR which was simulated is then removed, and replaced with the `State` object. Thus recurring executions of that IR section result in simple reading of the relevant information from the object, instead of repeating time-consuming quantum emulation.

### 4.9 Translation to OpenQASM 3.0

While the previous flow has been concerning simulation of quantum computation generated at run time, the Quff system does provide simple translation to `OpenQASM 3.0` [11] – a universal quantum assembly language used by most quantum compilers – so that the user may further optimise or run their code on real-world quantum systems.

## 5 Quff by Example — Grover's algorithm

Grover's algorithm [24] is widely known for being one of the first algorithms for demonstrating the applicability of quantum computation in the real world. The algorithm can be reduced to finding the solution(s) to a binary problem, such as searching an unsorted database, requiring $O(\sqrt{n})$ time complexity as opposed to the classical $O(n)$. The description of this algorithm in Quff lends itself to a succinct portrayal of high-level oracle definition, quantum temporary variables and automatic uncomputation.

The program code and a subsection of the generated quantum IR is displayed in Figure 8. Execution starts on line 12 with the `main()` function, within which we declare a lambda function `f` representing the oracle function to solve.

The generation of the quantum computation starts after the invocation of the `grover` function, where on line 3 the first quantum variable is created as a quantum register over $n$ (5) qubits, initially in the state $|00000\rangle$, represented as a quantum unsigned integer. It is then put into an equal superposition, with both the **`init`** and **`gate`** `H` operations appended into the quantum IR.

The usage of the now-quantum variable `q` as a function argument on line 5 results in the lambda function being evaluated as a quantum expression, expanding to a set of nested quantum `Operation`s, with the final one being an

```
1  fun grover(f: Function, n: Int): UInt {
2    val iter = calculateIterations(n)
3    H:(var q = 0u{n}) {
4      for (i in iter){
5        if (f(q)) Phase(@pi)
6        invertF(f, q)
7        diffuse(q)
8      }
9      return measure(q)
10   }
11 }
12
13 fun main(){
14   val f = { x ->
15     x *= 3u
16     x += 5u
17     return x == 17u
18   }
19   print(grover(f, 5))
20 }
```

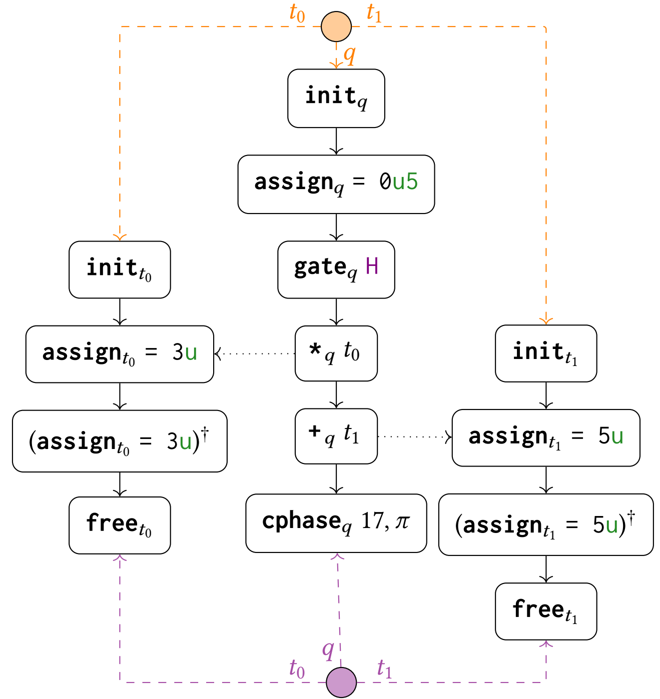



**Figure 8.** A Quff implementation for Grover's algorithm (left) searching for solutions to the equation $3x+5\ == 17,\ 0 \leq x < 2^5$, and a subsection of corresponding quantum IR (right) generated at run-time, representing the IR upto and including the execution of line 5. Note the functions `calculateIterations` and `diffuse` are omitted for brevity sake.

equality comparison with the state $|17\rangle$. All of these are added to the quantum IR in order. The condition $x == 17$ is then used to apply a controlled-phase operation to $q$ – applying a phase of $-1$ to the state $|17\rangle$. The arithmetic operations are then applied in inverse (but not the conditional statement on line 17 as it is never assigned to anything), followed by diffusion via. the `diffuse` function.

Note that temporary variables are used to represent $3u$ and $5u$ for the arithmetic operations, however the conditional equality of the quantum variable compared to $17u$ can be done with multiple controls and **cphase**.

## 6 Comparison of Quff

This section serves as comparison of quantum program definition when using the Quff language against two quantum development toolkits, namely the Qiskit python library [5], and the Silq language [8].

Qiskit was chosen due to it being the most commonly-used system for quantum programming, centered around API usage and circuit building. All quantum resources and effects are Python objects which must be explicitly defined, which rely on function calls to build up quantum circuits.

Silq was chosen as it is a high-level, type-safe, dedicated quantum language revolving around high-level expression and complex quantum annotation. Using a static type system where a `!` is used to denote classical types, quantum programs are created using expression and quantum keywords such as `qfree` and `lifted`. It is worth noting that Silq programs can not produce low-level quantum circuits, and relies on quantum simulation for compilation.

We feel these two languages give a broad scope over gate-based quantum programming for evaluation against the Quff language. All three produce quantum programs using a different modality; Qiskit's API usage & low-level gates, Silq's type-safety & quantum annotation, and Quff's dynamic typing & simplicity. Note that Qiskit may contain API functions which perform whole parts of quantum computation, such as the `Shor` function. However, due to the frequent nature of API-centered bugs in quantum code [6] – especially when high-level Qiskit API's are constantly changing – these are not used.

The sections of code written in each language details solely the quantum part of execution in the spirit of fair evaluation; for example, Qiskit typically requires 4 lines to cleanly send a program off for simulation, and 4 for the importing of necessary Qiskit libraries. Comparison occurs across 4 metrics, namely;

1. Lines of code;
2. Number of quantum high-level statements;

3. Number of quantum primitives commands;
4. Number of quantum-related annotations.

All metrics represent evaluations on readibility & the ability to identify and fix bugs within code – the more of both, the harder this can be, except for the number of quantum high-level statements which works in reverse. The last 3 metrics further represent complexity imposed by the language, leading to greater difficulty to learn, program and debug.

It is worth noting that the efficiency of compilation, circuit output & simulation is not evaluated, as it is not the main focal points of the paper. However, Quff does acheive similar circuit compile-time, circuit efficiency and simulation time when compared with Qiskit. With regards to Silq, Quff can be much faster to compile due to Silq's requirement for quantum simulation, while having slightly slowly simulation times. Furthermore, Silq cannot produce circuits as output.

All of the Quff code is provided in the text, while code for Silq and Qiskit can be seen in Appendix A.

### 6.1 $W_n$ State Creation

One of the most well-known three qubit states is the $W$ state, which is characterised by a superposition over three distinct states, each of which corresponding to a single qubit in the $|1\rangle$ state, while the other two are $|0\rangle$ – namely $W = \frac{1}{\sqrt{3}}(|100\rangle + |010\rangle + |001\rangle)$. Here, we give the reasoning of the creation of the state with respect to the Quff language, followed by the comparison of creating the state in Qiskit and Silq.

This state can then be generalised over $n$ qubits, resulting in the state

$$W_n = \frac{1}{\sqrt{n}}(|10\ldots0\rangle + |01\ldots0\rangle + \ldots + |00\ldots1\rangle)$$

From a computational state perspective, this can be seen as the bit-shifting of an unsigned integer of length $n$ initially in the state 1, where the amount to bit-shift by is then given by *multiple amounts at the same time* i.e. a combination of multiple computational states corresponding to $x \in \{0, 1, ..., n-1\}$. Note the direction of the shift does not matter – as the ordering of the linear composition of states is irrelevant due to the equal positive phase of each term – however we use left-shift for simplicity;

$$\begin{aligned} W_n &= \frac{1}{\sqrt{n}}\Big( |(0\ldots1) << 0\rangle + \ldots + |(0\ldots1) << (n-2)\rangle \\ &\qquad + |(0\ldots1) << (n-1)\rangle \Big) \\ &= \frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |(0\ldots1) << i\rangle \end{aligned}$$

Due to Quff's dynamic nature and quantum abstraction, this can be achieved in a single line (given $n$ is already declared within the program's scope) as shown in Figure 9.

```
1 var W_n = 1u{n} << H(0)
```

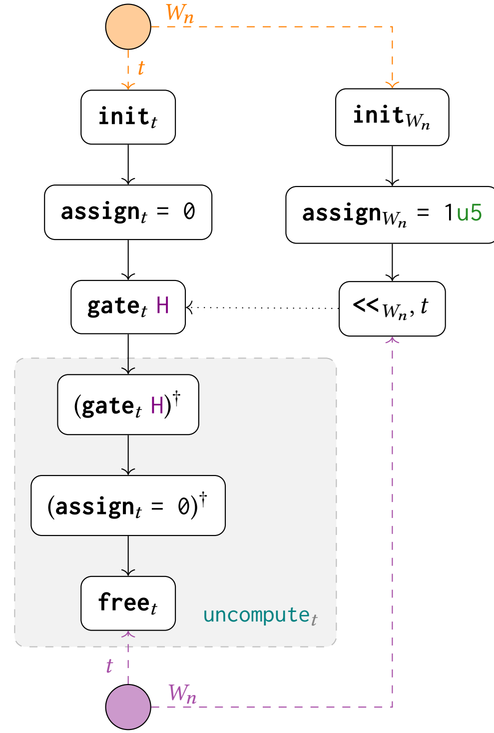



**Figure 9.** The Quff program excerpt for the creation of a $W_n$ state (top), and the corresponding quantum IR when $n = 5$ (bottom).

**Table 1.** Comparison of metrics for programs which generate a $W_5$ state. *Line count including whitespaces.

| | W State Creation | | | |
|---|---|---|---|---|
| | *Lines* | *Q. HL Stmnts* | *Q. Primitives* | *Q. Keywords* |
| Qiskit | 24 (26)* | 0 | 13 | 0 |
| Silq | 13 (15)* | 2 | 2 | 2 |
| **Quff** | **1** | **1** | **1** | **0** |

### 6.2 Shor's Algorithm

Shor's algorithm [68] is a well-known algorithm which uses the theory of period finding to perform prime factorisation of integers. It can do so in a time complexity polynomial to $O(\log N)$, as opposed to the most-efficient classical variant 'the general number field sieve', which works in sub-exponential time $O(e^{\log \log N})$ [51]. The quantum computation part of the algorithm uses controlled modular exponentiation and quantum phase estimation, alongside some classical pre- and post-processing. The theory behind the algorithm is out of the scope of this paper.

Note that in the evaluated Qiskit code for Shor's algorithm (subsubsection A.2.1), the program attempts to factorise 15, while the Quff and Silq code factorise over a generalised $N$. This is because the nature of SWAP gates used in the Qiskit implementation relies on knowledge of $N$ when writing the program – i.e. cannot be generalised.

```
1  fun shor(a, n, N){
2    H:(var qs = 0u{2*n-1}){
3      q: var b = 1u{n}
4      var i = 0
5      for (q in qs)
6        if (q) b = a**(2**i++) % N
7      invertF(qft, [qs])
8      return measure(qs)
9    }
10 }
```

**Figure 10.** The Quff program excerpt which performs the quantum computation involved in Shor's algorithm.

**Table 2.** Comparison of metrics for programs which implement the quantum computation involved in Shor's algorithm. * Line count including whitespaces.

| | **Shor's algorithm** | | | |
|---|---|---|---|---|
| | *Lines* | *Q. HL Stmnts* | *Q. Primitives* | *Q. Keywords* |
| Qiskit | 38 (42)* | 0 | 11 | 0 |
| Silq | 26 (31)* | 4 | 3 | 8 |
| **Quff** | **8** | **4** | **1** | **2** |

## 7 Related Work

**Operational Quantum Abstraction** High-level quantum abstraction is a relatively new strategy, with the usage of high-level classical control statements for quantum control available in the Silq language [8], while high-level operators are available in the isQ [25] & Qrisp languages [64], and the QHLS framework [9]. However, no language besides ours allows the combination of the two for complete high-level, imperative quantum expression in a manner similar to classical programs.

**High-level Quantum IR** To the best of our understanding, the Qunity language [76] is currently the only system which uses an IR of high-level operations, which is then decomposed to low-level circuits. However, the Quff quantum IR consists of arithmetic, logical and bit-wise operations akin to those used in classical programs.

**Quantum Keywords** Many quantum languages require the usage of various keywords, such as Silq's `qfree` and `const` for compile-time quantum safety [8], or the Q# `adjoint` for inverse expression [72]. These keywords allow for expressive program design while incurring additional development complexity. In contrast, Quff is the first which enables all quantum computation without necessitating keywords, while providing `q:` and `H:` for syntactic sugar.

**Quantum Typing** Nearly all quantum languages require the explicit stating of quantum types, such as Silq's `!` symbol [8]; Quipper's `QCData` type [22]; Qiskit's `QuantumRegister` object [5]; or isQ's `qbit` type [25]. Within our understanding, only Qunity [76] does not require the explicit stating of quantum types, where the context of data usage imposes quantum effects – however, Quff is the first to do so in an object-oriented, imperative language with a dynamic type-system.

**Hybrid Nature** The Quff language and runtime system is the only programming language which dynamically captures data as being of a quantum type, representing data which may only be represented as a quantum state. This dynamic nature inherently reduces the usage of quantum resources through lazy quantum evaluation.

## 8 Conclusions

In this era of early quantum software development, a focus on simplistic modalities for computation design is paramount for its early adoption. Throughout this paper we have demonstrated that the Quff language is suited for this purpose, as the first dynamically-typed, high-level quantum language with a syntax similar to that of current, highly-used classical programming languages – enabling of complete quantum program design without need for a single keyword. Our comparisons indicate that the novel quantum abstraction provided by Quff may decrease program length and quantum complexity compared to Qiskit and Silq.

The language leverages the idea of classical and quantum effects – in contrast to explicit quantum constructs. Being the first quantum programming language developed using GraalVM and the Truffle framework allows optimised run time with the novel dynamic type system, providing intuitive semantics for effective quantum software engineering. The Quff runtime system provides a powerful novel high-level quantum IR, leading to the generation of efficient circuits. It is the first to use lazy evaluation to optimize high-level, quantum computation, which may be simulated on a GPU accelerated quantum simulator, as well as conversion to quantum assembly code (QASM) to interact with quantum devices.

While the runtime system does not currently focus on optimisation of quantum programs, such as circuit and simulation efficiency, it leaves much room for this to be achieved in our future works.

# Appendix A Qiskit & Silq Code

## A.1 $W_n$ State Creation

```
1  def F_gate(qc,q,i,j,n,k) :
2    theta = np.arccos(np.sqrt(1/(n-k+1)))
3    qc.ry(-theta,q[j])
4    qc.cz(q[i],q[j])
5    qc.ry(theta,q[j])
6    qc.barrier(q[i])
7
8  def cxrv(qc,q,i,j) :
9    qc.h(q[i])
10   qc.h(q[j])
11   qc.cx(q[j],q[i])
12   qc.h(q[i])
13   qc.h(q[j])
14   qc.barrier(q[i],q[j])
15
16 q = QuantumRegister(n)
17 qc = QuantumCircuit(q)
18 qc.x(q[4])
19 F_gate(qc, q, 4, 3, 5, 1)
20 F_gate(qc, q, 3, 2, 5, 2)
21 F_gate(qc, q, 2, 1, 5, 3)
22 F_gate(qc, q, 1, 0, 5, 4)
23 qc.cx(q[3], q[4])
24 cxrv(qc, q, 2, 3)
25 qc.cx(q[1], q[2])
26 qc.cx(q[0], q[1])
```

**Listing 1.** The Qiskit program excerpt for the creation of a $W_5$ state [13]

```
1  i:=0:uint[n];
2  for j in [0..n){ i[j]:=H(i[j]); }
3
4  qs:=vector(2^n,0:\B);
5  qs[i]=X(qs[i]);
6
7  forget(i=\(qs:\B^(2^n))lifted{
8    i:=0:uint[n];
9    for j in [0..2^n){
10     if qs[j]{
11       i=j as uint[n];
12     }
13   }
14   return i;
15 }(qs));
```

**Listing 2.** The Silq program excerpt for the creation of a $W_n$ state, a total of 13 lines (not including whitespaces) [14].

## A.2 Shor's Algorithm

```
1  def init(qc, n, m):
2    qc.h(range(n))
3    qc.x(n+m-1)
4
5  def c_amod15(a, i):
6    if a not in [2,7,8,11,13]:
7      raise ValueError("'a' not valid.")
8    U = QuantumCircuit(4)
9    for iteration in range(i):
10     if a in [2,13]:
11       U.swap(0,1)
12       U.swap(1,2)
13       U.swap(2,3)
14     if a in [7,8]:
15       U.swap(2,3)
16       U.swap(1,2)
17       U.swap(0,1)
18     if a == 11:
19       U.swap(1,3)
20       U.swap(0,2)
21     if a in [7,11,13]:
22       for q in range(4):
23         U.x(q)
24   return U.to_gate().control()
25
26 def mod_exp(qc, n, m, a):
27   for i in range(n):
28     qc.append(c_amod15(a, 2**i), [i] + list(
       range(n, n+m)))
29
30 def inverse_qft(qc, qs):
31   qc.append(QFT(len(qs), do_swaps=False).
       inverse(), qs)
32
33 n = 4
34 m = 4
35 a = 7
36 qc = QuantumCircuit(n+m, n)
37 init(qc, n, m)
38 mod_exp(qc, n, m, a)
39 inverse_qft(qc, range(n))
40 qc.measure(range(n), range(n))
```

**Listing 3.** The Qiskit program excerpt which performs the quantum computation involved in Shor's algorithm.

```
def factorise[n:!\N](a:!\N, N:!uint[n]):!\R[]
    {
  got_factors := array(2, 0:!\R);

  while got_factors[0] == 0 && got_factors[1]
     == 0 {
    x := 0:uint[2*n-1];
    w := 0:uint[n];

    for i in [0..2*n] { x[i] := H(x[i]); }

    for  j in [0..2*n] {
      if x[j] { w = (a^(2^j)) % N; }
    }

    x := invQFT(x);
    got_factors = extract_factors(a as !\N, N
     as !\N, measure(x) as !\N);
    ...
  }
  return got_factors;
}

def invQFT[n:!\N](x:uint[n]):uint[n] {
  for i in [0..n) {
    for j in [n-i..n) {
      r := x[n-i-1];
      if r { x[j] := rotZ(-\pi/(2^(j-(n-i-1))
    ), x[j]); }
      forget(r = x[n-i-1]);
    }
    x[n-i-1] := H(x[n-i-1]);
  }
  return x;
}
```

**Listing 4.** The Silq program excerpt which performs the quantum computation involved in Shor's algorithm.

## References


[1] Muhammad Azeem Akbar, Arif Ali Khan, Sajjad Mahmood, and Saima Rafi. 2022. *Quantum Software Engineering: A New Genre of Computing*. Technical Report. https://doi.org/10.48550/arXiv.2211.13990 arXiv:2211.13990 [cs] type: article.

[2] Mahat Akhin and Mikhail et. al Belyaev. 2020. Kotlin language specification: Kotlin/Core. https://kotlinlang.org/spec/introduction.html Last accessed: 21/01/2024.

[3] Shaukat Ali and Tao Yue. 2020. Modeling Quantum programs: challenges, initial results, and research directions. In *Proceedings of the 1st ACM SIGSOFT International Workshop on Architectures and Paradigms for Engineering Quantum Software (APEQS 2020)*. Association for Computing Machinery, New York, NY, USA, 14–21. https://doi.org/10.1145/3412451.3428499

[4] T. Altenkirch and J. Grattage. 2005. A functional quantum programming language. In *20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05)*. 249–258. https://doi.org/10.1109/LICS.2005.1

[5] M. D. Sajid Anis, Héctor Abraham, AduOffei, Rochisha Agarwal, Gabriele Agliardi, Merav Aharoni, Ismail Yunus Akhalwaya, Gadi Aleksandrowicz, Thomas Alexander, Matthew Amy, Sashwat Anagolum, Eli Arbel, Abraham Asfaw, Anish Athalye, Artur Avkhadiev, Carlos Azaustre, Prathamesh Bhole, Abhik Banerjee, Santanu Banerjee, Will Bang, Aman Bansal, Panagiotis Barkoutsos, Ashish Barnawal, George Barron, George S. Barron, Luciano Bello, Yael Ben-Haim, M. Chandler Bennett, Daniel Bevenius, Dhruv Bhatnagar, Arjun Bhobe, Paolo Bianchini, Lev S. Bishop, Carsten Blank, Sorin Bolos, Soham Bopardikar, Samuel Bosch, Sebastian Brandhofer, Brandon, Sergey Bravyi, Nick Bronn, Fuller Bryce, David Bucher, Artemiy Burov, Fran Cabrera, Padraic Calpin, Lauren Capelluto, Jorge Carballo, Ginés Carrascal, Adam Carriker, Ivan Carvalho, Adrian Chen, Chun-Fu Chen, Edward Chen, Jielun Chen, Richard Chen, Franck Chevallier, Kartik Chinda, Rathish Cholarajan, Jerry M. Chow, Spencer Churchill, CisterMoke, Christian Claus, Christian Clauss, Caleb Clothier, Romilly Cocking, Ryan Cocuzzo, Jordan Connor, Filipe Correa, Abigail J. Cross, Andrew W. Cross, Simon Cross, Juan Cruz-Benito, Chris Culver, Antonio D. Córcoles-Gonzales, Navaneeth D, Sean Dague, Tareq El Dandachi, Animesh N. Dangwal, Jonathan Daniel, Marcus Daniels, Matthieu Dartiailh, Abdón Rodríguez Davila, Faisal Debouni, Anton Dekusar, Amol Deshmukh, Mohit Deshpande, Delton Ding, Jun Doi, Eli M. Dow, Eric Drechsler, Eugene Dumitrescu, Karel Dumon, Ivan Duran, Kareem El-Safty, Eric Eastman, Grant Eberle, Amir Ebrahimi, Pieter Eendebak, Daniel Egger, et al. 2021. Qiskit: An Open-source Framework for Quantum Computing. (2021). https://doi.org/10.5281/zenodo.2573505

[6] El Aoun Mohamed Raed, Li Heng, Foutse Khomh, and Openja Moses. 2021. Understanding Quantum Software Engineering Challenges: An Empirical Study on Stack Exchange Forums and GitHub Issues. (Sept. 2021). https://doi.org/10.26226/morressier.613b5418842293c031b5b5dd

[7] Francesco Bernardini, Abhijit Chakraborty, and Carlos Ordóñez. 2023. *Quantum computing with trapped ions: a beginner's guide.* Technical Report. https://doi.org/10.48550/arXiv.2303.16358 arXiv:2303.16358 [cond-mat, physics:physics, physics:quant-ph] type: article.

[8] Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. 2020. Silq: a high-level quantum language with safe uncomputation and intuitive semantics. ACM. https://doi.org/10.1145/3385412.3386007

[9] Chao, Christian Pilato, and Kanad Basu. 2024. QHLS: An HLS Framework to Convert High-Level Descriptions to Quantum Circuits. (2024). https://doi.org/10.1109/TCAD.2024.3391699

[10] MDN Contributors. 2023. JavaScript reference. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference Last accessed: 21/01/2024.

[11] Andrew Cross, Ali Javadi-Abhari, Thomas Alexander, Niel De Beaudrap, Lev S. Bishop, Steven Heidel, Colm A. Ryan, Prasahnt Sivarajah, John Smolin, Jay M. Gambetta, and Blake R. Johnson. 2022. OpenQASM 3: A Broader and Deeper Quantum Assembly Language. *ACM Transactions on Quantum Computing* 3, 3 (Sept. 2022), 12:1–12:50. https://doi.org/10.1145/3505636

[12] Manuel De Stefano, Fabiano Pecorelli, Dario Di Nucci, Fabio Palomba, and Andrea De Lucia. 2022. *Software Engineering for Quantum Programming: How Far Are We?* Technical Report. https://doi.org/10.48550/arXiv.2203.16969 arXiv:2203.16969 [cs] type: article.

[13] Pierre Decoodt. 2019. qiskit-community-tutorials/.../W State 1 - Multi-Qubit Systems. https://github.com/qiskit-community/qiskit-community-tutorials/blob/master/awards/teach_me_qiskit_2018/w_state/WState1-Multi-QubitSystems.ipynb Last accessed:

07/11/2023.

[14] ETH Zürich. [n. d.]. Silq - Examples - Generate W State. https://silq.ethz.ch/examples#/examples/2018_A4 Last accessed: 02/04/2024.

[15] Michael Felderer, Davide Taibi, Fabio Palomba, Michael Epping, Malte Lochau, and Benjamin Weder. 2023. Software Engineering Challenges for Quantum Computing: Report from the First Working Seminar on Quantum Software Engineering (WSQSE 22). *SIGSOFT Softw. Eng. Notes* 48, 2 (2023), 29–32. https://doi.org/10.1145/3587062.3587071

[16] Richard P. Feynman. 1986. Quantum mechanical computers. *Foundations of Physics* 16, 6 (June 1986), 507–531. https://doi.org/10.1007/BF01886518

[17] Python Software Foundation. 2024. The Python Language Reference. https://docs.python.org/3/reference/index.html Last accessed: 21/01/2024.

[18] Antonio García de la Barrera, Ignacio García-Rodríguez de Guzmán, Macario Polo, and Mario Piattini. 2023. Quantum software testing: State of the art. *Journal of Software: Evolution and Process* 35, 4 (2023), e2419. https://doi.org/10.1002/smr.2419

[19] Sunita Garhwal, Maryam Ghorani, and Amir Ahmad. 2021. Quantum Programming Language: A Systematic Review of Research Topic and Top Cited Languages. *Archives of Computational Methods in Engineering* 28, 2 (2021), 289–310. https://doi.org/10.1007/s11831-019-09372-6

[20] Craig Gidney and Martin Ekerå. 2021. How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits. *Quantum* 5 (2021), 433. https://doi.org/10.22331/q-2021-04-15-433

[21] Timothy David Goodrich. 2020. *Practical Graph Algorithms with Applications in Near-Term Quantum Computing*. Ph.D. https://www.proquest.com/docview/2407627362

[22] Alexander S. Green, Peter Lefanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. 2013. Quipper: A Scalable Quantum Programming Language. *ACM SIGPLAN Notices* 48, 6 (2013), 333–342. https://doi.org/10.1145/2499370.2462177

[23] Matthias Grimmer, Roland Schatz, Chris Seaton, Thomas Würthinger, Mikel Luján, and Hanspeter Mössenböck. 2018. Cross-Language Interoperability in a Multi-Language Runtime. *ACM Transactions on Programming Languages and Systems* 40, 2 (May 2018), 8:1–8:43. https://doi.org/10.1145/3201898

[24] Lov K. Grover. 1996. A fast quantum mechanical algorithm for database search. (1996). https://doi.org/10.48550/arXiv.quant-ph/9605043

[25] Jingzhe Guo, Huazhe Lou, Jintao Yu, Riling Li, Wang Fang, Junyi Liu, Peixun Long, Shenggang Ying, and Mingsheng Ying. 2023. isQ: An Integrated Software Stack for Quantum Programming. *IEEE Transactions on Quantum Engineering* 4 (2023), 1–16. https://doi.org/10.1109/TQE.2023.3275868

[26] Aram W. Harrow, Avinatan Hassidim, and Seth Lloyd. 2009. Quantum Algorithm for Linear Systems of Equations. *Physical Review Letters* 103, 15 (Oct. 2009), 150502. https://doi.org/10.1103/PhysRevLett.103.150502

[27] Bettina Heim, Mathias Soeken, Sarah Marshall, Chris Granade, Martin Roetteler, Alan Geller, Matthias Troyer, and Krysta Svore. 2020. Quantum programming languages. *Nature Reviews Physics* 2, 12 (Dec. 2020), 709–722. https://doi.org/10.1038/s42254-020-00245-7

[28] Dylan Herman, Cody Googin, Xiaoyuan Liu, Yue Sun, Alexey Galda, Ilya Safro, Marco Pistoia, and Yuri Alexeev. 2023. Quantum computing for finance. *Nature Reviews Physics* 5, 8 (Aug. 2023), 450–465. https://doi.org/10.1038/s42254-023-00603-1

[29] Essam H. Houssein, Zainab Abohashima, Mohamed Elhoseny, and Waleed M. Mohamed. 2022. Machine learning in the quantum realm: The state-of-the-art, challenges, and future vision. *Expert Systems with Applications* (2022), 116512. https://doi.org/10.1016/j.eswa.2022.116512

[30] Christian Humer, Christian Wimmer, Christian Wirth, Andreas Wöß, and Thomas Würthinger. 2014. A domain-specific language for building self-optimizing AST interpreters. *ACM SIGPLAN Notices* 50, 3 (Sept. 2014), 123–132. https://doi.org/10.1145/2775053.2658776

[31] Ali Javadi-Abhari. 2017. Towards a Scalable Software Stack for Resource Estimation and Optimization in General-Purpose Quantum Computers. (2017). https://dataspace.princeton.edu/handle/88435/dsp01jq085n573

[32] Philippe Jorrand. 2007. A Programmer's Survey of the Quantum Computing Paradigm. *International Journal of Business, Human and Social Sciences* 0.0, 8 (2007). https://doi.org/10.5281/zenodo.1331417

[33] J Kempe. 2003. Quantum random walks: An introductory overview. *Contemporary Physics* 44, 4 (July 2003), 307–327. https://doi.org/10.1080/00107151031000110776

[34] N. Khammassi, I. Ashraf, J, R. Nane, A, M, L. Lao, K. Bertels, and C. 2020. OpenQL : A Portable Quantum Programming Framework for Quantum Accelerators. *arXiv pre-print server* (2020). https://arxiv.org/abs/2005.13283

[35] P. Krantz, M. Kjaergaard, F. Yan, T. P. Orlando, S. Gustavsson, and W. D. Oliver. 2019. A quantum engineer's guide to superconducting qubits. *Applied Physics Reviews* 6, 2 (2019). https://doi.org/10.1063/1.5089550

[36] Michal Krelina. 2021. Quantum Technology for Military Applications. *EPJ Quantum Technology* (2021). https://doi.org/10.1140/epjqt/s40507-021-00113-y

[37] Anna M Krol, Aritra Sarkar, Imran Ashraf, Zaid Al-Ars, and Koen Bertels. 2022. Efficient decomposition of unitary matrices in quantum circuit compilers. *Applied Sciences* 12, 2 (2022), 759. https://doi.org/10.3390/app12020759

[38] Ales Kubicek, Athanasios Stratikopoulos, Juan Fumero, Nikos Foutris, and Christos Kotselidis. 2023. *TornadoQSim: An Open-source High-Performance and Modular Quantum Circuit Simulation Framework*. Technical Report. https://doi.org/10.48550/arXiv.2305.14398 arXiv:2305.14398 [quant-ph] type: article.

[39] Ryan LaRose. 2019. Overview and Comparison of Gate Level Quantum Software Platforms. *Quantum* 3 (March 2019), 130. https://doi.org/10.22331/q-2019-03-25-130

[40] Zhaoqi Leng, Pranav Mundada, Saeed Ghadimi, and Andrew Houck. 2019. Robust and efficient algorithms for high-dimensional black-box quantum optimization. (2019). https://doi.org/10.48550/ARXIV.1910.03591

[41] Frank Leymann and Johanna Barzen. 2020. The bitter truth about gate-based quantum algorithms in the NISQ era. *Quantum Science and Technology* 5, 4 (2020), 044007. https://doi.org/10.1088/2058-9565/abae7d

[42] Alessandro Luongo. 2023. *Quantum algorithms for data analysis*. 275 pages. https://quantumalgorithms.org/quantumalgorithms.pdf

[43] Alicia B. Magann, Matthew D. Grace, Herschel A. Rabitz, and Mohan Sarovar. 2021. Digital quantum simulation of molecular dynamics and control. *Physical Review Research* 3, 2 (2021). https://doi.org/10.1103/physrevresearch.3.023165

[44] Mario Piattini Manuel A. Serrano, Ricardo Pérez-Castillo. 2022. *Quantum Software Engineering* (1 ed.). Vol. 1. Springer Cham, Springer. 302 pages. https://doi.org/10.1007/978-3-031-05324-5

[45] John M. Martinis. 2015. Qubit metrology for building a fault-tolerant quantum computer. *npj Quantum Information* 1, 1 (2015), 15005. https://doi.org/10.1038/npjqi.2015.5

[46] Jarrod R. McClean, Jonathan Romero, Ryan Babbush, and Alán Aspuru-Guzik. 2016. The theory of variational hybrid quantum-classical algorithms. *New Journal of Physics* 18, 2 (Feb. 2016), 023023. https://doi.org/10.1088/1367-2630/18/2/023023

[47] Eñaut Mendiluze, Shaukat Ali, Paolo Arcaini, and Tao Yue. 2021. Muskit: A Mutation Analysis Tool for Quantum Software Testing. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1266–1270. https://doi.org/10.1109/ASE51524.2021.9678563 ISSN: 2643-1572.

[48] Andriy Miranskyy and Lei Zhang. 2019. On Testing Quantum Programs. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. 57–60.

https://doi.org/10.1109/ICSE-NIER.2019.00023

[49] Andriy Miranskyy, Lei Zhang, and Javad Doliskani. 2020. Is your quantum program bug-free?. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER '20)*. Association for Computing Machinery, New York, NY, USA, 29–32. https://doi.org/10.1145/3377816.3381731

[50] E. Moguel, J. Berrocal, J. García-Alonso, and J. M. Murillo. 2020. A Roadmap for Quantum Software Engineering: Applying the Lessons Learned from the Classics. https://www.semanticscholar.org/paper/A-Roadmap-for-Quantum-Software-Engineering%3A-the-the-Moguel-Berrocal/bca1ed2a4df94d6e54ecaad5c4971d84c57affdc

[51] Brian Antony Murphy. 1999. *Polynomial Selection for the Number Field Sieve Integer Factorisation Algorithm*. Ph. D. Dissertation. The Australian National University.

[52] Michael A. Nielsen and Isaac L. Chuang. 2010. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press. https://doi.org/10.1017/CBO9780511976667

[53] Oracle. 2013. Graal Project. https://openjdk.org/projects/graal/ Last accessed: 10/11/2023.

[54] Oracle. 2023. GraalVM. https://www.graalvm.org/ Last accessed: 10/11/2023.

[55] Oracle. 2023. Truffle Language Implementation Framework. https://www.graalvm.org/graalvm-as-a-platform/language-implementation-framework/ Last visited: 25/10/23.

[56] Luca Paolini, Mauro Piccolo, and Margherita Zorzi. 2019. QPCF: Higher-Order Languages and Quantum Circuits. *Journal of Automated Reasoning* 63, 4 (2019), 941–966. https://doi.org/10.1007/s10817-019-09518-y

[57] Anouk Paradis, Benjamin Bichsel, Samuel Steffen, and Martin Vechev. 2021. Unqomp: synthesizing uncomputation in Quantum circuits. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 222–236. https://doi.org/10.1145/3453483.3454040

[58] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J. Love, Alán Aspuru-Guzik, and Jeremy L. O'Brien. 2014. A variational eigenvalue solver on a photonic quantum processor. *Nature Communications* 5, 1 (July 2014), 4213. https://doi.org/10.1038/ncomms5213

[59] Mario Piattini, Guido Peterssen Nodarse, Ricardo Pérez Castillo, José Luis Hevia Oliver, Manuel A. Serrano, Guillermo Hernández, Ignacio García Rodríguez de Guzmán, Claudio Andrés Paradela, Macario Polo Usaola, Ezequiel Murina, Luis Jiménez, Juan Carlos Marqueño González, Ramsés Gallego, Jordi Tura, Frank Phillipson, Juán M. Murillo, A. Niño, and Moisés Rodríguez. 2020. The Talavera Manifesto for Quantum Software Engineering and Programming. (2020). https://core.ac.uk/display/478202536?source=2

[60] John Preskill. 2012. Quantum computing and the entanglement frontier. (2012). https://doi.org/10.48550/ARXIV.1203.5813

[61] John Preskill. 2018. Quantum Computing in the NISQ era and beyond. *Quantum* 2 (Aug. 2018), 79. https://doi.org/10.22331/q-2018-08-06-79

[62] Neilson Carlos Leite Ramalho, Higor Amario de Souza, and Marcos Lordello Chaim. 2024. *Testing and Debugging Quantum Programs: The Road to 2030*. Technical Report. https://doi.org/10.48550/arXiv.2405.09178 arXiv:2405.09178 [quant-ph] type: article.

[63] Naoto Sato and Ryota Katsube. 2023. Locating Buggy Segments in Quantum Program Debugging. *ArXiv* abs/2309.04266 (2023). https://api.semanticscholar.org/CorpusID:261660328

[64] Raphael Seidel, Nikolay Tcholtchev, Sebastian Bock, and Manfred Hauswirth. 2023. Uncomputation in the Qrisp High-Level Quantum Programming Framework. In *Reversible Computation*, Martin Kutrib and Uwe Meyer (Eds.). Springer Nature Switzerland, Cham, 150–165.

[65] Peter Selinger. 2004. Towards a quantum programming language. *Mathematical Structures in Computer Science* 14, 4 (Aug. 2004), 527–586. https://doi.org/10.1017/S0960129504004256

[66] Manuel A. Serrano, José A. Cruz-Lemus, Ricardo Perez-Castillo, and Mario Piattini. 2022. Quantum Software Components and Platforms: Overview and Quality Assessment. *ACM Comput. Surv.* 55, 8, Article 164 (dec 2022), 31 pages. https://doi.org/10.1145/3548679

[67] Vivek V. Shende, Igor L. Markov, and Stephen S. Bullock. 2004. Minimal universal two-qubit controlled-NOT-based circuits. *Physical Review A* 69, 6 (2004), 062321. https://doi.org/10.1103/PhysRevA.69.062321

[68] Peter W. Shor. 1997. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM J. Comput.* 26, 5 (Oct. 1997), 1484–1509. https://doi.org/10.1137/S0097539795293172

[69] Dmitry Solenov and Vladimir Privman. 2006. Evaluation of Decoherence for Quantum Computing Architectures: Qubit System Subject to Time-Dependant Control. *International Journal of Modern Physics B* 20, 11n13 (2006), 1476–1495. https://doi.org/10.1142/s0217979206034066

[70] Damian S. Steiger, Thomas Häner, and Matthias Troyer. 2018. ProjectQ: an open source software framework for quantum computing. *Quantum* 2 (2018), 49. https://doi.org/10.22331/q-2018-01-31-49

[71] Samuel Stein, Yufei Ding, Nathan Wiebe, Bo Peng, Karol Kowalski, Nathan Baker, James Ang, and Ang Li. 2021. EQC : Ensembled Quantum Computing for Variational Quantum Algorithms. (2021). https://doi.org/arxiv:2111.14940

[72] Krysta Svore, Martin Roetteler, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, and Andres Paz. 2018. Q#: Enabling scalable quantum computing and development with a high-level domain-specific language. *Proceedings of the Real World Domain Specific Languages Workshop 2018 on - RWDSL2018* (2018). https://doi.org/10.1145/3183895.3183901

[73] Himanshu Thapliyal, Edgard Muñoz-Coreas, T. S. S. Varun, and Travis S. Humble. 2018. Quantum Circuit Designs of Integer Division Optimizing T-count and T-depth. (2018). https://doi.org/10.48550/arXiv.1809.09732

[74] Jules Tilly, Hongxiang Chen, Shuxiang Cao, Dario Picozzi, Kanav Setia, Ying Li, Edward Grant, Leonard Wossnig, Ivan Rungger, George H. Booth, and Jonathan Tennyson. 2022. The Variational Quantum Eigensolver: A review of methods and best practices. *Physics Reports* 986 (Nov. 2022), 1–128. https://doi.org/10.1016/j.physrep.2022.08.003

[75] Juha J. Vartiainen, Mikko Möttönen, and Martti M. Salomaa. 2004. Efficient Decomposition of Quantum Gates. *Physical Review Letters* 92, 17 (April 2004), 177902. https://doi.org/10.1103/PhysRevLett.92.177902

[76] Finn Voichick, Liyi Li, Robert Rand, and Michael Hicks. 2023. Qunity: A Unified Language for Quantum and Classical Computing. *Proc. ACM Program. Lang.* 7, POPL, Article 32 (jan 2023), 31 pages. https://doi.org/10.1145/3571225

[77] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to rule them all. ACM Press. https://doi.org/10.1145/2509578.2509581

[78] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-optimizing AST interpreters. *ACM SIGPLAN Notices* 48, 2 (Oct. 2012), 73–82. https://doi.org/10.1145/2480360.2384587

[79] Pengzhan Zhao, Jianjun Zhao, and Lei Ma. 2021. Identifying Bug Patterns in Quantum Programs. In *2021 IEEE/ACM 2nd International Workshop on Quantum Software Engineering (Q-SE)*. 16–21. https://doi.org/10.1109/Q-SE52541.2021.00011

[80] Li Zhou, Gilles Barthe, Pierre-Yves Strub, Junyi Liu, and Mingsheng Ying. 2023. CoqQ: Foundational Verification of Quantum Programs. *Proc. ACM Program. Lang.* 7, POPL, Article 29 (jan 2023), 33 pages. https://doi.org/10.1145/3571222

[81] Xinlan Zhou, Debbie W. Leung, and Isaac L. Chuang. 2000. Methodology for quantum logic gate construction. *Phys. Rev. A* 62 (Oct 2000), 052316. Issue 5. https://doi.org/10.1103/PhysRevA.62.052316

[82] Engin Şahin. 2020. Quantum arithmetic operations based on quantum fourier transform on signed integers. *International Journal of Quantum Information* 18, 06 (9 2020), 2050035. https://doi.org/10.1142/s0219749920500355

Received 2024-05-25; accepted 2024-06-24

# Towards Realistic Results for Instrumentation-Based Profilers for JIT-Compiled Systems

Humphrey Burchell
University of Kent
Canterbury, United Kingdom
h.burchell@kent.ac.uk

Octave Larose
University of Kent
Canterbury, United Kingdom
o.larose@kent.ac.uk

Stefan Marr
University of Kent
Canterbury, United Kingdom
s.marr@kent.ac.uk

## Abstract

Profilers are crucial tools for identifying and improving application performance. However, for language implementations with just-in-time (JIT) compilation, e.g., for Java and JavaScript, instrumentation-based profilers can have significant overheads and report unrealistic results caused by the instrumentation.

In this paper, we examine state-of-the-art instrumentation-based profilers for Java to determine the realism of their results. We assess their overhead, the effect on compilation time, and the generated bytecode. We found that the profiler with the lowest overhead increased run time by 82×. Additionally, we investigate the *realism* of results by testing a profiler's ability to detect whether inlining is enabled, which is an important compiler optimization. Our results document that instrumentation can alter program behavior so that performance observations are unrealistic, i.e., they do not reflect the performance of the uninstrumented program.

As a solution, we sketch late-compiler-phase-based instrumentation for just-in-time compilers, which gives us the precision of instrumentation-based profiling with an overhead that is multiple magnitudes lower than that of standard instrumentation-based profilers, with a median overhead of 23.3% (min. 1.4%, max. 464%). By inserting probes late in the compilation process, we avoid interfering with compiler optimizations, which yields more realistic results.

***CCS Concepts:*** • **Software and its engineering → Just-in-time compilers**.

**ACM Reference Format:**
Humphrey Burchell, Octave Larose, and Stefan Marr. 2024. Towards Realistic Results for Instrumentation-Based Profilers for JIT-Compiled Systems. In *Proceedings of the 21st ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '24), September 19, 2024, Vienna, Austria.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3679007.3685058

This work is licensed under a Creative Commons Attribution 4.0 International License.

*MPLR '24, September 19, 2024, Vienna, Austria*
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1118-3/24/09
https://doi.org/10.1145/3679007.3685058

## 1 Introduction

Profilers are the go-to tool for developers to identify the program part that takes up most time and may benefit from optimization. Instrumentation-based profilers insert probes into the program, which then collect information about the program's behavior. In contrast, sampling periodically interrupts the program to collect information, e.g., records the program stack, to derive a probabilistic picture of the program's behavior. Both approaches are widely used for ahead-of-time- as well as just-in-time-compiled language implementations [8, 15, 19, 20].

Unfortunately, both sample- and instrumentation-based profiling of just-in-time-compiled programs affect program execution, reducing the accuracy of profiling results [3, 10, 14, 21], i.e., results may not represent the true performance behavior. Instrumenting code at the source or bytecode level changes how it is optimized [10]. Sampling suffers from safepoint bias, which means profilers do not sample all program parts with equal probability. Safepoint bias can also lead to misinterpreted samples, since the profiler only has a partial understanding of how the compiler altered a program's structure to achieve better performance [3, 14].

The one benefit of instrumentation is its high precision. Results may not be accurate, but probes count or measure reliably. Thus, we want to better understand instrumentation-based profilers on the Java Virtual Machine (JVM). To this end, we measure their overhead, which is typically increasing run time by one or two orders of magnitude. Furthermore, we assess the impact of instrumentation on compilation, and find that they can not detect whether inlining is enabled or disabled, which means their results are unrealistic.

As a way forward, we propose a new approach to instrumentation-based profiling on top of JIT compilers that improves the accuracy of profiles compared to source- and bytecode-level instrumentation by only instrumenting the code late in the compilation process.

Inspired by Basso et al. [2], we insert our instrumentation with a compiler phase of the Graal JIT compiler. This avoids changing which methods are selected for compilation, which methods are inlined, and it minimizes the impact from how the compiler optimizes the code. This is similar to instrumenting binaries of ahead-of-time-compiled programs [18], but with a lower engineering effort and thus, we believe, a suitable way forward for just-in-time compilers.

## 2 Background

This section introduces profiling, our terminology, the Graal compiler, and the challenges profilers have with inlining.

### 2.1 Profilers and Profiling Techniques

Profiling allows developers to observe a program's execution. A profiler may record, e.g., a program's CPU, GPU, or memory utilization. Profilers help identify performance issues, e.g., by pinpointing sections of code that consume the most CPU time, and thus, they can guide optimization efforts.

Instrumentation-based profilers insert probes into a program to record the information. For example, a probe at the beginning of each method can count how many times a method is called. This finds frequently called methods and can enable developers to identify underlying performance problems. Probes can be inserted at the source, bytecode, or native code level. We will evaluate JProfiler,[1] VisualVM,[2] and YourKit[3] as state-of-the-art instrumentation-based profilers.

CPU sampling interrupts the program to gather a snapshot of the current state of both hardware and software. This includes recording the call stack, instruction pointer, memory usage, and thread state, which are used to construct a profile. The interrupts occur at regular intervals, but could be random [14]. Safepoints [1] are crucial for garbage collection and to ensure that the VM is in a state where the stack can be correctly read and the program counter can be used to indentify the currently executing method. Thus, samples are typically interpreted based on the closest safepoints. However, this can lead to inaccurate profiles [14].

### 2.2 Accuracy, Precision, and Realism

Accuracy and precision are often used interchangably. However, in our work we use two distinct meanings.

*Accuracy* measures how close the reported profiler results are to what happens during executions without a profiler, i.e., how close the results are to the ground truth, which we unfortunately cannot determine directly. *Precision* refers to how close measurements are to each other between runs. Thus, it measures how consistent a profiler gives the same, but not necessarily correct answer.

As *realism* we understand a weaker notion of accuracy, which measures how close the reported profilers results are to an execution with a profiler that does not affect run-time optimizations. This allows for the general impact and bias introduced by using a profiler, but is meant to allow us to assess how instrumentation influences optimization heuristics. For instance, we would consider a profiler unrealistic when it significantly changes the effectiveness of specific optimizations, because the normal execution would not be subject to this change, and would show different performance properties.

[1] https://www.ej-technologies.com/products/jprofiler/overview.html
[2] https://visualvm.github.io/
[3] https://www.yourkit.com/

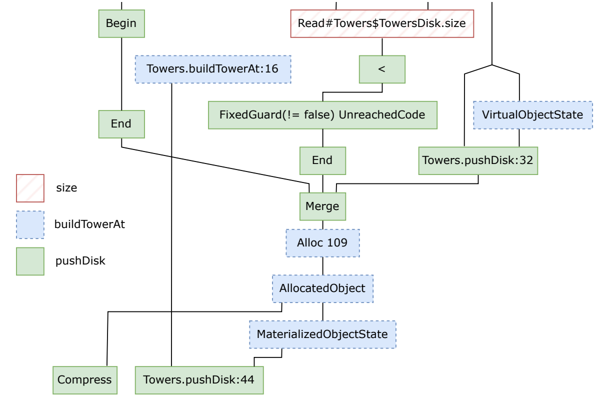



**Figure 1.** A GraalIR graph showing that nodes from inlined methods can end up being intermixed in a compilation unit.

### 2.3 Graal Compiler

The Graal compiler is a just-in-time (JIT) compiler implemented in Java. It compiles Java bytecode at run time to machine code. For this, it uses GraalIR, a graph-based *sea of nodes* [4] intermediate representation (IR) with explicit control flow edges [5]. It enables optimizations such as dead code elimination, loop transformation, and inlining by adding, transforming, or removing graph nodes. Each optimization is typically implemented in its own compiler phase.

### 2.4 Profilers and Inlining

Inlining replaces a method call with the body of the called method. This is a vital optimization, because it enables optimizations on the combination of caller and callee. Graal does inlining at the GraalIR level. After inlining the nodes from a callee, optimizations such as loop peeling can move and duplicate nodes and nodes from different methods can end up mixed together. Figure 1 illustrates this for the Tower benchmark [13] with nodes from the `buildTowerAt`, `size`, and `pushDisk` methods, each in a different color, without clear method boundaries between them. This makes it impossible to simply instrument the beginning and end of an inlined method, since they no longer exist.

Inlining thus complicates determining where time is spent. A sampler needs to know which method the currently executed instruction belongs to. For instrumentation, it depends on when probes are inserted. If they are inserted at the source or bytecode level, probes may be duplicated with the rest of the code, for instance during loop peeling, increase the overhead, and likely prevents optimizations, e.g., inlining [10].

If probes are added after inlining, then it may require many probes to isolate the different parts, and correctly attribute the execution time. This would be needed in our example in Figure 1 to accurately distinguish the methods. A simple profiler may only instrument the root method and attribute the time of the whole compilation unit to this method. However, a major part of the time may be spent on inlined code.

# 3 State of the Art in Instrumentation on JIT-Compiling JVMs

We will now analyze instrumentation-based profilers by examining their overhead, assessing their impact on the compilation process, and evaluating the realism of their results.

## 3.1 Experimental Setup

We ran our experiments with Graal on top of the HotSpot JVM in OpenJDK 21.0.2.[4] By using Graal's *libgraal* variant, we ensure that Graal itself is ahead-of-time compiled, which ensures the best possible compilation times from the start. All benchmarks were run on a machine with an AMD Ryzen 5 36000 6-core processor, which uses the Zen 2 architecture, 32 GB DDR4 RAM, and Rocky Linux 9.4 with a Linux kernel version 5.14.0. We chose to use the Are We Fast Yet benchmarks [13], because they are well-understood and deterministic. The suite includes 5 macro-benchmarks and 9 micro-benchmarks. We configured the benchmarks so that a single iteration takes about 100ms and we run each benchmark for 300 iterations. In this configuration, the benchmarks quickly reach a stable state and it is a good trade-off between overall run time and the number of measurements collected. All profilers profile immediately from the benchmark start. The benchmarks are executed with ReBench [12], which adapts the system's settings to minimize interference and noise.

## 3.2 Assessing Run-time Overhead

To assess the overhead instrumentation-based profilers introduce, we measure for each profiler its default settings using full instrumentation, i.e., without excluding any packages. This means for example that Java's standard library is instrumented, too. As a consequence of full instrumentation, we ran the benchmarks for only 10 iterations, because the high overhead made running the benchmarks for longer impractical. We verified that the relevant JIT compilation still occurs within the first iteration of the benchmark run.

Unfortunately, VisualVM does not seem to be scriptable and we could not execute our benchmarks automatically. This made it impractical run all experiments. Though, we ran the DeltaBlue benchmark, which is roughly in line with the other profilers, and we report results for it where relevant.

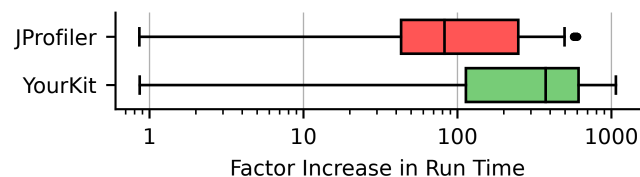



**Figure 2.** Overhead of instrumentation-based profilers for the Are We Fast Yet benchmarks. The overhead is expressed as a factor over the uninstrumented run time.

[4]We used a Graal from April 2024: https://github.com/oracle/graal/commit/249d3e4abd2f357461c5ceb682791e22b2c8a92f

Figure 2 reports the run-time overhead over the uninstrumented execution of all benchmarks. While fully instrumenting a program is expected to result in high overhead, the extent of this overhead can be substantial enough to render the use of a instrumentation-based profiler impractical. For VisualVM, which is not in the figure, profiling DeltaBlue increases the run time by 485×. This overhead is similar to the median overhead we found for YourKit and JProfiler.

## 3.3 Assessing Impact on Compilation

To understand why these profilers incur such high overhead, we assess the impact of instrumentation on the amount of code generated, and how it affects inlining. We extracted these details from Graal's compilation log, which is enabled with `-Djdk.graal.PrintCompilation=true`.

***Impact on Code Size.*** For each profiler, we collect the time spent in compilation, the total amount of bytecode, the size of the generated native code, and the memory allocations that occurred during compilation for each of our benchmarks.

The overhead we saw in Figure 2 is likely due to the added instrumentation itself. The increased bytecode and native code size seen in Figure 3 suggests that the probes cause the additional code and that the compiler is unable to optimize the instrumented code effectively.

When YourKit is attached, the added instrumentation causes the native code size to increase by a median of 341%,

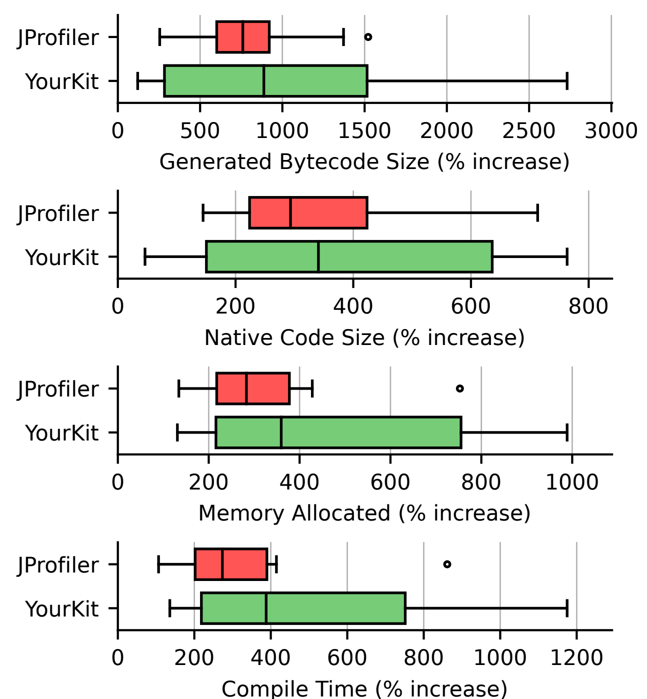



**Figure 3.** Increase of bytecodes, native code size, memory allocation, and compile time for attached instrumentation-based profilers on the Are We Fast Yet benchmarks.

which contributes to the overhead of 374×. The instrumentation likely also changes inlining and optimization decisions, further contributing to the slowdown. For example, instrumentation can cause some methods to become too large to be inlined, thus affecting the overall performance [10]. The increased code size and the resulting changes to optimization decision suggests that any results obtained from such profilers are likely less realistic than for other types of profilers.

***Impact of Instrumentation on Inlining.*** We have measured how inlining statistics change when instrumentation profilers are attached. The data for Figure 4 was collected from Graal's inlining log. When running the benchmarks without a profiler, the number of inlined methods is much lower, indicating that the instrumentation requires many more instrumentation-related methods to be inlined, which explains the earlier seen increase in native code size.

These numbers also suggest that the instrumentation is likely to dominate the execution of the benchmarks. For instance, small methods will end up consisting of more instrumentation code than application behavior. At the same time, the compiler is not able to remove the instrumentation, because it is indistinguishable from normal application code. As a result, the behavior of an instrumented program likely correlates with method activation counts. This is a strong indication that the profiled behavior does likely not resemble the normal program behavior, making profiling results "unrealistic," since optimizations do not give the same benefit anymore, but often change the performance behavior of a program significantly.

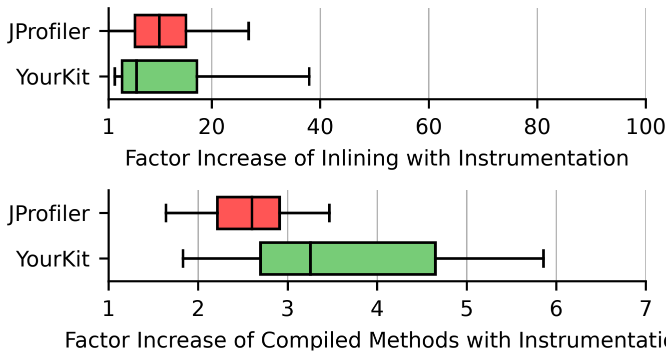



**Figure 4.** The increase in compiled methods and inlining caused by attached instrumentation profilers for AWFY benchmarks. The increase is expressed as a factor compared to the results of uninstrumented executions.

To understand better how much instrumentation changes the benchmark behavior, we ran the DeltaBlue benchmark with each instrumentation profiler, once normally, and once with inlining disabled.[5] We would expect a drastic change in behavior between inlining enabled and disabled.

[5]Inlining is disabled with the Graal compiler flag `-Dgraal.Inline=false`

**Table 1.** Profiling results for Async and JProfiler, with and without inlining enabled.

| Profiler | Inline | Method | % |
|---|---|---|---|
| Async | yes | deltablue.Plan | 23.3 |
| | | Vector.forEach | 17.7 |
| | | ScaleConstraint.execute | 4.9 |
| | | Vector.append | 3.7 |
| | | ScaleConstraint.recalculate | 3.5 |
| | no | EqualityConstraint.execute | 19.9 |
| | | ScaleConstraint.execute | 8.2 |
| | | DMH.newInvokeSpecial | 4.5 |
| | | Plan$$Lambda.apply | 3.7 |
| | | Variable.getValue | 3.6 |
| JProfiler | yes | EqualityConstraint.execute | 14.0 |
| | | Plan.lambda$execute$0 | 10.0 |
| | | Vector.forEach | 9.0 |
| | | Variable.getValue | 6.0 |
| | | ScaleConstraint.execute | 5.0 |
| | no | EqualityConstraint.execute | 14.5 |
| | | Plan.lambda$execute$0 | 10.0 |
| | | Vector.forEach | 8.0 |
| | | Variable.getValue | 6.0 |
| | | ScaleConstraint.execute | 5.0 |

We ran this experiment also with the Async-profiler, which uses sampling instead of instrumentation. We assume that sampling provides results closer to the ground truth, since it does not alter program behavior as much as instrumentation.

Table 1 shows the results for Async and JProfiler. The full results are in the appendix in **??**. These tables show that the profiles with and without inlining are close to identical for the instrumentation-based profilers. For Async, the sampling profiler, this is however not the case. Here enabling inlining drastically changes the profile as we would expect.

JProfiler reports the methods `Plan.lambda$execute$0`, `Vector.forEach`, and `EqualityConstraint.execute` as the ones taking most time, with and without inlining. Without inlining, Async reports `EqualityConstraint.execute`, `ScaleConstraint.execute`, and `newInvokeSpecial` from the JVM's method handle system as most important. Though with inlining, it reports `deltablue.Plan`, `Vector.forEach` and `ScaleConstraint.execute`, which suggests that inlining and subsequent optimizations change the observable behavior significantly.

For VisualVM and YourKit, inlining has also no major effect on the profiles. To us, this means that the instrumentation prevents us from seeing the impact of inlining, which itself enables many subsequent optimizations.

With this, we conclude that the probes used in state-of-the-art instrumentation approaches change the application behavior to such a degree, that the run-time behavior becomes unrealistic.

### 3.4 Instrumentation Bias towards Activation Count

When instrumentation alters an application's behavior, to such a degree that profiles strongly correlate with activation counts, they may no longer provide actionable guidance. Simply optimizing the most activated method may not be feasible or provide the desired performance gains, because the compiler may have already realized these benefits.

Figure 5 illustrates a worst-case scenario where a profile based on activation counts may misdirect optimization efforts. In this example. the `execute()` methods of `ActionA` and `ActionB` could be identified as most called. However, optimizing them is likely fruitless. A realistic profile would likely direct attention to the part of the program that dominates run time after optimizations, e.g., the bubble sort.

```
1 class ActionA { int id; void execute() {} }
2 class ActionB { int id; void execute() {} }
3 var actions = getMixOfManyActions();
4 bubbleSortById(actions);
5 framework.execute(actions);
```

**Figure 5.** Worst-case scenario for instrumentation profilers. Highest activation counts may misguide optimization efforts.

## 4 Improving Realism with Late-compiler-phase-based Instrumentation

In this section, we sketch late-compiler-phase instrumentation to improve realism of profiles and account for inlining.

### 4.1 Late-compiler-phase Instrumentation

Late-compiler-phase instrumentation inserts probes into compilation units in the latest practical JIT compiler phase to avoid interfering with optimizations. When we insert probes, most optimizations are already applied, which minimizes the observer effect and run-time overhead.

At the high-level, the compilation process remains unchanged, too. The JVM uses its normal heuristics to select a method for JIT compilation and the Graal compiler optimizes it with its many phases. Our implementation adds two phases to the process. The first is placed late in the highest tier, where high-level information is still available, which we use to collect details about the compiled method and methods that have been inlined. We also prepare the resolution of a memory address for our second phase. Though, our first phase does not insert any instrumentation nodes, which avoids interfering with optimizations.

Our second phase is added as late as possible to the low tier and adds our instrumentation nodes. These nodes record the CPU cycles at the start and each exit from the compilation unit. It also instruments calls into non-inlined methods. Further nodes are inserted to compute the CPU cycles taken directly by this compilation unit, and to add the result to the unit's entry in a global array for all compilation units. For this, we use the previously prepared memory address, which minimizes the run-time computations. The array is processed right before the JVM shuts down, to compute and output the overall profile.

Since we instrument code in the JIT compiler, only methods that are compiled will collect profiling information. Methods that are interpreted only thus will not be profiled. However, for many use cases, the methods relevant for performance are invoked often and therefore get JIT compiled.

### 4.2 Attributing Cycles to Inlined Methods

As discussed in Section 2.4, inlining is a major challenge for profilers when it comes to correctly attributing where a program spends its time, because inlining and subsequent compiler optimizations may cause an inlined method to be arbitrarily intermixed with parts from other methods.

To attribute time to inlined methods after all optimizations, we estimate the cycle cost for the remaining elements. We know for each GraalIR node from which method it originates. Based on branch probabilities and loop counts collected at run time, we then estimate which fraction of the overall compilation unit comes from a specific method. This allows us to identify which methods make up the hottest compilation units and avoids the overhead of instrumenting each remaining part of an inlined method separately.

### 4.3 Evaluation

To evaluate our late-compiler-phase-based instrumentation, we compare it with sampling- and the other instrumentation-based profilers. The setup is the same as in Section 3.1, which gives both instrumentation-based profilers and samplers enough time to profile the program in a stable state. However, some minor engineering issues prevented us from using libgraal for our implementation, which we call Bubo. Thus, all experiments with Bubo use *jargraal*, i.e., the Java version of Graal that is subject to JIT compilation itself. We believe this has no major impact on our results. In the worst case, it disadvantages Bubo compared to other profilers.

***Comparison with Classic Instrumentation.*** As shown in Figure 2, the median overhead for the instrumentation profilers is in the range of 82× to 374× over all benchmarks. This means at the median, JProfiler causes programs to take 82× more time compared to their uninstrumented version.

In contrast, Figure 6 shows that Bubo has a median overhead of only 23.3% (min. 1.4%, max. 464%), and having a generally lower impact on the execution of a program.

***Comparison with Sampling.*** Sampling profilers are expected to have a lower overhead than instrumentation-based profilers, since their overhead is proportional to the sampling frequency, instead of incurring a constant overhead for every instrumented method. Figure 6 shows that Bubo has a higher

median overhead with 23.3% (min. 1.4%, max. 464%) than the sampling profilers. Bubo's median overhead of 23.3% is also higher than the 75th percentile of the sampling profilers, as indicated by the right edge of the boxes. Nonetheless, Bubo's overhead is much closer to that of sampling-based profilers than that of instrumentation-based ones.

We believe these first results show that the approach could make instrumentation-based profiling more practical and realistic.

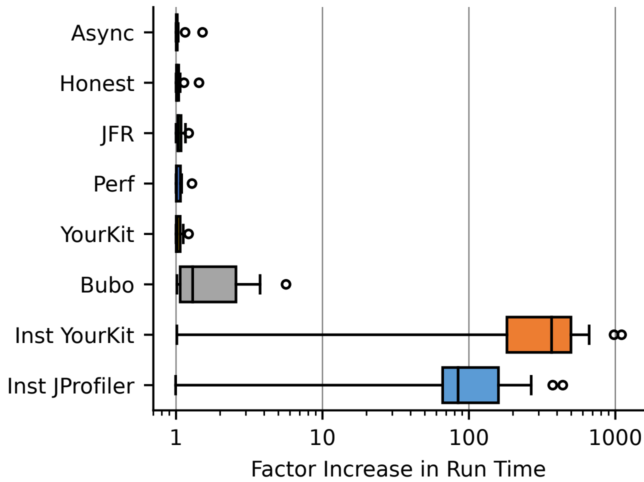



**Figure 6.** Run-time overhead for each profiler expressed as factor over the uninstrumented run time. Async has the lowest median overhead with 1% (min. 0.1%, max. 52%). Bubo's median overhead is 23.3% (min. 1.4%, max. 464%).

## 5 Related Work

The most relevant work is on instrumentation at the compiler level and binary rewriting.

Most notably, Zheng et al. [21] and Basso et al. [2] inspired our late-compiler-phase instrumentation. Zheng et al. [21] illustrate the impact of JIT compiler optimizations on, for instance, object allocations and method invocations, demonstrating the need to work alongside the JIT compiler to understand which of these operations are still present after optimization. Both [2, 21] focused on specific compiler optimizations and compiler events to better understand the compiler and performance issues with it. We on the other hand focus on profiling applications.

Much earlier work such as gprof [7], also instrumented programs as part of ahead-of-time (AOT) compilation. Today's compilers such as GCC and LLVM also support it, e.g., to enable profile-guided optimizations [16].

However, it seems more common today for application profiling to instrument binaries after compilation to avoid interfering with optimizations. Dynamic binary instrumentation can be used to make instrumentation very targeted for use in production, e.g., by sampling programs using instrumentation at configurable frequencies set by the user [11]. Others optimize instrumentation by combining multiple probes into one to reduce the overhead without losing information [9] or by using self-modifying instrumentation [17].

## 6 Conclusion

In this work, we show that state-of-the-art instrumentation-based profilers for the JVM have high overhead and report unrealistic results. We found that the lowest median overhead is 82× across the Are We Fast Yet benchmarks. The overhead can be explained with the cost of instrumentation, which is visible in the increased bytecode size, native code size, and amount of inlining. We further argue that the reported profiles are unrealistic, because they do not change when inlining is turned off, which indicates that the instrumentation negates most compiler optimizations.

To overcome these issues, we proposed late-compiler-phase-based instrumentation. It minimizes interference with compiler optimizations and as a result delivers more realistic profiles than other instrumentation-based profilers.

In our prototype implementation, it reduces the median profiler overhead on the Are We Fast Yet benchmarks to 23.3% (min. 1.4%, max. 464%), which is more similar to sampling-based profilers. Furthermore, we attribute the cycles for a compilation unit to the methods fragments that remain in the compilation unit after optimization to account for inlining. However, our prototype does not support multithreading.

***Future Work.*** The main benefit of instrumentation-based profiling over sampling is its precision, i.e., that it gives consistent results (see Section 2.2). However, it's not generally possible to assess the accuracy of profilers and samplers often disagree with eachother [3, 14]. Thus, an important open question is how to better approximate the ground truth profile for any given program. One could possibly use hardware simulators to determine the ground truth and thereby assess the accuracy of profilers [6].

One could also consider combining sampling and late-compiler-phase instrumentation to reduce the overhead, and gain precision for specific parts of a profile. A hybrid solution would also allow profiling of executions in the interpreter.

Other future work is more engineering focused. At this point, Bubo instruments all methods, but perhaps one would want to select manually which methods to instrument as in classic instrumentation-based profilers. Adding support for multithreaded application and ensuring the profile data is collected correctly would be needed to make Bubo work with most JVM applications.

## Acknowledgments

This work was supported by a grant (EP/V007165/1) and studentship (2619006) from EPRSC as well as a Royal Society Industry Fellowship (INF\R1\211001). The authors would also like to thank Matteo Basso and the GraalVM community for their advice and discussions.

# A Appendix

**Table 2.** A complete version of Table 1. Comparison of methods percentages across different profilers for the DeltaBlue benchmark with and without inlining.

| Profiler | Inlining | Method | Percentage |
|---|---|---|---|
| **Async** | yes | deltablue.Plan | 23 |
| | | Vector.forEach | 17 |
| | | ScaleConstraint.execute | 4 |
| | | Vector.append | 3 |
| | | ScaleConstraint.recalculate | 3 |
| | no | EqualityConstraint.execute | 19 |
| | | ScaleConstraint.execute | 8 |
| | | invoke.DirectMethodHandle$Holder.newInvokeSpecial | 4 |
| | | Plan$$Lambda.0x00007fcf5800d6c0.apply | 3 |
| | | Variable.getValue | 3 |
| **JProfiler** | yes | EqualityConstraint.execute | 14 |
| | | Plan.lambda$execute$0 | 10 |
| | | Vector.forEach | 9 |
| | | Variable.getValue | 6 |
| | | ScaleConstraint.execute | 5 |
| | no | EqualityConstraint.execute | 14 |
| | | Plan.lambda$execute$0 | 10 |
| | | Vector.forEach | 8 |
| | | Variable.getValue | 6 |
| | | ScaleConstraint.execute | 5 |
| **VisualVM** | yes | Plan.lambda$execute$0 | 14 |
| | | Vector.forEach | 12 |
| | | Variable.getValue | 6 |
| | | Planner.addPropagate | 4 |
| | | AbstractConstraint.satisfy | 2 |
| | no | Plan.lambda$execute$0 | 24 |
| | | Vector.forEach | 12 |
| | | Variable.getValue | 6 |
| | | Planner.addPropagate | 4 |
| | | AbstractConstraint.satisfy | 2 |
| **YourKit** | yes | deltablue.Plan.lambda$execute$0 | 24 |
| | | som.Vector.forEach | 17 |
| | | deltablue.EqualityConstraint.execute | 10 |
| | | deltablue.Planner.addPropagate | 3 |
| | | deltablue.AbstractConstraint.satisfy | 2 |
| | no | deltablue.Plan.lambda$execute$0 | 36 |
| | | som.Vector.forEach | 17 |
| | | deltablue.EqualityConstraint.execute | 10 |
| | | deltablue.Planner.addPropagate | 4 |
| | | deltablue.AbstractConstraint.satisfy | 2 |

## References


[1] Ole Agesen. 1998. *GC Points in a Threaded Environment.* Technical Report SMLI TR-98-70. Sun Microsystems.

[2] Matteo Basso, Aleksandar Prokopec, Andrea Rosà, and Walter Binder. 2023. Optimization-Aware Compiler-Level Event Profiling. *ACM Trans. Program. Lang. Syst.* 45, 2, Article 10 (jun 2023), 50 pages. https://doi.org/10.1145/3591473

[3] Humphrey Burchell, Octave Larose, Sophie Kaleba, and Stefan Marr. 2023. Don't Trust Your Profiler: An Empirical Study on the Precision and Accuracy of Java Profilers. In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR 2023)*. ACM, 100–113. https://doi.org/10.1145/3617651.3622985

[4] Cliff Click and Michael Paleczny. 1995. A simple graph-based intermediate representation. In *IR '95: Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations* (San Francisco, California, United States). ACM, 35–49. https://doi.org/10.1145/202529.202534

[5] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. 2013. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages (VMIL '13)*. ACM, 1–10. https://doi.org/10.1145/2542142.2542143

[6] Björn Gottschall, Lieven Eeckhout, and Magnus Jahre. 2021. TIP: Time-Proportional Instruction Profiling. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21)*. ACM, 15–27. https://doi.org/10.1145/3466752.3480058

[7] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. 1982. gprof: a Call Graph Execution Profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction* (Boston, Massachusetts, USA) *(SIGPLAN '82)*. ACM, 120–126. https://doi.org/10.1145/800230.806987

[8] Berkin Ilbeyi and C. Batten. 2016. JIT-assisted fast-forward embedding and instrumentation to enable fast, accurate, and agile simulation. *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2016), 284–295. https://doi.org/10.1109/ISPASS.2016.7482103

[9] Naveen Kumar, Bruce R. Childers, and Mary Lou Soffa. 2005. Low overhead program monitoring and profiling. *SIGSOFT Softw. Eng. Notes* 31, 1 (sep 2005), 28–34. https://doi.org/10.1145/1108768.1108801

[10] Elena Machkasova, Kevin Arhelger, and Fernando Trinciante. 2009. The observer effect of profiling on dynamic Java optimizations. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. ACM, New York, NY, USA, 757–758. https://doi.org/10.1145/1639950.1640000

[11] Scott Mahlke, Tipp Moseley, Richard Hank, Derek Bruening, and Hyoun Kyu Cho. 2013. Instant profiling: Instrumentation sampling for profiling datacenter applications. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (CGO '13)*. IEEE Computer Society, USA, 1–10. https://doi.org/10.1109/CGO.2013.6494982

[12] Stefan Marr. 2023. *ReBench: Execute and Document Benchmarks Reproducibly*. https://doi.org/10.5281/zenodo.8219119 Version 1.2.0.

[13] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. 2016. Cross-Language Compiler Benchmarking—Are We Fast Yet?. In *Proceedings of the 12th Symposium on Dynamic Languages* (Amsterdam, Netherlands) *(DLS'16)*. ACM, 120–131. https://doi.org/10.1145/2989225.2989232

[14] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2010. Evaluating the Accuracy of Java Profilers. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10)*. ACM, 187–197. https://doi.org/10.1145/1806596.1806618

[15] Marek Olszewski, Keir Mierle, Adam Czajkowski, and Angela Demke Brown. 2007. JIT Instrumentation - A Novel Approach To Dynamically Instrument Operating Systems. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (Lisbon, Portugal) *(EuroSys '07)*. ACM, 3–16. https://doi.org/10.1145/1272996.1273000

[16] Adam Preuss. 2010. *Implementation of Path Profiling in the Low-Level Virtual-Machine (LLVM) Compiler Infrastructure.* Technical Report TRID-ID TR10-05. 1–16 pages. https://doi.org/10.7939/R3GF0MX64

[17] Amitabha Roy, Steven Hand, and Tim Harris. 2011. Hybrid Binary Rewriting for Memory Access Instrumentation. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments* (Newport Beach, California, USA) *(VEE '11)*. ACM, 227–238. https://doi.org/10.1145/1952682.1952711

[18] Amitabh Srivastava and Alan Eustace. 1994. ATOM: A System for Building Customized Program Analysis Tools. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation* (Orlando, Florida, USA) *(PLDI '94)*. ACM, 196–205. https://doi.org/10.1145/178243.178260

[19] Maja Vukasovic and Aleksandar Prokopec. 2023. Exploiting Partially Context-sensitive Profiles to Improve Performance of Hot Code. *ACM Transactions on Programming Languages and Systems* 45 (2023), 1–64. https://doi.org/10.1145/3612937

[20] April W. Wade, P. Kulkarni, and Michael R. Jantz. 2017. AOT vs. JIT: impact of profile data on code quality. *Proceedings of the 18th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (2017). https://doi.org/10.1145/3078633.3081037

[21] Yudi Zheng, Lubomír Bulej, and Walter Binder. 2015. Accurate Profiling in the Presence of Dynamic Compilation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'15)*. ACM, 433–450. https://doi.org/10.1145/2814270.2814281


Received 2024-05-25; accepted 2024-06-24

# Toward Declarative Auditing of Java Software for Graceful Exception Handling

Leo St. Amour
Virginia Tech
Blacksburg, USA
lstamour@vt.edu

Eli Tilevich
Virginia Tech
Blacksburg, USA
tilevich@cs.vt.edu

## Abstract

Despite their language-integrated design, Java exceptions can be difficult to use effectively. Although Java exceptions are syntactically straightforward, negligent practices often result in code logic that is not only inelegant but also unsafe. This paper explores the challenge of auditing Java software to enhance the effectiveness and safety of its exception logic. We revisit common anti-patterns associated with Java exception usage and argue that, for auditing, their detection requires a more nuanced approach than mere identification. Specifically, we investigate whether reporting such anti-patterns can be prioritized for subsequent examination. We prototype our approach as HÄNDEL, in which anti-patterns and their priority, or weight, are expressed declaratively using probabilistic logic programming. Evaluation with representative open-source code bases suggests HÄNDEL's promise in detecting, reporting, and ranking the anti-patterns, thus helping streamline Java software auditing to ensure the safety and quality of exception-handling logic.

***CCS Concepts:*** **• Software and its engineering** → Automated static analysis; **Software safety**; **Error handling and recovery**; **• Computing methodologies → Probabilistic reasoning**; *Logic programming and answer set programming*.

***Keywords:*** Software Auditing, Static Program Analysis, Probabilistic Reasoning, Exceptions, Java, Logic Programming

**ACM Reference Format:**
Leo St. Amour and Eli Tilevich. 2024. Toward Declarative Auditing of Java Software for Graceful Exception Handling. In *Proceedings of the 21st ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '24), September 19, 2024, Vienna, Austria.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3679007.3685057

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*MPLR '24, September 19, 2024, Vienna, Austria*
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1118-3/24/09
https://doi.org/10.1145/3679007.3685057

## 1 Introduction

Java was not the first mainstream programming language with a built-in exception-handling mechanism. Dating back to the 1960s [35], several prior languages, including Ada [28] and C++ [26], supported handling runtime errors and exceptional conditions in a structured manner. Drawing inspiration from these languages, the design of Java has reconsidered several aspects of exceptions, including a standardized exception hierarchy, checked exceptions, and strict exception type checking [24]. After nearly 30 years, Java has experienced enormous success and now rules the world of enterprise software with billions of lines of legacy code. Unfortunately, analyzing legacy Java code reveals that exceptions have become a double-edged sword. Although it promotes the intended structured handling of exceptional conditions, it also allows undisciplined and ill-conceived programming practices [5, 6, 12, 17]. Unless following a strict set of coding principles, when it comes to exceptions, Java programmers are prone to exhibit common poor coding practices—often referred to as *anti-patterns* [9, 19].

Before a code base can be included in high-stakes environments, such as government or critical infrastructure systems, it must be audited for adherence to safety standards. Software auditing verifies and validates that a code base complies with the necessary standards and meets all its baseline requirements [10]. Unfortunately, auditing large code bases is notoriously time-consuming and tedious, requiring significant resources and expertise [32]. Consequently, auditing can greatly benefit from automated tools and processes [16].

Exception handling needs auditing as it fulfills the critical role of addressing exceptional runtime conditions. Certain features of Java exceptions make code susceptible to anti-patterns. Anders Hejlsberg, the lead C# architect, points out in an interview that "checked exceptions become such an irritation that people completely circumvent the feature… checked exceptions have actually degraded the quality of the system in the large" [48]. Indeed, Java programmers often circumvent the necessity to handle checked exceptions by creating empty catch clauses or catching a generic superclass exception type. Extensive research has codified such Java exception anti-patterns and studied their prevalence in legacy code bases [5, 6, 12, 38, 44, 53].

This paper focuses on the challenge of auditing Java software to ensure the effectiveness and safety of exception

```
 1 public void myCreateBucket(String name) {
 2   S3Client s3 = newS3Client();
 3   try {
 4     CreateBucketRequest req = newCBRequest(name);
 5     s3.createBucket(req);
 6     // throws BucketAlreadyExistsException,
 7     //        BucketAlreadyOwnedByYouException,
 8     //        AwsServiceException,
 9     //        SdkClientException, and S3Exception
10       ...
11   } catch (SdkException e) {
12     ... // handle exception appropriately
13   }
14   s3.close();
15 }
```

**Figure 1.** Code snippet for facilitating the exposition and definitions of Java exception anti-patterns; adapted from [2].

handling. Auditing tools that report many false positives cause information overload, particularly for large commercial code bases. With unlimited resources, an auditor could examine each reported case to confirm its validity. However, this practice would be infeasible in realistic settings. When it comes to auditing, detecting exception anti-patterns can benefit from a more nuanced treatment than the current methods that report their findings using boolean logic.

To address the challenges of boolean reporting, we introduce a novel approach that leverages probabilistic reasoning to detect and weigh possible anti-patterns. The weighted results can be used to guide further manual inspection. We prototype our approach as Händel, which concisely expresses anti-patterns in ProbLog, a probabilistic dialect of Prolog. This design enables declarative specifications that can be easily examined and tweaked. Despite the interpretive nature of ProbLog execution, Händel shows promising performance and memory consumption characteristics. An evaluation with representative Java benchmark applications demonstrates the potential of probabilistic reasoning for reporting suspected exception anti-patterns as well as Händel's ability to point out the likelihood of their presence. This paper makes the following contributions:

- We introduce a novel approach for reporting exception anti-patterns based on probabilistic reasoning to guide subsequent auditing efforts.
- We describe our prototype implementation, Händel, an extensible analysis framework that leverages probabilistic logic programming to identify potential exception anti-patterns; in Händel, analysis results are weighed via highly configurable ProbLog predicates.
- We assess our approach's suitability for auditing by applying Händel to a set of representative Java benchmark applications.

## 2 Exception-handling Anti-patterns

Previous works have established a Java exception-handling anti-pattern taxonomy [5]. This section revisits common anti-patterns defined in this taxonomy. Consider the Java method `myCreateBucket` depicted in Figure 1. This code snippet uses the Amazon AWS Java SDK [3]. Specifically, it creates a new bucket by using the API to interact with the Amazon S3 service. The API `createBucket` method throws numerous exceptions, as stated in the listing as comments. All these exceptions are sub-classes of the base class `SdkException`. This inheritance relationship makes it possible to use the base class exception in the `catch` clause for handling all possible exceptions.

Whether this code snippet matches an anti-pattern cannot be determined definitively. In some contexts, using the base class is appropriate for the desired level of exception handling. In others, it may be necessary to handle each of the potentially thrown exception sub-classes specially. In particular, from an auditing standpoint, we might need to be able to specify the degree to which a code base matches an anti-pattern. The reported degrees would then prioritize subsequent manual auditing. We next revisit common exception anti-patterns and argue that boolean reporting logic might be unnecessarily rigid for software auditing.

### 2.1 Anti-pattern 1: Catch Generic / Over-Catch

One of the most common Java exception anti-patterns is *Catch Generic* or *Over-Catch* (AP1), in which a handler catches an exception type that is a supertype of the thrown exceptions [38]. This type relationship is called subsumption [1]. This anti-pattern branches into two distinct types: (1) "catch generic", a program catching a high-level, generic exception type (i.e., Throwable, Exception, Error, or RuntimeError); (2) subsumption, or "over-catch" [53], a handler catching multiple different lower-level exceptions [5]. In both cases, the anti-pattern reflects that the caught exception type is a super-class of the types of exceptions thrown. As a result, the exception handler is less likely to appropriately account for the nuances of each possible sub-class of exception. For example, in our motivating example, we might want to be able to distinguish between `BucketAlreadyExistsException` and `BuckedAlreadyOwnedByYouException`, as we want to ensure that these particular exceptional conditions are handled specially. At the same time, it may be acceptable for the remaining exceptions to be handled generically.

### 2.2 Anti-pattern 2: Throws Generic / Over-Throws

Another Java exception anti-pattern is *Throws Generic* (AP2), in which a method's "throws" statement propagates a generic exception type [5]. Similar to AP1, this anti-pattern derives from using a super-class exception type. Because the specific thrown exception types are lost, any handlers that catch the thrown exceptions become incapable of specializing

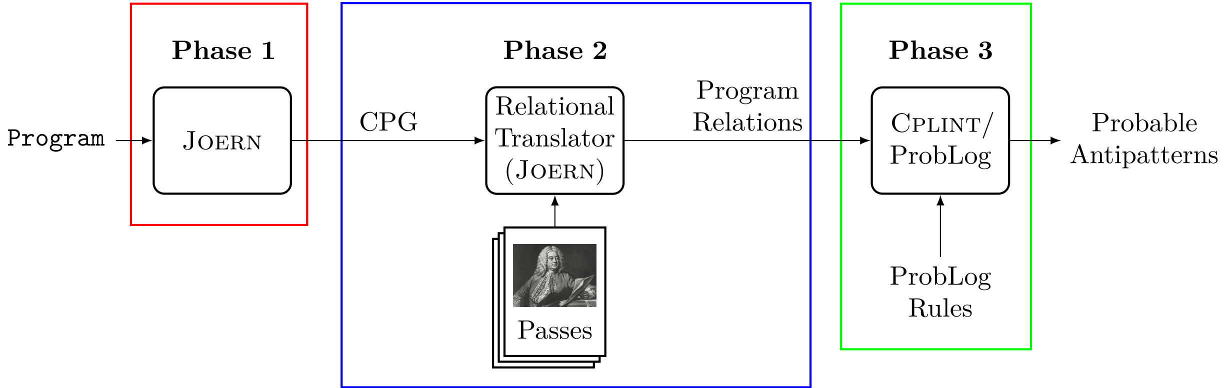



**Figure 2.** Händel system overview and data-flow diagram

their handling. Inspired by the over-catch anti-pattern, we propose a more encompassing *Over-Throws* anti-pattern in which the type specified in a throws statement subsumes the types of exceptions thrown. For example, the control flow of `createBucket` could pass through a method that propagates `SdkException`, which may or may not be problematic depending on the context.

### 2.3 Anti-pattern 3: Unhandled Exceptions

Yet another anti-pattern we consider is *Unhandled Exception* (AP3), in which a handler fails to catch all reachable exceptions [5, 44]. This anti-pattern describes the practice of implementing an exception handler that reaches but does not catch a thrown unchecked exception. By definition, the Java semantics do not require unchecked exceptions to be handled. However, within certain contexts, it may be necessary to handle such runtime exceptions gracefully. For example, assume the `createBucket` method in our motivating example additionally threw an `IllegalArgumentException`. If this exception is not handled, the program will terminate prematurely. Software auditors might want to ensure that specific unchecked exceptions are caught so that the software does not fail within a high-stakes environment.

## 3 Händel's Design and Implementation

Auditing contexts may differ greatly, depending on the software domain, deployment environment, and security/privacy restrictions. In light of these observations, our design must exhibit high degrees of transparency and configurability. The key insight of our design lies in employing probabilistic logic programming to specify our analyses. Transparency is achieved through comprehensible logical rules that specify anti-pattern detection. Configurability is achieved through special or custom logic predicates.

We prototype our approach as Händel, which draws inspiration from the renowned baroque composer George Frideric Händel. Just as Händel's compositions are celebrated for their elegance, precision, and gracefulness, Händel promotes these qualities in Java exception-handling code. Figure 2 presents an overview of the Händel framework, comprising three distinct phases, whose implementations we detail next.

### 3.1 Phase I: Generating Code Property Graph

In Händel's first phase, a program is converted into a code property graph (CPG). A CPG is a powerful program representation that combines a program's abstract syntax tree (AST), control flow graph (CFG), and program dependency graph (PDG) into a single, joint structure [52]. The AST represents the program's source code and syntactic structure; the CFG captures how execution flows between program statements; and the PDG encodes data dependencies throughout the program. A CPG provides the advantages of all three graphs in a single, convenient representation. While CPGs were originally designed for describing and identifying software vulnerabilities, its applicability extends beyond security-based analyses.

Because they provide a rich expression of a program's properties, we adopted CPGs as an intermediate representation for our analyses. The AST sub-graph contains Java exception-handling constructs. Each try/catch/finally structure has a corresponding sub-tree in the AST. A "try" control structure is at the sub-tree's root, and each block comprises the child nodes. We build our anti-pattern analyses by combining the exception-handling structures expressed in the AST with the control-flow relationships encoded in the CFG sub-graph. To obtain a CPG representation of a program, Händel uses Joern, a CPG framework [22].

### 3.2 Phase II: Translating CPGs into ProbLog Facts

Händel's second phase translates a given CPG into ProbLog facts representing the program's control-flow and exception-handling relationships. As depicted in Figure 2, this phase accepts a set of Händel passes as input. Each pass translates a targeted set of program constructs into ProbLog facts.

#### 3.2.1 Control-flow Pass.

This pass outputs a set of relations describing how execution transitions between program statements and methods. Specifically, it translates a program's CFG and constructs its probabilistic CFG (Prob-CFG), a weighted CFG, in which an edge's weight represents the probability of following that edge [41]. The logic for applying probabilities to control-flow edges is based on branch selectivity [40]. Each branch condition in the program is converted into satisfiability modulo theory (SMT) constraints and solved using the automata-based model counter (ABC) SMT solver [4]. This pass outputs a set of facts representing an *intra-procedural* CFG/Prob-CFG for each method. The declarative ProbLog rules then infer the set of *inter-procedural* edges, thus demonstrating an additional advantage of structuring this analysis using probabilistic logic programming. Specifically, this pass outputs the following facts:

- `method(Entry,Name)`: CFG node `Entry` is the entry point for method `Name`.
- `cfg_edge(X,Y)`: Edge between CFG nodes `X` and `Y`.
- `P::prob_cfg_edge(X,Y)`: Edge between CFG nodes `X` and `Y` with probability `P`.
- `calls(Meth,Callee,Site)`: `Meth` calls `Callee` at CFG node `Site`.
- `returns(Meth,Site)`: `Meth` returns at CFG node `Site`.

#### 3.2.2 Exception-handling Pass.

This pass traverses the CPG and identifies nodes relevant to exception-handling constructs. Specifically, it identifies exception-handling control structures (i.e., try and catch blocks), caught exception types, thrown exceptions, and propagated exceptions. This pass outputs the following exception-handling facts:

- `method_throws(Meth,Exc)`: `Meth` propagates `Exc`.
- `throws(X,Exc)`: `Exc` is thrown at CFG node `X`.
- `catches(Try,Catch,Exc)`: Control structure `Try` handles `Exc` in block `Catch`.
- `in_try(X,Try)`: CFG node `X` is in the try block of control structure `Try`.
- `in_catch(X,Try)`: CFG node `X` is in the catch block of control structure `Try`.
- `subclass(T1,T2)`: Type `T1` is a sub-class of `T2`.

### 3.3 Phase III: Identifying Probable Anti-patterns

Händel's third and final phase employs ProbLog to identify potential anti-patterns and their corresponding weights. As its logic engine, Händel utilizes Cplint [36, 37], an implementation of ProbLog provided as a library for SWI-Prolog[51]. We selected Cplint due to its full support of the ProbLog syntax and the robustness of SWI-Prolog.

Händel infers inter-procedural control edges and paths from the control flow facts extracted in Phase II. The rules in Figure 3 specify the relationships that infer inter-procedural control flow edges and paths between nodes `X` and `Y`.

The `icfg_edge` rule on line one represents intra-procedural edges. Any existing CFG edges not originating from a call should be included in the inter-procedural CFG. The rule on line two represents edges between functions. There is an edge between nodes `X` and `Y` if `X` is a call site and `Y` is the callee. The rule on line three establishes a back-edge from a called function to its call site. There is an edge between nodes `X` and `Y` if `X` is the return site of a function that was called by the node immediately preceding `Y`. The `icfg_path` rules on lines six and seven demonstrate how to infer whether a path between `X` and `Y` exists using the `icfg_edge` relationships. We introduce a nearly identical set of rules for `prob_icfg_edge` and `prob_icfg_path` by replacing instances of `cfg_edge` with `prob_cfg_edge`. These rules are integrated into additional rules that define our anti-patterns.

```
1 icfg_edge(X,Y) :- cfg_edge(X,Y), \+calls(_,_,X).
2 icfg_edge(X,Y) :- calls(_,Y,X).
3 icfg_edge(X,Y) :- calls(_,M,Z), returns(M,X),
4     cfg_edge(Z,Y).
5
6 icfg_path(X,Y) :- icfg_edge(X,Y).
7 icfg_path(X,Y) :- icfg_edge(X,Z), icfg_path(Z,Y).
```

**Figure 3.** Inter-procedural edge and path inference rules

```
1 P::exception_distance(N) :- P is 1-(1/N)+0.2.
2
3 antipattern1(Catch,CaughtExc,Throw,ThrownExc,N) :-
4     throws(Throw,ThrownExc),
5     catches(Try,Catch,CaughtExc),
6     is_subclass(ThrownExc,CaughtExc,N),
7     exception_distance(N),
8     in_try(TryNode,Try),
9     in_catch(CatchNode,Try),
10    icfg_path(TryNode,Throw),
11    icfg_path(Throw,CatchNode).
```

**Figure 4.** Specification for AP1: catch generic / over-catch

Figure 4 presents the ProbLog rule for AP1, which identifies whether an exception handler `Catch` is over-catching. If an exception is thrown along a path between a try block and its catch block, and the thrown exception is a sub-class of the caught exception, then the handler, `Catch`, may match AP1. The result is weighted via the `exception_distance` predicate. For our purposes, the further apart the two exceptions are in the type hierarchy, the more likely it is that AP1 has been matched. Depending on the audit, the weight predicate can be modified. For example, `exception_distance` can be updated to use a different formula or replaced with a different predicate that represents the targeted standard.

For brevity, we omit the AP2 and AP3 specifications. However, they are equally as comprehensible and configurable as AP1. AP2 is also defined by type subsumption, so it uses `exception_distance` as its weight predicate. The weight predicate for AP3 is derived from the Prob-CFG as the cumulative probability of the edges along a `prob_icfg_path`. If a program throws an exception along a typical—or probable—path, the exception's graceful handling should be prioritized.

**Table 1.** Summary of Händel evaluation; runtime reported in seconds (s) and memory consumption in megabytes (MB)

| | Phase III Time / Memory | | |
|---|---|---|---|
| **Benchmark** | **AP1** | **AP2** | **AP3** |
| fop-events | 14.79 / 343.5 | 36.03 / 996.3 | 37.2 / 976.3 |
| fop-sandbox | 1.4 / 21.3 | 104.78 / 2302.9 | 98.43 / 2062.5 |
| fop-util | 0.86 / 17.5 | 8.94 / 316.2 | 11.22 / 369.2 |
| h2o-algos | 0.42 / 14.7 | 0.4 / 17.4 | 0.39 / 14.4 |
| h2o-avro-parser | 0.6 / 25.2 | 0.68 / 36.1 | 0.76 / 40.8 |
| h2o-clustering | 0.42 / 14.7 | 0.56 / 36.7 | 0.65 / 39.9 |
| h2o-genmodel | 14.64 / 572.4 | 11.92 / 494 | 37.19 / 1331.7 |
| h2o-hive | 1.95 / 37.2 | 53.17 / 1809.5 | 45.45 / 1528.6 |
| h2o-orc-parser | 1.11 / 19.8 | 4.73 / 273.4 | 7.45 / 377.8 |
| h2o-persist-drive | 93.0 / 49.9 | 1.1 / 73.9 | 1.71 / 114.8 |
| h2o-persist-gcs | 0.81 / 28.2 | 1.2 / 60.9 | 1.61 / 82 |
| h2o-persist-http | 0.44 / 23.4 | 0.38 / 20.7 | 0.41 / 23.4 |
| h2o-persist-s3 | 134.67 / 3165.7 | 19.9 / 494.1 | 143.41 / 3330.9 |
| h2o-security | 0.5 / 25.3 | 0.62 / 40.5 | 0.48 / 25.1 |
| h2o-webserver-iface | 0.5 / 23.9 | 0.41 / 14.8 | 0.5 / 24.2 |
| sunflow-image | 1.83 / 24.6 | 2.52 / 124.7 | 3.09 / 205.2 |
| sunflow-system | 1.09 / 19.2 | 2.14 / 196.2 | 2.34 / 229.7 |

## 4 Evaluation

We evaluate the efficacy of Händel by applying it to 17 benchmark programs. These benchmarks were selected from the dacapo-23.11-chopin benchmark suite [7]. Specifically, we analyzed a subset of packages from the fop, h2o, and sunflow projects. These packages ranged from 150-1600 LoC with an average of ~670 LoC. For each benchmark, we ran Händel to check for each anti-pattern and captured the time and memory consumption associated with each phase. Across the 17 benchmarks, Phase I averaged a runtime of 5.53 seconds and 323.6 MB of memory, and Phase II averaged a runtime of 36.8 seconds and 4501 MB of memory. Table 1 presents the runtime and memory of Phase III.

To study how Händel performs compared to existing approaches, we implemented two additional rules that mirror those provided by checkstyle 10.16.0 [11] and PMD 7.1.0 [34], two popular static Java bug finding tools [39]. Specifically, we implemented a rule that detects when a program catches an exception that is too generic and another that detects if a method propagates a generic exception. Although reminiscent of AP1 and AP2, these rules focus on the mere presence of generic exception types rather than analyzing control-flow sensitive subsumption relationships. These specifications utilize `generic_expression` as a weight predicate, which assigns exception types arbitrary weights. Suppose an audit permits catching `Exception`; in this scenario, an auditor can either remove the corresponding `generic_exception` fact from the database or deprioritize the result by setting the probability to a small value.

We applied our generic catch, generic throw, and their comparable rules in checkstyle and PMD to our benchmark programs, measuring each framework's identified violations, runtime, and memory consumption. The results between checkstyle and PMD contained small discrepancies due to varying definitions of a "generic" exception or "illegal" propagation. However, due to the flexible and modifiable definition of our weight predicate, Händel identified the same potential violations as the other frameworks individually. Figure 5 presents the runtime and memory consumption for each framework's generic catch detection. We observed comparable metrics for each framework's throw detection.

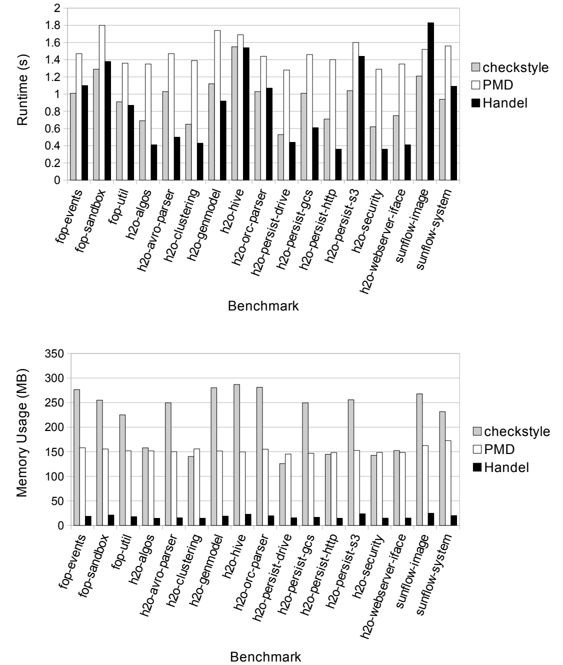



**Figure 5.** Runtime and memory performance for detecting a generic catch using Händel, checkstyle, and PMD

## 5 Discussion

Next, we discuss the implications of our preliminary evaluation, which aims to answer this question: Is probabilistic logic a suitable approach for detecting and reporting Java exception anti-patterns?

When evaluating an approach's suitability for a software auditing task, one must consider both usability and performance. The approach's programming interface should be amenable to easy expression and modification, while the resulting performance should be capable of efficiently accommodating the auditing needs. Our evaluation indicates Händel's promise in achieving both objectives, while future work will determine to what extent.

We evaluate Händel's usability by examining the expressiveness of its anti-pattern specifications. We report our findings based on our experiences constructing the ProbLog rules. Notice that the rule in Figure 4 is simple but logical. These

properties should make them amenable to comprehension and modification by software auditors. We intend further to study the expressiveness of Händel through usability studies. In terms of modification, we found that exposing Händel's configuration as ProbLog predicates presents an intuitive interface. Our design contrasts existing approaches, such as PMD or checkstyle, which utilize XML configuration files, a format designed for easier computer processing rather than human comprehension.

Often, expressiveness comes at the cost of performance. Therefore, an objective of our evaluation was to determine if the highly declarative detection logic of Händel would exhibit acceptable performance characteristics. Our benchmarks show promising performance and memory trends. The runtime of each anti-pattern varied across the evaluation benchmarks, with times ranging from less than half a second to 143 seconds, with an average of 15.9 seconds. We see a similar variation in memory consumption, which ranges from 14.4 to 3,224.4 MB, with an average of 427.7 MB. Although our benchmark programs are relatively small, the observed costs should be acceptable in most auditing scenarios. Furthermore, Händel is still in its prototyping phase, so we have not explored optimizations based on the underlying logic engine or methods to reduce the database size. We expect that applying such optimizations would strictly improve Händel's performance.

We also compared Händel with closely related tools concerned with finding defects in Java programs: checkstyle and PMD. Because these tools are not designed to detect control-flow-based defects, we introduced two additional specifications into Händel similar to closely related defect specifications in these existing tools. Across all evaluation benchmarks, Händel showed comparable or superior performance levels, which aligns with the well-known efficiency of logical inference. Furthermore, in Händel's case, the performance expenditure is a front-loaded one-time cost. Phases I and II, which are more expensive than existing approaches, only need to be executed once to establish the database of facts. All subsequent analyses can utilize the same database and benefit from the performance improvements.

## 6 Related Work

This work is related to Java anti-patterns, automated software auditing, and declarative program analysis. Prior works have defined catalogs of Java anti-patterns, including exception handling [5, 6, 12, 31, 43], dependency injection [27], concurrency [15], and performance [47] and demonstrated their presence in real software. Prior efforts have focused on creating tools to identify these anti-patterns [46, 54]. Our approach differs in specifying anti-patterns in a probabilistic logic language, thus providing weighted results.

Auditing and validating software is extremely resource-intensive, requiring significant money, developer effort, time, and expert knowledge [16, 32]. As a result, prior works have focused on automating the process [10, 30] or developing tools to identify software flaws [33]. In contrast to prior works that focus on identifying specific flaws, our work provides a more general framework for detecting software defects that can be expressed as logical rules.

Händel builds on extensive prior research on applying logic languages to solving program analysis problems. Logic languages are an effective means for specifying sophisticated and scalable analyses in a declarative manner [13, 21]. Past applications vary from calculating large-scale points-to relationships [8, 49, 50] to identifying structural program dependencies [18] or code property violations [45]. Händel's design takes inspiration from program analysis frameworks that express their analyses in a logic language [8, 23, 29]. The key novelty of our approach lies in employing *probabilistic* logic programming [14, 20, 25, 42] for specifying and executing program analyses.

## 7 Future Work and Conclusion

This paper presented a novel approach that facilitates auditing Java exception-handling logic. Our approach takes advantage of probabilistic reasoning and uses a logic language to declaratively express and configure auditing rules. We prototyped our approach in Händel. As a proof-of-concept, we revisited and specified three Java exception anti-patterns with Händel and demonstrated its ability to detect and prioritize potential anti-pattern matches.

Encouraged by our preliminary results, we plan to further study Händel's ability to specify auditing standards and detect violations. We envision our future work following three lines of inquiry. First, we plan to explore how extensible Händel is. To that end, Händel can be extended to support additional Java anti-patterns and anti-patterns in other languages. Second, we plan to explore Händel's performance and scalability. These characteristics are essential for Händel to provide practical benefits to software auditors. We can implement optimizations and expand our evaluation set to include larger code bases. Finally, we plan to explore Händel's usability further. We can conduct usability studies to understand whether Händel achieves the desired expressiveness and configurability.

As software reliability and quality remain an acute problem in software development, the need for approaches that facilitate auditing will only increase. By reporting on our experiences with Händel, we contribute novel designs and insights for using probabilistic reasoning in service of software auditing.

## Acknowledgments

The authors thank the anonymous reviewers, whose insightful comments helped improve this paper. This research is supported by NSF through the grant #2232565.

## References

[1] Martin Abadi and Luca Cardelli. 2012. *A theory of objects*. Springer Science & Business Media, New York, NY, USA.

[2] Amazon. 2024. *Amazon S3 examples using SDK for Java 2.x*. Retrieved May 1, 2024 from https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/java_s3_code_examples.html

[3] Amazon. 2024. *Developer Guide - AWS SDK for Java 2.X*. Retrieved May 1, 2024 from https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide

[4] Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan. 2015. Automata-Based Model Counting for String Constraints. In *Computer Aided Verification (CAV '15)*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Springer Cham, Cham, Switzerland, 255–272. https://doi.org/10.1007/978-3-319-21690-4_15

[5] Guilherme Bicalho de Pádua and Weiyi Shang. 2017. Studying the Prevalence of Exception Handling Anti-Patterns. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 328–331. https://doi.org/10.1109/ICPC.2017.1

[6] Guilherme Bicalho de Pádua and Weiyi Shang. 2018. Studying the relationship between exception handling practices and post-release defects. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR '18)*. ACM, New York, NY, USA, 564–575. https://doi.org/10.1145/3196398.3196435

[7] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, et al. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA '06)*. ACM, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[8] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, New York, NY, USA, 243–262. https://doi.org/10.1145/1640089.1640108

[9] William J. Brown, Raphael C. Malveau, Hays W. McCormick III, and Thomas J. Mowbray. 1998. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, Inc., New York, NY, USA.

[10] W. L. Bryan, S. G. Siegel, and G. L. Whiteleather. 1982. Auditing Throughout the Software Life Cycle: A Primer. *Computer* 15, 03 (March 1982), 57–67. https://doi.org/10.1109/MC.1982.1653973

[11] Checkstyle. 2024. *Checkstyle*. Retrieved May 26, 2024 from https://checkstyle.sourceforge.io

[12] Roberta Coelho, Jonathan Rocha, and Hugo Melo. 2018. A Catalogue of Java Exception Handling Bad Smells and Refactorings. In *Proceedings of the 25th International Conference on Pattern Languages of Programs (PLoP '18)*. The Hillside Group.

[13] Steven Dawson, Coimbatore R. Ramakrishnan, and David S. Warren. 1996. Practical Program Analysis Using General Purpose Logic Programming Systems—A Case Study. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation (PLDI '96)*. ACM, New York, NY, USA, 117–126. https://doi.org/10.1145/231379.231399

[14] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. 2007. ProbLog: A Probabilistic Prolog and Its Application in Link Discovery. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI '07)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2462–2467.

[15] Mattias De Wael, Stefan Marr, and Tom Van Cutsem. 2014. Fork/Join Parallelism in the Wild: Documenting Patterns and Anti-Patterns in Java Programs Using the Fork/Join Framework. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '14)*. ACM, New York, NY, USA, 39–50. https://doi.org/10.1145/2647508.2647511

[16] Elfriede Dustin, Thom Garrett, and Bernie Gauf. 2009. *Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality*. Pearson Education.

[17] Felipe Ebert, Fernando Castor, and Alexander Serebrenik. 2015. An Exploratory Study on Exception Handling Bugs in Java Programs. *Journal of Systems and Software* 106 (Aug. 2015), 82–101. https://doi.org/10.1016/j.jss.2015.04.066

[18] Michael Eichberg, Sven Kloppenburg, Karl Klose, and Mira Mezini. 2008. Defining and Continuous Checking of Structural Program Dependencies. In *Proceedings of the 30th international conference on Software engineering (ICSE '08)*. ACM, New York, NY, USA, 391–400. https://doi.org/10.1145/1368088.1368142

[19] Martin Fowler. 2019. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, Boston, MA, USA.

[20] Norbert Fuhr. 2000. Probabilistic Datalog: Implementing Logical Information Retrieval for Advanced Applications. *Journal of the American Society for Information Science* 51, 2 (2000), 95–110. https://doi.org/10.1002/(SICI)1097-4571(2000)51:2<95::AID-ASI2>3.0.CO;2-H

[21] Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. 2011. Datalog and Emerging Applications: An Interactive Tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data (SIGMOD '11)*. ACM, New York, NY, USA, 1213–1216. https://doi.org/10.1145/1989323.1989456

[22] Joern. 2024. *Joern: The Bug Hunter's Workbench*. Retrieved May 1, 2024 from https://joern.io

[23] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On Synthesis of Program Analyzers. In *Computer Aided Verification (CAV '16)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer Cham, Cham, Switzerland, 422–430. https://doi.org/10.1007/978-3-319-41540-6_23

[24] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. 2000. The Java (TM) Language Specification.

[25] Angelika Kimmig, Bart Demoen, Luc De Raedt, Vítor Santos Costa, and Ricardo Rocha. 2011. On the Implementation of the Probabilistic Logic Programming Language ProbLog. *Theory and Practice of Logic Programming* 11, 2-3 (March 2011), 235–262. https://doi.org/10.1017/S1471068410000566

[26] Andrew Koenig and Bjarne Stroustrup. 1990. Exception Handling for C++. *Journal of Object-Oriented Programming* 3, 2 (1990), 137–171.

[27] Rodrigo Laigner, Marcos Kalinowski, Luiz Carvalho, Diogo Mendonça, and Alessandro Garcia. 2019. Towards a Catalog of Java Dependency Injection Anti-Patterns. In *Proceedings of the XXXIII Brazilian Symposium on Software Engineering (SBES '19)*. ACM, New York, NY, USA, 104–113. https://doi.org/10.1145/3350768.3350771

[28] David C. Luckham and W. Polak. 1980. Ada Exception Handling: An Axiomatic Approach. *ACM Transactions on Programming Languages and Systems* 2, 2 (April 1980), 225–233. https://doi.org/10.1145/357094.357100

[29] Mayur Naik. 2020. *Petablox: Large-Scale Software Analysis and Analytics Using Datalog*. Technical Report. Georgia Technology Research Institute, Atlanta, GA, USA.

[30] Anh Nguyen-Duc, Manh Viet Do, Quan Luong Hong, Kiem Nguyen Khac, and Anh Nguyen Quang. 2021. On the Adoption of Static Analysis for Software Security Assessment–A Case Study of an Open-Source e-Government Project. *Computers & Security* 111 (Dec. 2021), 102470. https://doi.org/10.1016/j.cose.2021.102470

[31] Ana Filipa Nogueira, José C. B. Ribeiro, and Mário A. Zenha-Rela. 2017. Trends on Empty Exception Handlers for Java Open Source Libraries. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER '17)*. IEEE, 412–416. https://doi.org/10.1109/SANER.2017.7884644

[32] Christian Payne. 2002. On the Security of Open Source Software. *Information Systems Journal* 12, 1 (2002), 61–78. https://doi.org/10.

1046/j.1365-2575.2002.00118.x

[33] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. 2015. VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, New York, NY, USA, 426–437. https://doi.org/10.1145/2810103.2813604

[34] PMD. 2024. *PMD: An extensible cross-language static code analyzer*. Retrieved May 26, 2024 from https://pmd.github.io/

[35] George Radin. 1978. The Early History and Characteristics of PL/I. *ACM SIGPLAN Notices* 13, 8 (Aug. 1978), 227–241. https://doi.org/10.1145/960118.808389

[36] Fabrizio Riguzzi and Terrance Swift. 2010. Tabling and Answer Subsumption for Reasoning on Logic Programs with Annotated Disjunctions. In *Technical Communications of the International Conference on Logic Programming, Leibniz International Proceedings in Informatics (LIPIcs, Vol. 7)*. Schloss Dagstuhl—Leibniz-Zentrum für Informatik, Saarbrüken, Germany, 162–171. https://doi.org/10.4230/LIPIcs.ICLP.2010.162

[37] Fabrizio Riguzzi and Terrance Swift. 2013. Well–definedness and efficient inference for probabilistic logic programming under the distribution semantics. *Theory and practice of logic programming* 13, 2 (2013), 279–302. https://doi.org/10.1017/S1471068411000664

[38] Martin P. Robillard and Gail C. Murphy. 1999. Analyzing Exception Flow in Java Programs. *ACM SIGSOFT Software Engineering Notes* 24, 6 (1999), 322–337. https://doi.org/10.1145/318774.319251

[39] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. 2004. A Comparison of Bug Finding Tools for Java. In *15th International Symposium on Software Reliability Engineering*. IEEE, 245–256. https://doi.org/10.1109/ISSRE.2004.1

[40] Seemanta Saha, Mara Downing, Tegan Brennan, and Tevfik Bultan. 2022. PReach: A Heuristic for Probabilistic Reachability to Identify Hard to Reach Statements. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. ACM, New York, NY, USA, 1706–1717. https://doi.org/10.1145/3510003.3510227

[41] Seemanta Saha, Laboni Sarker, Md Shafiuzzaman, Chaofan Shou, Albert Li, Ganesh Sankaran, and Tevfik Bultan. 2023. Rare Path Guided Fuzzing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23)*. ACM, New York, NY, USA, 1295–1306. https://doi.org/10.1145/3597926.3598136

[42] Taisuke Sato and Yoshitaka Kameya. 2001. Parameter Learning of Logic Programs for Symbolic-Statistical Modeling. *Journal of Artificial Intelligence Research* 15 (2001), 391–454. https://doi.org/10.1613/jair.912

[43] Demóstenes Sena, Roberta Coelho, Uirá Kulesza, and Rodrigo Bonifácio. 2016. Understanding the Exception Handling Strategies of Java Libraries: An Empirical Study. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. ACM, New York, NY, USA, 212–222. https://doi.org/10.1145/2901739.2901757

[44] Saurabh Sinha, Alessandro Orso, and Mary Jean Harrold. 2004. Automated Support for Development, Maintenance, and Testing in the Presence of Implicit Flow Control. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*. IEEE, 336–345. https://doi.org/10.1109/ICSE.2004.1317456

[45] Leo St. Amour. 2017. *Interactive Synthesis of Code-Level Security Rules*. Master's thesis. Northeastern University, Boston, MA, USA. https://doi.org/10.17760/d20467254

[46] Ashish Sureka. 2016. Parichayana: An Eclipse Plugin for Detecting Exception Handling Anti-Patterns and Code Smells in Java Programs. (Dec. 2016). https://doi.org/10.48550/arXiv.1701.00108 arXiv:arXiv:1701.00108

[47] Catia Trubiani, Riccardo Pinciroli, Andrea Biaggi, and Francesca Arcelli Fontana. 2023. Automated Detection of Software Performance Antipatterns in Java-Based Applications. *IEEE Transactions on Software Engineering* 49, 4 (April 2023), 2873–2891. https://doi.org/10.1109/TSE.2023.3234321

[48] Bill Verners and Bruce Eckel. 2003. *The Trouble with Checked Exceptions: A Conversation with Anders Hejlsberg, Part II*. Retrieved May 15, 2024 from https://www.artima.com/articles/the-trouble-with-checked-exceptions

[49] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. 2005. Using Datalog with Binary Decision Diagrams for Program Analysis. In *Programming Languages and Systems*. Springer Berlin Heidelberg, Berlin, Germany, 97–118. https://doi.org/10.1007/11575467_8

[50] John Whaley and Monica S. Lam. 2004. Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI '04)*. ACM, New York, NY, USA, 131–144. https://doi.org/10.1145/996841.996859

[51] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. 2012. Swi-prolog. *Theory and Practice of Logic Programming* 12, 1-2 (2012), 67–96. https://doi.org/10.1017/S1471068411000494

[52] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 590–604. https://doi.org/10.1109/SP.2014.44

[53] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. 2014. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*. USENIX, 249–265.

[54] Lei Zhang, Yanchun Sun, Hui Song, Weihu Wang, and Gang Huang. 2012. Detecting Anti-Patterns in Java EE Runtime System Model. In *Proceedings of the Fourth Asia-Pacific Symposium on Internetware (Internetware '12)*. ACM, New York, NY, USA, 1–8. https://doi.org/10.1145/2430475.2430496

Received 2024-05-25; accepted 2024-06-24

# Dynamic Possible Source Count Analysis for Data Leakage Prevention


Eri Ogawa
The University of Tokyo
IBM Research - Tokyo
Tokyo, Japan
ogawa@rsg.ci.i.u-tokyo.ac.jp

Tetsuro Yamazaki
The University of Tokyo
Tokyo, Japan
yamazaki@csg.ci.i.u-tokyo.ac.jp

Ryota Shioya
The University of Tokyo
Tokyo, Japan
shioya@ci.i.u-tokyo.ac.jp



## Abstract

Dynamic Taint Analysis (DTA) is a widely studied technique that can effectively detect various attacks and information leakage. In the context of detecting information leakage, *taint* is a flag added to data to indicate whether secret data can be inferred from it. DTA tracks the flow of tainted data in a language runtime environment and identifies secret data leakage when tainted data is transmitted externally.

We found that existing DTAs can produce false negatives and false positives in complex data flows because of the binary nature of taint. Since taint is binary, meaning either secret data is inferable (=1) or non-inferable (=0), it cannot represent intermediate states that may slightly infer the secret data, and these states are quantized to 0 or 1. As a result of this quantization, existing methods are unable to distinguish between outputs that are practically secure and those that pose a real security threat in complex data flows, resulting in false positives and false negatives.

To address this problem, we introduce the concept of Possible Source Count (PSC) and propose Dynamic Possible source Count Analysis (DPCA), which tracks PSC instead of taint. PSC is a metric that indicates how many secrets can be identified by observing the data. DPCA tracks and computes the PSC of each data item using dynamic symbolic execution. By evaluating the PSC of data that reaches the sink point, DPCA can effectively distinguish between data that is practically secure and data that poses a security threat.



***CCS Concepts:*** • **Security and privacy** → **Information flow control**; • **Software and its engineering** → Runtime environments.

***Keywords:*** Taint Analysis, Information Flow Control, Security, Information Leakage

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.



*MPLR '24, September 19, 2024, Vienna, Austria*


© 2024 Copyright held by the owner/author(s).


ACM ISBN 979-8-4007-1118-3/24/09
https://doi.org/10.1145/3679007.3685065

**ACM Reference Format:**
Eri Ogawa, Tetsuro Yamazaki, and Ryota Shioya. 2024. Dynamic Possible Source Count Analysis for Data Leakage Prevention. In *Proceedings of the 21st ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '24), September 19, 2024, Vienna, Austria.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3679007.3685065


## 1 Introduction

With the increasing importance of security, information leakage prevention methods using *Dynamic Taint Analysis (DTA)* [44] have been widely studied. DTA is a method that is integrated into the runtime environment or hardware, and it can effectively prevent information leakage without relying on pattern detection. In an information leakage prevention method using DTA, a flag called *taint* is assigned to secret data to be protected. The taint is propagated from input to output during computation to track the flow of data containing the taint. During the flow tracking, tainted data that is about to be output to the outside world is detected as a leakage of important information.

In taint propagation, handling a data flow known as an *implicit flow* [5, 38, 42] is generally difficult. The implicit flow is a data flow in which the output is determined depending on the input, even without explicit operations or assignments. A typical example of the implicit flow is a control flow via a conditional branch. When secret data is used in the conditional expression of a branch, the information of the secret data can be indirectly obtained by observing the result of the branch. For example, in a statement such as `if(x==0){y=1}`, by observing whether `y` is `1`, one can infer whether the value of `x` is `0` or not.

Some existing methods propagate taint in implicit flows [10, 13, 20], but simply propagating taint in all implicit flows often results in inappropriate taint propagation [10]. For example, consider a program that checks whether the length of an input password meets a specific length requirement. In this case, the password, which is secret data, must be used as the input to the conditional expression that checks the length. Suppose that the taint is propagated to all variables assigned in the then/else statements. Then, the taint will also be propagated to the result "the password length meets the requirement", and this result cannot be output because it is tainted. This phenomenon, known as *over-tainting*, involves

taint propagation to data that does not reveal the original secret, thus preventing program operation. Although the above example is very simplistic, Section 3 illustrates how over-tainting makes it almost impossible for a web application to communicate with an external server.

To overcome such over-tainting problems, some existing methods selectively propagate taint by following specific heuristics [13, 20]. Such methods propagate taint when an exact value comparison is made, for example, `if(x==0){y=1}`, since the value of `x` can be inferred from `y`, as inferred from the earlier example. On the other hand, they do not propagate taint in cases of greater than or less than comparisons, such as `if(x>0){y=1}`, because it is difficult to estimate the input from observing `y`. However, following such heuristics fails to handle cases where `if(x>0){y=1}` and `if(x<2){y++}` are executed in combination. In such a case, if `y` equals 2, it implies that `x` is 1. Thus, taint should also be propagated to `y` in such cases, but this is often omitted in the above heuristic-based methods. Such an omission of taint propagation is called *under-tainting*.

The problems described stem from taint being binary, indicating whether secret information might be inferred from certain data. For example, knowing whether a password satisfies specified length requirements allows only negligible inference about the password itself. If we strictly propagate taint based on even the slightest potential for inferring the password, such information cannot be displayed, despite posing no significant security threat. Conversely, omitting taint propagation based on heuristics can lead to potential security risks, as input values might be inferred from a combination of branching conditions.

Building on the discussion above, we propose *Dynamic Possible Source Count Analysis (DPCA)*, a method that quantifies recoverable secret information at each data, rather than merely determining if secret information can be inferred. We introduce a metric called a *possible source count (PSC)*, which quantifies the number of identifiable inputs by observing each data. For instance, output data with a PSC of 1 indicates a high risk, as the input is determinable in a single way. Conversely, a PSC of $2^{128}$ suggests the input is practically impossible to deduce, thereby signifying safety. DPCA assigns a PSC to data, tracking its flow to assess the magnitude of PSC at sink points. This method enables DPCA to effectively distinguish between outputs that are practically safe and those that pose genuine security threats. Additionally, DPCA monitors control flows that influence each data and effectively detects leaks stemming from complex combinations of conditional branches.

Our contributions are summarized as follows:

- We introduced PSC, which quantifies the number of identifiable inputs by observing data. Using PSC, we formulated existing heuristic-based dynamic taint propagation issues.
- We propose DPCA, which computes and tracks the PSC of data using dynamic symbolic execution. DPCA evaluates the PSC of data that has reached the sink point, effectively distinguishing between output that is practically secure and output that poses a security threat. As a result, DPCA solves the under-tainting and over-tainting issues in existing DTAs.
- We implemented DPCA on the JavaScript code instrumentation platform and showed that it can more accurately evaluate programs that existing methods cannot correctly detect for information leakage.

**Listing 1.** Example of an malicious program

```
1  // e-mail address is secret data
2  function checkAddress(email) {
3    // Validate the email format
4    if (email.length == 0)
5      return false;
6    // Check other rules...
7    ...
8    // Maliciously send the email to a remote server
9    fetchSync("https://example.com/api?" + enc(email));
10   // Assume the email is valid
11   return true;
12 }
```

## 2 Background

### 2.1 Assumed Information Leakage

We base our discussion of information leakage on the following assumptions: A program receives secret information (e.g., password, credit card number, and identification ID) as input and uses this information to perform operations. When the program communicates with external parties such as servers or third-party APIs, if the data sent externally includes secret information, this can be observed by a third party, potentially leading to information leakage. We assume that the attacker can change and observe a target program (e.g., source code) and can observe only public output.

A typical example of the above scenario is when an attacker distributes a program containing malicious code under the guise of a useful program or library, leading the victim to download and execute it on their own computer. In particular, here we assume a case where a script downloaded from a server is executed on a web browser. Listing 1 is an example of such a script. This script contains a function that checks whether the email address has the correct formatting, contained within a library distributed by the attacker. The function secretly sends the email address to an external server via a code inserted in line 9. In this example, `email` is sent directly to the external server, and thus we can easily see that it contains an attack, but an attacker can cleverly rewrite the program to conceal the attack in various ways. The objective of our research is to prevent such information leakage caused by attack programs without interfering with normal program execution.

**Listing 2.** Example of implicit flow

```
let y;
if (x == 0)
  y = 0;
else
  y = 1;
```

## 2.2 Preventing Information Leakage with DTA

DTA is a method implemented in a language runtime environment or hardware to detect and prevent a variety of attacks [44]. The application of DTA to prevent information leakage has been widely studied [15, 20, 29, 43, 49, 51]. In the information leakage prevention method using DTA, a flag called *taint* is added to secret data to be protected. The taint is propagated from input to output during computation to track the flow of data with the taint. As a result of the flow tracking, data with the taint detected being output to the outside is identified as a leakage of important information.

Information Flow Control (IFC) is a technique for analyzing information flow in a program to detect vulnerabilities and prevent information leakage. Especially for dynamic languages such as JavaScript, dynamic IFC has been widely studied due to its dynamic nature [1, 3, 9, 16, 17, 37, 39].

Dynamic IFC analyzes information flow similarly to DTA. Dynamic IFC assigns H and L (high and low) security levels to each variable, similar to taint, and tracks how the security level of each variable changes with the information flow. The label of each variable is propagated from its input to its output for each operation or flow, preventing the H level from leaking out. Since Dynamic IFC performs a similar analysis to DTA, which is described in detail later, we consider IFC as part of DTA.

## 2.3 Tracking Information Flow in DTA

In this section, we divide data flows into explicit and implicit flows and explain how DTA tracks them. An explicit flow is a data flow where there is a direct dependency between data through assignments or operations. For example, consider the operation $y = x + 1$. In this operation, by observing the output value $y$, one can completely determine the input value $x$. In this way, observing the output in explicit flows enables us to determine the input value, fully or partially. Thus, in DTA, if the input value on the explicit flow is tainted, it is propagated to the output.

An implicit flow is a data flow where information is propagated between data through conditional branches. In an implicit flow, there is no explicit dependency between data through operations or assignments. More specifically, in an implicit flow, the output variable updates differently based on the input variables in a conditional expression of a branch. In such a case, information about the input can be obtained from the conditional expression and the output value. Listing 2 presents an example of implicit flow. In this example, by observing the output value $y$ after the code executes, one can determine whether the input $x$ is 0.

**Listing 3.** Example of web application

```
function webApp(password) { // password is tainted
  if (!isAuthorized(password)) {
    return false; // Authorization failure
  }
  else {
    let url = "http://server.com/api?ok";
    let response = fetchSync(url);  // sink point
    renderHTML(response);
    return true;
  }
}
```

If there is an implicit flow and secret data is used as the input in the conditional expression, the information from the secret data can be obtained from the output, as described above. For this reason, existing DTA-based methods that propagate the taint in implicit flows have been proposed [10, 13, 19, 20]. In the simplest implementation of taint propagation in implicit flows, when tainted data is used in a conditional expression, the taint is propagated to all variables updated within the then/else statements. For example, in the Listing 2 example above, if $x$ has a taint, the taint is propagated to $y$. In the next section, we detail DTA that propagate taint in implicit flows and discuss their associated problems.

# 3 Issue of Dynamic Taint Analysis

In this section, we will discuss in detail the challenges associated with taint propagation in implicit flows, specifically *over-tainting* and *under-tainting*."

## 3.1 Over-tainting

Over-tainting is a phenomenon where taint is unnecessarily propagated to data that provides no clues for recovering the original secret data. This can prevent the program from operating correctly. In particular, over-tainting often occurs when taint is indiscriminately propagated in implicit flows.

One such method that causes over-tainting is Dytan [10]. Dytan enables taint propagation in implicit flows by effectively detecting data dependencies through control flow graph (CFG) analysis. However, Dytan indiscriminately propagates taint from all inputs in the conditional expression to all variables updated in the then/else statements, often leading to over-tainting [13, 20]. Similarly, Dynamic IFC propagates labels to its output in all implicit flows, resulting in over-tainting comparable to that caused by Dytan [3].

For example, consider a typical web application such as Listing 3. The program authenticates itself locally using the secret information, `password`. If the authentication succeeds, the program issues an HTTP request using a specific URL, obtains the information necessary for HTML rendering from the server, and renders the HTML using this information. Through the conditional branch in the second line,

**Listing 4.** Example of information leakage through a combination of branches

```
1 for (let i=0; i < password.length; i++) {
2   x = password[i].charCodeAt();
3   let y = 0;
4   if (x >= 48)
5     y++;
6   if (x <= 57)
7     y++;
8 }
```

an implicit flow transfers taint from the password variable to the url variable, as seen in the sixth line. Propagation rules such as Dytan and IFC propagate the taint held by the password to the url in such a case. The fetchSync API that performs HTTP access is a sink point, and arriving tainted data prompts DTA to detect a potential information leakage. However, the server cannot recover the contents of the password from the transmitted data and it is practically secure.

As shown in the above example, indiscriminate taint propagation in implicit flows can hinder server communication after a branch using tainted data, even if the code is secure. This makes writing practical programs challenging. Because conditional branches with secret data are common, Dytan and IFC frequently result in widespread over-tainting, severely disrupting normal program operations.

### 3.2 Under-tainting

To overcome the over-tainting issue, existing research [13, 19, 20] has proposed a method to selectively propagate taint for each conditional branch. These methods heuristically select branches unlikely to lead to information leakage by avoiding taint propagation for these branches. In this way, the existing methods aim to prevent information leakage via implicit flows without interfering with the correct operation of programs not susceptible to attacks.

The existing methods focus on how much the conditional expression restricts the range of inputs. The more limited the possible values of the input when the conditional expression is satisfied, the easier it is to determine the input based on the output. In existing selective propagation methods, taint is propagated only for conditional branches where the range of inputs is strongly restricted. For example, as mentioned above, if $(x == 0)$ in line 2 of Listing 2 holds, $x$ can be uniquely determined as 0 by observing $y$. Such a precise comparison significantly limits the input range, and since $x$ can be easily determined from $y$, the existing methods propagate the taint in such instances.

However, these existing methods of selective propagation have the problem of under-tainting, in which necessary parts of the taint propagation are omitted. Specifically, information leakage can be performed by combining branches with weakly restricted inputs.

Listing 4 is an example of a program that allows such information leakage. If password is secret information and tainted, taint is propagated by explicit flow to x. The conditional branches at lines 4 and 6 have implicit flows that include tainted values in the conditional expressions, but each conditional branch has a wide range of possible values to make the conditional expression true. For this reason, existing methods assume that the input cannot be identified from the output in these conditional branches and do not propagate taint.

However, upon complete execution of this code, it may be possible to determine the possible value range for each character in x, i.e., password, by observing y. For example, if the value of y is 2, the range of possible values of x can be narrowed down to $48 <= x <= 57$, representing the ASCII digits 0 to 9. This is because while the individual conditions $x >= 48$ and $x <= 57$ each cover a wide range, their conjunction $x >= 48 \&\& x <= 57$ significantly narrows the range. Thus, taint should be propagated to y, and existing methods that omit this propagation result in under-tainting.

## 4 Possible Source Count

### 4.1 Overview

We introduce a Possible Source Count (PSC), which quantifies the number of identifiable inputs by observing data, contrasting with binary taint analysis, which determines whether secret information can be inferred. A smaller PSC makes it easier to identify the secret data, increasing the risk of data leakage. For example, a PSC of 1 means the input is inferable to a single value, indicating high risk upon observation. Conversely, a PSC of $2^{128}$ indicates that inferring the secret input is practically impossible, indicating safety upon observation. In short, PSC measures the difficulty of recovering the original secret from observed data.

Using the example shown in Figure 1, we describe PSC. The secret information input by the user is called *source secret*, and the data output by the program, observable from the outside, is called *public output*. Figure 1 shows that a program may receive five possible source secrets (s1 - s5) and produce two possible public outputs (o1, o2). The program outputs o1 if s1 through s4 are input, and o2 if s5 is input. Assume

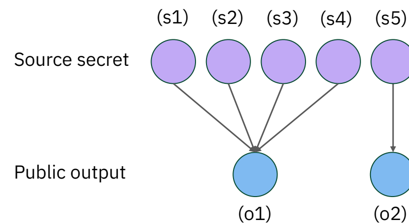



**Figure 1.** A relationship between source secrets and public outputs

**Listing 5.** PSC examples

```
1  // source secret data (32bit integer)
2  x1 = readSource();          // PSC(x1)  = 1
3  // binary operation
4  x2 = x1 * 3;     // PSC(x2) = 1
5  x3 = x1 & 7;     // PSC(x3) = 2^29
6  // if-statement
7  if (x1 == 0) {
8      x4 = 0;                // PSC(x4) = 1
9  } else {
10     x4 = 1;                // PSC(x4) = 2^32-1
11 }
```

that an observer knows all these source secret and public output relationships in advance. Generally, we assume the relationship between source secrets and public outputs is known if the observer can read the program source code.

If o1 and o2 are observed, the PSC is as follows:

- When the observer detects o1 as a result of the program execution, the source secret was one of s1 to s4. In this case, the PSC for o1, representing the number of possible source secrets, is 4.
- Conversely, when o2 is observed, the observer can conclusively determine that the source secret was s5. Here, the PSC for o2, indicating the uniqueness of the source secret, is 1.

It should be noted that the PSC is not a static value but a dynamic one, determined by the public output observed in a single program execution. In other words, if the input changes with each execution, a different PSC will be observed. In the given example, the PSC when observing o2 is smaller than when observing o1, making it riskier to observe o2

### 4.2 PSC Calculation

This section describes more concrete examples of PSCs in program execution, as shown in Listing 5. In the second line of Listing 5, a 32-bit integer is read as the source secret and assigned to `x1`. Since the source secret comprises the secret data itself, it can be uniquely identified by observing this data, and thus $PSC(\texttt{x1})$ is 1.

The PSC for the result of an operation, such as addition or multiplication, varies depending on the operation type as follows:

- In line 4 of Listing 5, `x1` multiplied by 3 is substituted into `x2`. When the observer knows that `x2` is the result of the operation `x2 = x1 * 3`, then by observing `x2`, `x1` can also be uniquely identified. Furthermore, since $PSC(\texttt{x1})$ is 1, the resulting $PSC(\texttt{x2})$ is also 1.
- In line 5, the bit-wise AND of `x1` and 7 is substituted into `x3`. When the observer observes `x3`, he can identify the lower 3 bits ($7 = 2^3 - 1$) of `x1`, but not the upper 29 bits because they are cleared. As a result, the observer can identify `x1` within $2^{29}$ possibilities, and $PSC(\texttt{x3})$ becomes $2^{29}$.

In an if-statement, the PSC obtained may vary greatly depending on the conditional expression used and the dynamic execution flow as follows.

- In line 8, `x4` is assigned 0 if `x1 == 0`. In this case, $PSC(()\texttt{x4})$ becomes 1. This is because when an observer observes that `x4` is 0, the observer knows that `x1` was 0, allowing `x1` to be uniquely identified.
- Conversely, if 1 is assigned to `x4` through the else statement in line 9, then $PSC(\texttt{x4})$ becomes $2^{32} - 1$, which equals 4294967295. When the observer observes that `x4` is 1, they only know that `x1` is not 0. Therefore, the PSC is the number of all possible values of a 32-bit integer minus one.

### 4.3 PSC and Security Risk

In this section, we discuss the values of PSC and the degree of risk in information leakage using several examples. The magnitude of the observed PSC significantly reflects the risk level when information is observed. For example, we often see on receipts that only the last four digits of a credit card number are printed, while the other digits are obscured. Although credit card numbers are very sensitive personal information, these last four digits are generally not considered to be a critical secret. When these last four digits are observed by someone, considering the credit card number has 16 digits each ranging from 0 to 9, the PSC of this data is $10^{16-4} = 10^{12}$, representing a vast number of possibilities. Thus, even if the last four digits of a credit card number are observed, there are still $10^{12}$ possible combinations, making it difficult to precisely identify the complete credit card number.

On the other hand, leaking data with low PSCs is highly risky. For example, if instead of the last four digits, the last 12 digits of a credit card number are leaked, the PSC of the last 12 digits is $10^{16-12} = 10,000$. This is particularly concerning for credit card numbers, as they could potentially be brute-forced on a website. When the number of possible entries is narrowed down to just several thousand, it becomes feasible for an observer to correctly identify the complete credit card number.

### 4.4 Problem Analysis with PSC

We consider the issues associated with tainting described in Section 3 in terms of PSC as follows:

#### 4.4.1 Over-tainting.

As described in Section 3, when taint is propagated indiscriminately across all branches, over-tainting can occur, disrupting the normal execution of the program. We found that over-tainting stems from the propagation of taint to safe variables with a large PSC, where it is impossible to infer the inputs.

Consider the example of a web application shown in Listing 3, specifically a scenario in which the application accesses an external server on line 7. This access signifies successful

password authentication in the local environment to an external server; however, the external server cannot feasibly recover the password.

For this reason, we can consider its output to be generally secure; however, a DTA that indiscriminately propagates taint on all branches regards it as dangerous and prevents any access to external servers. Specifically, if the DTA propagates taint through an implicit flow from `password` to `url`, the tainted data will be used in a `fetchSync` operation and blocked. Nonetheless, given the high PSC of `url`, deducing `password` from it is virtually impossible, access to the external server is practically secure.

#### 4.4.2 Under-tainting.

As described in Section 3, selectively propagating taint according to the heuristics leads to under-tainting, where information leakage occurs depending on the combination of branches. This happens because the DTA, when selectively propagating taint, omits taint propagation if the secret source is considered unidentifiable within each independent conditional branch. However, by considering multiple combinations of branches, an observer can deduce the secret source from the public output. This means that if the PSC of the public output is very small due to the combination of branches, the existing DTA may erroneously omit the propagation of taint.

### 4.5 Relationship between Taint and PSC

From the above discussion, we conclude that it is necessary to allow the leakage of data with a so large PSC that the secret data cannot be recovered in order to allow normal execution of the program, while it is necessary to properly detect information leakage of data that results in having a small PSC by passing through multiple flows.

We consider the PSC to be an extension of taint. When taint is 0 (= the data from which no secret information can be recovered), the PSC for that data has a maximum value of $2^{\text{\{bit width of the source secret\}}}$ (= observing the data does not identify the secret information at all). On the other hand, when taint is 1 (= the data uniquely identifies the secret information), its PSC has a minimum value of 1 (= observing the data can uniquely identify the secret information).

Actual applications have many variables with the PSC between the minimum and maximum. For example, although a variable with a PSC of (`{maximum PSC}-1`) is data that is only slightly related to the source secret, its PSC is almost the same as the maximum value, and the observer cannot identify the source secret if the bit width of the source secret is sufficiently large. However, taint, being binary, cannot represent such data well. Even for data with a PSC sufficiently large that the secret data cannot be recovered, existing methods that propagate at all branches set taint to 1, causing over-tainting. On the other hand, the method that selectively sets taint to 0 at conditional branches, allows leakage of the secret information even though the public output has a small PSC in some cases. In other words, over-tainting and under-tainting are caused by rounding taint to 1 or 0 because taint is binary and cannot express the size of the PSC.

## 5 Dynamic Possible Source Count Analysis

We propose the Dynamic Possible source Count Analysis (DPCA), a method that propagates and tracks PSCs. DPCA monitors how secret sources are utilized in computations across different operations and branches. It then estimates the PSC of public outputs at the sink point, using this data to assess the associated risk levels.

### 5.1 Naive Method

As mentioned above, PSC quantifies the number of possible secret sources that can be identified by observing data. A straightforward method to determine PSC involves inputting all possible input patterns into a program, executing it, and creating a brute-force map of inputs to outputs. By comparing this map to the actual output of the program, the PSC for that output can be determined. For example, in the case of Figure 1, we can create a map by recording the outputs for all inputs from s1 to s5, and the PSC can be computed from the map when o1 or o2 is actually observed.

However, a brute-force method that operates with all possible inputs is impractical. Therefore, we propose a more lightweight method that dynamically computes the PSC for outputs during a specific execution, based on their values and the control flow paths they traverse. Specifically, DPCA actively tracks both explicit and implicit data flows involving computations with secret sources and computes the PSC based on this tracked flow information when the data is output. Further details of this method are described below.

### 5.2 Possible Source Count Tracking

#### 5.2.1 Tracking Symbol and Path Constraint.

DPCA dynamically computes and records the symbol (sym) and path constraint (pc) of each variable in the program whenever it is updated. These terms are almost synonymous with symbol and path constraint in symbolic execution. However, in DPCA, only secret sources are represented as symbols, while all other variables in the program are recorded with the specific values used in their computations.

We use Listing 6 to illustrate an example of sym and pc tracking. The detailed rules are described in Section 5.4.

**Listing 6.** Examples of sym and pc

```
1 let secret = read();
2 let x = secret + 1;
3 let y = 1;
4 let z = y;
5 if (secret > 1)
6     y = 2;
7 if (secret < 4)
8     y++;
```

**Listing 7.** PSC Computation

```
let psc = 0;
for (let i = min(secret); i < max(secret); i++) {
  if (apply(sym,i) == public_output && apply(pc,i))
    psc++;
}
```

- In the first line, a variable `secret` is used to read the source secret. At this assignment, `sym: secret` is saved. Since there is no specific path constraint (pc) here, `pc: true` is preserved. In cases where pc is not restricted, further mentions of pc are omitted.
- In the second line, the value 1 is added to `secret`, and `x` stores `sym: (secret + 1)`. This operation, which is an example of an explicit flow, updates the sym accordingly.
- In the third and fourth lines, the sym for `y` and `z` are updated to `sym: 1`, indicating that these variables are not directly related to the source secret. For such variables, runtime values are directly assigned, which simplifies the storage of sym and pc and reduces the computational overhead.
- Due to the implicit flow in line five, `y` now stores `sym: 2, pc: (secret > 1)`.
- Each time it encounters an implicit flow, the pc is combined using an AND operation. In line 8, `y` updates to `sym: 3, pc: (secret > 1 AND secret < 4)`.

DPCA is a dynamic method, and unlike static symbolic execution, sym and pc use actual values obtained dynamically for variables other than the source secret. These values reflect the specific conditions and inputs of the current execution, along with the constraints of the path traversed. This approach allows PSC computation to be performed in a more lightweight manner.

#### 5.2.2 PSC Computation.

Listing 7 shows a naive way to compute PSC using sym and pc of the public output, as previously described. `min(secret)` and `max(secret)` indicate the minimum and maximum values that can be taken by `secret`. For example, if the source secret is an 8-bit integer, the minimum and maximum values would be -128 to 127, respectively. `public_output` is the value of the public output that was actually observed. `apply(sym,i)` computes the result of substituting the value of `i` for `secret` in the expression stored in sym. For instance, if `sym: (secret + 1)`, then `apply(sym,2)` would yield 3. `apply(pc,i)` similarly returns true or false when `i` is substituted. The `psc` obtained by executing this algorithm reflects the PSC that the public output represents.

This algorithm computes the number of all possible source secrets that satisfy the constraints imposed by both explicit and implicit flows which determine the public output, which is equivalent to PSC. Although this algorithm is significantly less computationally expensive than running the entire program, it may still be computationally expensive to try all possible source secrets.

We introduce several heuristics to obtain a PSC in a lightweight manner. Specifically, a pc is analyzed to reduce the number of candidate source secrets that need to be evaluated. For example, if a pc includes a matching comparison such as `{secret == 0}`, only one possible value for the secret satisfies it. If a pc involves a range, such as `{secret > 0 && secret <= 10}`, we determine the lower (1) and upper (10) limits for `secret`. Our implementation then computes the number of satisfying values as 10 by subtracting the lower limit from the upper. By employing heuristics for such conditional expressions, a PSC can be computed in most cases without needing to execute the entire for-loop.

#### 5.2.3 Discussion on Errors.

The PSC computed by the algorithm described may differ from that obtained through brute force execution of the entire program, potentially leading to an overestimated risk. This is because the algorithm assumes that an observer can uniquely identify the path taken through the program based on the output value, simplifying the analysis to avoid complexity.

For instance, in the program `if(sec==0) out=0; else out=1;`, observing `out` directly indicates whether the 'if' or 'else' branch was executed, revealing that `sec` is 0 when `out` is 0. Conversely, in the scenario `if(sec==0) out=0; else out=0;`, observing `out` does not disclose the value of `sec`, as it remains 0 regardless of the branch taken.

DPCA operates under the assumption that different execution paths assign different values to variables. This assumption can lead DPCA to compute a PSC that is lower than the actual PSC obtained by brute force, suggesting a higher risk of information leakage than is truly present and potentially causing a false positive. However, this does not result in false negatives. The practice of assigning identical values in different paths is rare, implying that false positives are seldom a concern in subsequent evaluations.

### 5.3 Information Leakage Detection

If a small PSC is computed for data at a sink point, DPCA interprets this as an indication of information leakage, suggesting that the source secret could potentially be inferred. In such cases, a threshold value for detecting information leakage should be set based on the characteristics of the source secret. Typically, there is a noticeable discrepancy in PSC values between scenarios where information is leaked and where it is secure. Thus, we can establish a relatively high threshold value for PSC to enhance safety without interfering with correct program operation. Details on the observed PSCs and their implications are further discussed in Section 6.

**Table 1.** Propagation rules within explicit flows.

| Operation | Example | Propagation |
|---|---|---|
| source secret | `sec_input()` | *sym* : (sec) *pc* : () |
| internal data | `1` | *sym* : (1) *pc* : () |
| assignment | `y = x` | *sym* : (`x.sym`) *pc* : (`x.pc`) |
| unary operation | `-x` | *sym* : (-(`x.sym`)) *pc* : (`x.pc`) |
| binary operation | `x1 + x2` | *sym* : ((`x1.sym`) + (`x2.sym`)) *pc* : ((`x1.pc`)&&(`x2.pc`)) |

**Table 2.** Propagation rules within implicit flows.

| Operation | Example | Propagation |
|---|---|---|
| if-statement | `if(x) {...}` | *sym* : () *pc* : (`x.sym=={x value}`) |
| array access | `x = table[key]` | *sym* : ((`table[key].sym`)) *pc* : ((`table[key].pc`)&& (`key.pc`)&& (`key.sym == {key value}`)) |
| property access | `x = Obj.key` | *sym* : (`Obj.key`) *pc* : (`Obj.pc`) |

### 5.4 Detailed Tracking Rules

Section 5.2 outlined the basic methods for tracking sym and pc; this section describes their more detailed tracking rules.

**5.4.1 Explicit Flows.** Propagation rules for sym and pc in explicit flows are summarized in Table 1. In these rules, `x.sym` represents the sym associated with variable `x`. Notably, only secret sources are symbolized, while all other variables are recorded with the specific values used during program execution. These rules are applied to all statements involving explicit flows. Examples illustrating the application of these rules in actual programs can be found in Section 6.

**5.4.2 Implicit Flows.** Table 2 summarizes the propagation rules for three typical implicit flows. In these flows, each pc is systematically combined using an AND operation with the conditional expressions it encounters, as described in Section 5.2.

***If-statement:*** As illustrated in the table, within an if-statement, the sym of the conditional expression is added as the pc of the variable assigned in the conditional clause. The term `{x value}` in the table refers to the runtime value of `x`. For example, consider the expression `if(x){y=1}`, where `x` is a boolean variable assumed to be true. Suppose `x` has `{sym: (secret<0)}` and the value (true) arises from an explicit flow involving a comparison with `secret`. In this scenario, `y` would be assigned `{sym: (1), pc: ((secret<0)==true)}`. This stores the comparison between the sym of `x` and its actual value (true) in the pc of `y`. If `x` were false, then `{pc: ((secret<0)==false)}` would be recorded as the pc for `y`. Typically, the conditionals in if-statements are evaluated as boolean variables following explicit flow rules, and the propagation rules described in this section are then applied.

***Array Access:*** In array access, both the retrieved elements and the key determine the sym and pc of the output. In the table, `{key value}` denotes the runtime value of `key`. The rule `{(key.sym == {key value})}` in the pc reflects an implicit flow, where the value of `key` indirectly determines the output when the array is accessed with `key`. For instance, if `key` has `{sym: secret}` and its runtime value is 2, then `{sym: (secret==2)}` is incorporated into the output PC. If `key` does not contain any source secret in its sym, then the condition `{(key.sym =={key value})}` is invariably true and can be omitted.

***Property Access:*** In JavaScript, property accesses, such as accessing the length of an object, are recorded directly in the sym and used in PSC computation. For example, in the expression `y = x.length`, if `x` has a sym and pc represented as `{sym: secret, pc: ()}`, then the sym and pc for `y` become `{sym: secret.length, pc: ()}`. This indicates that the sym for `y` directly inherits `secret` appended with `.length`, reflecting the property access.

An analysis of the control flow graph (CFG) is essential for tracking implicit flows, as it allows for the identification of each conditional branch's scope of influence, which is crucial for applying the rules described. DPCA is implemented on a JavaScript code instrumentation platform, details of which are provided in Section 6. The CFG analysis is performed by this infrastructure, facilitating the tracking of control and data dependencies.

## 6 Evaluation

### 6.1 Evaluation Methodology

We implemented DPCA as a JavaScript framework designed to process JavaScript programs by taking them as input and integrating PSC tracking directly into the application's source code through instrumentation. For code instrumentation, we utilize Linvail [8], a platform suited for implementing shadow executions. Shadow executions involve tagging runtime values with analysis-specific data, enabling DPCA to dynamically and accurately track the flow of secrets within the program.

Linvail is capable of inserting individual traps for various JavaScript statements—such as function calls, if-statements, and binary operations—and can execute arbitrary processing within these traps. We have integrated the PSC tracking process, described in the previous section, into these traps using Linvail. DPCA stores sym and pc for each variable managed by Linvail and updates these attributes within the traps as part of the PSC tracking process.

We implemented two functions, `tagAsSource(x)` and `tagAsSink(x)` for our evaluation. `tagAsSource(x)` is designed to receive a variable that holds a source secret, signaling to DPCA to manage and track it as such. `tagAsSink(x)` designates its input as a sink point. For instance, specifying `tagAsSink(console.log)` instructs DPCA to treat

console.log as a sink point, thereby enabling the framework to perform a PSC check on any data input to the specified function.

As with DPCA, we implement Dytan [10], DTA++ [20], and GIFC [39] in Linvail to compare detection accuracy and performance with DPCA.

### 6.2 Benchmarks

For our evaluation, we use the following benchmarks:

- Four attack programs from AntiTaintDroid [36]
- Five attack programs from GIFC [39]
- Four benign programs:
  - Program implementing password authorization
  - Program implementing password length validation
  - Program implementing password equivalence check
  - Program implementing device ID check

AntiTaintDroid [36] comprises a set of benchmark programs designed to test taint propagation on Android devices. We have ported the programs related to implicit flow to JavaScript. Additionally, we have selected five IFC accuracy benchmarks from GIFC [39]. We show the five results within 32 benchmarks since they uniformly exhibit the same outcomes in this evaluation. Both AntiTaintDroid and GIFC benchmarks are designed to test for information leakage, specifically measuring the ability of DPCA and other existing tools to correctly detect such leaks.

On the other hand, we prepared four benign programs that do not leak information to test whether a web application can operate correctly with information leakage detection tools. These benign programs are presented from Listings 8 to 11. Each program represents a segment of code typically used in typical web applications that manage secret information. While these programs handle secret information, the output they produce is related to but insufficient to reconstruct the original secrets. Therefore, they qualify as benign programs that do not leak secret information. These benchmarks are used to verify whether the execution terminates correctly without any false detection of information leakage. Both programs use the strings 'temp1234' and 'temp0123' as source secrets in cases involving two distinct inputs. These source secrets, provided at runtime, remain unobserved by attackers.

### 6.3 Detection Accuracy

Table 2 presents the taint, label, and PSC attached to the public output by existing methods (DTA++, Dytan, and GIFC) and the proposed method (DPCA). The term "labeled" in GIFC has the same meaning as "tainted" in DTA. The columns labeled 'sym, pc' list the symbols (sym) and path constraints (pc) for variables at the sink point, focusing only on the parts vital for PSC computation. PSC is derived using the method described in the previous section. For clarity, outputs that were not tainted/labeled despite potential information leakage are marked with (FN), and outputs that were tainted/labeled without intended leakage are marked with (FP).

**Listing 8.** passAuthorization: Program with password authorization

```
1  tagAsSink(fetch);
2  let pass = tagAsSource(pass);
3  function is_authorized(pwd) {
4    let cachedPass = null;
5    let cookies = document.cookie;
6    let cookiesArray = cookies.split(';');
7    for(let c of cookiesArray){
8      let cArray = c.split('=');
9      if(cArray[0].indexOf('pass') > -1){
10       cachedPass = decodeURIComponent(cArray[1]);
11     }
12   }
13   return (cachedPass == pwd);
14 }
15
16 if (!is_authorized(pass)) {
17   return;
18 }
19 let url = "http://server.com/api?ok";
20 fetch(url).then(response => {
21   return response.json();
22 })
```

**Listing 9.** passLenCheck: Program with password length validation

```
1  tagAsSink(console.log);
2  let pass = tagAsSource(pass);
3  let str;
4  if (pass.length >= 8) {
5    str = "valid password!"
6  } else {
7    str = "invalid password"
8  }
9  console.log(str);
```

**Listing 10.** passEqCheck: Program with password equivalence check

```
1  tagAsSink(console.log);
2  let pass = tagAsSource(pass1);
3  let confirm = tagAsSource(pass2);
4  function CheckPassword(pass, confirm){
5    if(pass != confirm){
6      return "Input values do not match";
7    }
8    return "Input values match";
9  }
10 console.log(CheckPassword(pass, confirm));
```

**Listing 11.** deviceIdCheck: Program with deviceID check

```
1  tagAsSink(fetch);
2  let cachedId = tagAsSource(cachedId);
3  let url = "http://server.com/api?ok";
4  navigator.mediaDevices
5    .enumerateDevices()
6    .then((devices) => {
7      devices.forEach((device) => {
8        let id = tagAsSource(device.deviceId);
9        if (id === cachedId)
10         fetch(url);
11     });
12   })
13   .catch((err) => {
14     console.log(`${err.name}: ${err.message}`);
15   });
16 }
```

| Benchmarks | | DTA++ | Dytan | GIFC | DPCA | |
|---|---|---|---|---|---|---|
| | | | | | PSC | sym, pc |
| Attack programs | countTo Trick | untainted (FN) | tainted | labeled | 1 | pc: (sec[0]>115 &&<br>sec[0]<=116)<br>sym: "t" |
| | encodeing Trick | tainted | tainted | labeled | 1 | pc: (sec[0]===116)<br>sym: "t" |
| | exception Trick | tainted | tainted | labeled | 1 | pc: (sec[0]===116)<br>sym: "t" |
| | lookupTable Trick | untainted (FN) | tainted | labeled | 1 | pc: (sec[0]===116)<br>sym: "t" |
| | test1 | tainted | tainted | labeled | 1 | pc:<br>sym: sec |
| | test2 | tainted | tainted | labeled | 1 | pc:<br>sym: sec |
| | test3 | tainted | tainted | labeled | 1 | pc:<br>sym: sec |
| | test4 | tainted | tainted | labeled | 1 | pc:<br>sym: sec |
| | test5 | tainted | tainted | labeled | 1 | pc<br>sym: sec[0] |
| Benign Programs | pass Autorization | tainted (FP) | tainted (FP) | labeled (FP) | $2^{64}$ | pc: (sec1[0]==sec2[0])<br>sym:<br>"http://hoge.com/api?fuga" |
| | passLenCheck | tainted (FP) | tainted (FP) | labeled (FP) | $2^{64}$ | pc: (sec.length >=8)<br>sym: " valid password !" |
| | passEqCheck | untainted | tainted (FP) | labeled (FP) | $2^{64}$ | pc: (sec1!=sec2)<br>sym: False |
| | deviceIdCheck | tainted (FP) | tainted (FP) | labeled (FP) | $2^{64}$ | pc: (sec1==sec2)<br>sym:<br>"http://hoge.com/api?fuga" |

**Figure 2.** Detection results. FN: False negative, which means that a secret source leaks to the public output. FP: False positive, which means that a benign program is incorrectly detected as an information leakage. sym, pc: symbol (sym) and path constraint (pc) observed at the sink point (only the key details are described here for simplicity).

The result shows that DPCA computes very large PSCs for the attack programs and small PSCs for the benign programs. Comparing the PSCs of the columns of DPCA, there is a large difference from 1 for the attack programs to $2^{64}$ for the benign programs. This means that DPCA is able to correctly detect whether secret information is leaked or not in all cases, no matter which threshold from 1 to $2^{64}$ was selected in this evaluation. As observed in this experiment, since there is a large difference in the computed PSC between attack programs and benign programs, it is desirable to set a threshold with a somewhat large PSC (e.g., $2^{32}$ in this experiment), taking safety into account, and use DPCA as a warning of danger. Note that the threshold of PSC to be detected as information leakage is not universal and should be considered depending on the nature of the target program and the importance of the source secret as described in Section 5.2.

We discuss the results of the attack programs and the benign program in detail below.

**Listing 12.** countToTrick: Program to leak information via for-loop

```
1  tagAsSink(console.log)
2  let pass  = tagAsSource('temp1234')
3  let out = "";
4  for (let i = 0; i < pass.length; i++) {
5    let y = 0;
6    let z = pass[i].charCodeAt();
7    for (let j=0; j<z; j++) {
8      y = y + 1;
9    }
10   out = out + String.fromCharCode(y);
11 }
12 console.log(out);
```

***Attack Programs.*** Attack programs (AntiTaintDroid and GIFC benchmarks) are programs that intend to leak the secret information in this evaluation. They send the source secret to the sink point via various implicit flows and aim to be leaked as public output. Therefore, information leak detection tools must correctly detect the public output of such programs as information leakage.

DPCA shows a very small PSC (=1) for all attack programs. The PSC of 1 means that observing the public output uniquely identifies the source secret, which is very dangerous because it is the same as the source secret being leaked to the outside world as it is. Therefore, DPCA successfully detects the public outputs with such small PSCs as information leaks.

Dytan and GIFC also succeeded in detecting attack programs. Since they propagate taints in all implicit flows, false negatives are unlikely to occur. On the other hand, DTA++ fails to detect attacks in some programs. This is because DTA++ selectively propagates taints in implicit flows, taking practicality into account.

For example, countToTrick is one of the attack programs. It uses a for-loop to leak secret data to the outside world. Listing 12 is a part of countToTrick. It converts the secret data into byte value and indirectly propagates the secret data to $y$ by going through the for-loop for the number of times of the value. This for-loop checks the conditional branch ($i < val$) to determine whether to exit the loop. However, as the possible values of this conditional expression are not strongly limited, the propagation rule of DTA++ does not allow taint propagation in the for-loop, which results in false negatives and allows information leakage. Since Dytan and GIFC propagate taints in all branches, they can detect information leakage in this program.

DPCA can accurately computes the PSC by tracking the conditional expression even in for-loop. Listing 12 converts the for-loop conditional expression (`j<z`) into an expression with only the source secret as a variable (`secret`) and stores it in path constraint (pc). For example, when `i=0` and `j=0`, the sym obtained from `j<z` is `{0<secret[0]}` and pc is `{0<secret.length}`. This is derived from the flow tracking rule of binary operations in DPCA, `{sym:(j.sym)<(z.sym)}`

and `{pc:(j.pc)&&(z.pc)}`. Since `z` extracts the index of 0 from `pass`, which is the source secret, at line 6, it has `{sym: secret[0]}`. As `j` is newly defined at line 7, it has no source secret in its sym (`{sym: 0}`). Therefore, the sym obtained from `(j<z)` is `{0<secret[0]}`. And from the tracking rule of if-statement, `{0<secret[0]}` is added to pc of `y` in line 8. By looping around to the end condition of the for-loop in this way, y finally contains `{pc: (0<secret.length && 0<secret[0] && 1<secret[0] && .... 115<secret[0] && 116>=secret[0])}`. Since the pc before and after the end condition of the loop `{115<secret[0] && 116>=secret[0]}` can be transformed into `{secret[0] ==116}`, DPCA computes the value satisfying this as 1 (PSC=1). Thus, DPCA accurately computes PSC by tracking sym and pc in attack programs, and succeeds in detecting information leakage.

***Benign Programs.*** Benign programs are generic web applications that are not intended to leak information in this evaluation. Although they use a source secret, the source secret cannot be recovered even if the public output sent to the sink point is observed. Therefore, information leak detection tools are required to detect that the public output of such programs is not information leakage and not to prevent the execution of the programs.

DPCA shows a very large PSC (=$2^{64}$) for all benign programs. This means that the source secret cannot be identified even if the public output is observed, in other words, no information leakage will occur. Therefore, DPCA does not detect public outputs with such a large PSC as information leakage and does not interfere with normal program execution.

Most of the existing methods generate false positives in benign programs. In particular, since Dytan and GIFC propagate taint in all implicit flows, if there is a conditional branch that touches the source secret even if it does not lead to information leakage, the taint is propagated to many of the subsequent processes, and false positives are likely to occur.

For example, passEqCheck shown in Listing 10 is a program that asks users to enter their passwords twice and checks their equivalence, which is common in web applications. This benchmark registers two passwords, `pass` and `confirm`, as source secrets, and displays their comparison results. Since the conditional expression, `(if (pass ! = confirm))` in line 5 touches source secrets, Dytan and GIFC propagate taint to the subsequent process. Then the string of the comparison result has taint and its outflow is detected as an information leakage. In reality, however, the source secret cannot be recovered from the comparison result because both source secret values are unverifiable by the observer. Therefore, the attack detection is false positive. Based on the propagation rules, as DTA++ does not propagate taint from a conditional expression of `if (pass ! = confirm)`, it does not cause a false positive in passEqCheck. However, it causes false positives in other benign programs. Its selective

**Table 3.** Performance Results (ms)

| Benchmarks | Baseline | DTA++ | Dytan | GIFC | DPCA |
|---|---|---|---|---|---|
| 24 | 43 | 1802 | 1294 | 1379 | 2953 |
| AES | 27 | 1773 | 1293 | 1307 | 3758 |
| FFT | 10 | 37 | 37 | 37 | 38 |
| FT | 3 | 31 | 29 | 30 | 33 |
| HN | 16 | 2404 | 2236 | 2259 | 5790 |
| KS | 22 | 1981 | 1871 | 1929 | 3611 |
| LZW | 13 | 85 | 74 | 80 | 99 |
| MD5 | 9 | 92 | 88 | 89 | 115 |

propagation method may not be sufficient for general web applications.

DPCA can accurately compute the PSC in such a program by tracking the conditional expressions. In line 5 of Listing 10, `(if (pass ! = confirm))`, since `pass` and `confirm` have different source secrets, the pc from which the string in line 6 is obtained is `{pc: (sec1 != sec2)}` (`sec1` is the symbol of `pass`, and `sec2` is the symbol of `confirm`). Applying this formula to the PSC computation program yields the following:

**Listing 13.** PSC computation for passEqCheck

```
1 let psc = 0;
2 for (let sec1 = 0; sec1 < 2^64; sec1++) {
3   for (let sec2 = 0; sec2 < 2^64; sec2++) {
4     if (sec1 === sec2)
5       psc++;  // psc = 2^64
6   }
7 }
```

DPCA can express with PSC that the source secret cannot be recovered from the result of the comparison if there are two source secrets. Therefore, it does not generate a false positive and does not interfere with normal program execution.

### 6.4 Performance

We conducted performance benchmarks to measure the impact of DPCA on the performance of the original application (the baseline) and compare it with DTA++, Dytan, and GIFC in this regard. We use 8 benchmarks that are used in [2, 39].

Table. 3 shows the execution time in milliseconds to run the performance benchmarks. The numbers are the average time of ten executions. Since both the existing method and DPCA are implemented on top of the code instrumentation tools, the insertion of instructions for the flow tracking has a performance impact. DPCA stores sym and pc in each variable for PSC computation and updates them for each instruction, and also computes PSC at the sink point, which requires 1.1x to 2.9x overhead compared to existing methods. In particular, AES has the highest PSC overhead due to its longer code length, more traps to be inserted, and the management of a large number of arrays.

However, the implementation of DPCA in this experiment is naive, and there is room for optimization. We also believe that a direct implementation in the language runtime, rather than in the code instrumentation tool, would significantly speed up the process. Speeding up DPCA is our future work.

## 7 Related Work

DTA is originally a method to prevent attacks that exploit program vulnerabilities such as SQL injection and cross-site scripting [10–12, 18, 22, 23, 25, 26, 28, 30–33, 44, 45, 47, 48, 50]. To detect unknown vulnerabilities in a program, DTA assigns the taint to untrusted external inputs and propagates the taint according to the data flow. We check whether the data is tainted in the parts that perform important operations that affect the system, such as the SQL engine and the input part of the file API. If the taint is found, it is assumed that an attack has been inserted from the outside, and the program can be stopped to prevent the attack.

DTA methods for information leakage prevention have been widely studied, but most of them do not consider the implicit flow [6, 7, 11, 14, 17, 21, 24, 34, 40, 46]. In addition to the aforementioned Dytan [10] and DTA++ [20], there are also PIFT [52] methods that consider taint propagation in implicit flows. PIFT proposes a lightweight DTA method to prevent information leakage in a mobile environment. PIFT focuses on the temporal locality from the time a value is loaded from memory to the time it is stored to efficiently track the data flow including implicit flows. However, PIFT has an issue in terms of tracking accuracy.

Especially due to the recent increase in injection attacks for web applications, many taint analysis frameworks for script languages are proposed [22, 25, 28, 45, 47, 48]. CSSE [32] and Phan [28] are taint analysis frameworks built on PHP to detect vulnerabilities in web applications. Vogt et al. [47] built a taint tracking system on top of a JavaScript VM to prevent cross-site scripting (XSS) attacks. JSBAF [48] is also a taint analysis framework for JavaScript that combines dynamic and static analysis to find vulnerabilities in web applications.

IFC (Information Flow Control) is a method of detecting vulnerabilities and preventing information leaks by analyzing information flow and enforcing compliance with defined information flow policies. Especially in JavaScript, dynamic IFC has been widely studied due to its dynamic nature [1, 3, 9, 16, 17, 37, 39]. Dynamic IFC assigns H and L (high and low) security levels to each variable and monitors how the security level of each variable changes to prevent the H level from being leaked to the outside world. Flow monitoring includes implicit flows. Several dynamic IFC techniques introduce a P (partially leaked) label in addition to H/L to monitor implicit flows. However, as we discussed before, dynamic IFC can cause false positives and stop the execution of safe programs that include implicit flows [3].

Quantitative Information Flow (QIF) is an analytical technique to calculate the amount of confidential information leaked in the output data of a system [4, 27, 35, 41]. QIF models the correspondence between the output which an attacker can observe and secret information as a communication channel in information theory. The method computes entropy to represent the amount of secret information in the output. The computation of the amount of information leaked from the program can be done automatically by generating its state transition diagram (discrete-time Markov chain). QIF can identify information leakage in a program, including implicit flow if the entire program is analyzed accurately. However, there are issues such as state explosion when calculating the state transition diagram for a large program. In addition, static information leakage which QIF deals with and dynamic leakage have different meanings as discussed in [4]. Under the assumption that an attacker has no prior knowledge of secret information and attempts to leak secret information by repeating many trials against the attack target, the static information leakage discussed in QIF is effective. On the other hand, this paper focuses on the case where an attacker identifies secret information in a program that is input by a good user, based only on the contents of the program and the observed values of the public output. In this case, the source secret is not a random value since a good user tries to input the correct secret information. Under these assumptions, it is more appropriate to consider dynamic leakage rather than static information leakage as assumed by QIF [4].

## 8 Conclusion

We introduced the PSC, a metric that quantifies how many secrets can be identified by observed data, and proposed DPCA which tracks this metric. DPCA assesses the PSC of data as it reaches a sink point, effectively distinguishing outputs that are practically secure from those that pose a security threat. We implemented DPCA on a JavaScript code instrumentation platform and demonstrated its capability to evaluate programs more accurately than existing DTA methods, which often fail to detect information leakage properly.

One of our future works is the optimization of DPCA for obtaining a PSC using model counting. Model counting, also known as Sharp-SAT, involves determining the number of variable combinations that satisfy a given boolean formula. In this context, PSC can be computed by applying model counting to a conditional formula composed of sym and pc. This approach is expected to be more comprehensive and efficient, particularly in handling complex conditions.

## Acknowledgement

This work was supported by JST PRESTO JPMJPR21P5.

## References


[1] Thomas Austin and Cormac Flanagan. 2012. Multiple Facets for Dynamic Information Flow. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Vol. 47. 165–178. https://doi.org/10.1145/2103621.2103677

[2] Abhishek Bichhawat, Vineet Rajani, Deepak Garg, and Christian Hammer. 2014. Information Flow Control in WebKit's JavaScript Bytecode. In *Principles of Security and Trust*, Martín Abadi and Steve Kremer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 159–178.

[3] Abhishek Bichhawat, Vineet Rajani, Deepak Garg, and Christian Hammer. 2021. Permissive runtime information flow control in the presence of exceptions. *Journal of Computer Security* 29 (03 2021), 1–41. https://doi.org/10.3233/JCS-211385

[4] Nataliia Bielova. 2016. Short Paper: Dynamic leakage: A Need for a New Quantitative Information Flow Measure. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security* (Vienna, Austria) *(PLAS '16)*. Association for Computing Machinery, New York, NY, USA, 83–88. https://doi.org/10.1145/2993600.2993607

[5] Lorenzo Cavallaro, Prateek Saxena, and R. Sekar. 2008. On the Limits of Information Flow Techniques for Malware Analysis and Containment. In *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Vol. 5137. 143–163.

[6] Quan Chen and Alexandros Kapravelos. 2018. Mystique: Uncovering Information Leakage from Browser Extensions. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1687–1700.

[7] Yu-Yuan Chen, Pramod A. Jamkhedkar, and Ruby B. Lee. 2012. A Software-Hardware Architecture for Self-Protecting Data. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. 14–27. https://doi.org/10.1145/2382196.2382201

[8] Laurent Christophe, Elisa Gonzalez Boix, Wolfgang De Meuter, and Coen De Roover. 2016. Linvail: A General-Purpose Platform for Shadow Execution of JavaScript. In *Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 260–270. https://doi.org/10.1109/SANER.2016.91

[9] Andrey Chudnov and David Naumann. 2015. Inlined Information Flow Monitoring for JavaScript. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. 629–643. https://doi.org/10.1145/2810103.2813684

[10] James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: A Generic Dynamic Taint Analysis Framework. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '07)*.

[11] Michael Dalton, Hari Kannan, and Christos Kozyrakis. 2007. Raksha: A Flexible Information Flow Architecture for Software Security. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '07)*.

[12] Ali Davanian, Zhenxiao Qi, Yu Qu, and Heng Yin. 2019. DECAF++: Elastic Whole-System Dynamic Taint Analysis. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. USENIX Association, 31–45.

[13] Leandro S. de Araújo, Leandro A. J. Marzulo, Tiago A. O. Alves, Felipe M. G. França, Israel Koren, and Sandip Kundu. 2020. Building a Portable Deeply-Nested Implicit Information Flow Tracking. In *Proceedings of the 17th ACM International Conference on Computing Frontiers (CF '20)*.

[14] Daniel Y. Deng, Daniel Lo, Greg Malysa, Skyler Schneider, and G. Edward Suh. 2010. Flexible and Efficient Instruction-Grained Run-Time Monitoring Using On-Chip Reconfigurable Fabric. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '43)*. 137–148.

[15] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. 2007. Dynamic Spyware Analysis. In *USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference (ATC'07)*. Article 18, 14 pages.

[16] Willem Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. 2012. FlowFox: A web browser with flexible and precise information flow control. *Proceedings of the ACM Conference on Computer and Communications Security*, 748–759. https://doi.org/10.1145/2382196.2382275

[17] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. 2014. JSFlow: Tracking Information Flow in JavaScript and Its APIs. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. 1663–1671.

[18] Andrew Henderson, Aravind Prakash, Lok Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. 2014. Make It Work, Make It Right, Make It Fast: Building a Platform-Neutral Whole-System Dynamic Binary Analysis Platform. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 248–258. https://doi.org/10.1145/2610384.2610407

[19] Byeongho Kang, TaeGuen Kim, BooJoong Kang, Eul Gyu Im, and Minsoo Ryu. 2014. TASEL: Dynamic Taint Analysis with Selective Control Dependency. In *Proceedings of the Conference on Research in Adaptive and Convergent Systems (RACS '14)*. 272–277. https://doi.org/10.1145/2663761.2664219

[20] Min Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. 2011. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *Proceedings of the Network and Distributed System Security Symposium (NDSS '11)*.

[21] Hari Kannan, Michael Dalton, and Christos Kozyrakis. 2009. Decoupling Dynamic Information Flow Tracking with a dedicated coprocessor. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems Networks*. 105–114. https://doi.org/10.1109/DSN.2009.5270347

[22] Rezwana Karim, Frank Tip, Alena Sochurkova, and Koushik Sen. 2020. Platform-Independent Dynamic Taint Analysis for JavaScript. *IEEE Transactions on Software Engineering* 46 (2020), 1364–1379.

[23] Jingfei Kong, Cliff C. Zou, and Huiyang Zhou. 2006. Improving Software Security via Runtime Instruction-Level Taint Checking. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability (ASID '06)*. 18–24. https://doi.org/10.1145/1181309.1181313

[24] Jinyong Lee, Ingoo Heo, Yongje Lee, and Yunheung Paek. 2015. Efficient Dynamic Information Flow Tracking on a Processor with Core Debug Interface. In *Proceedings of the 52nd Annual Design Automation Conference (DAC '15)*. Article 79, 6 pages. https://doi.org/10.1145/2744769.2744830

[25] Sebastian Lekies, Ben Stock, and Martin Johns. 2013. 25 Million Flows Later: Large-Scale Detection of DOM-Based XSS. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*. 1193–1204.

[26] K. Li, R. Shioya, M. Goshima, and S. Sakai. 2009. String-Wise Information Flow Tracking against Script Injection Attacks. In *Proceedings of the 15th IEEE Pacific Rim International Symposium on Dependable Computing*.

[27] Stephen McCamant and Michael D. Ernst. 2008. Quantitative Information Flow as Network Flow Capacity. *SIGPLAN Not.* 43, 6 (jun 2008), 193–205. https://doi.org/10.1145/1379022.1375606

[28] Mattia Monga, Roberto Paleari, and Emanuele Passerini. 2009. A hybrid analysis framework for detecting web application vulnerabilities. In *2009 ICSE Workshop on Software Engineering for Secure Systems*. 25–32. https://doi.org/10.1109/IWSESS.2009.5068455

[29] Andreas Moser, Christopher Kruegel, and Engin Kirda. 2007. Exploring Multiple Execution Paths for Malware Analysis. In *Proceedings of the IEEE Symposium on Security and Privacy (SP '07)*.

[30] James Newsome and Dawn Song. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS '05)*. https://doi.org/10.1184/R1/6468716.v1

[31] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. 2005. Automatically Hardening Web Applications Using Precise Tainting. In *Proceedings of IFIP International Information Security Conference*.

[32] Tadeusz Pietraszek and Chris Vanden Berghe. 2006. Defending Against Injection Attacks Through Context-Sensitive String Evaluation. In *Proceedings of Recent Advances in Intrusion Detection (RAID '05)*. 124–145.

[33] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. 2006. Argos: An Emulator for Fingerprinting Zero-Day Attacks for Advertised Honeypots with Automatic Signature Generation. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*. 15–27. https://doi.org/10.1145/1217935.1217938

[34] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. 2006. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*. 135–148.

[35] Seemanta Saha, Surendra Ghentiyala, Shihua Lu, Lucas Bang, and Tevfik Bultan. 2023. Obtaining Information Leakage Bounds via Approximate Model Counting. *Proc. ACM Program. Lang.* 7, PLDI, Article 167 (jun 2023), 22 pages.

[36] Golam Sarwar, Olivier Mehani, Roksana Boreli, and Mohamed Ali Kaafar. 2013. On the Effectiveness of Dynamic Taint Analysis for Protecting Against Private Information Leaks on Android-based Devices. In *10th International Conference on Security and Cryptography (SECRYPT '13)*.

[37] Bassam Sayed, Issa Traoré, and Amany Abdelhalim. 2018. IF-Transpiler: Inlining of Hybrid Flow-Sensitive Security Monitor for JavaScript. *Computers & Security* 75 (02 2018), 92–117. https://doi.org/10.1016/j.cose.2018.01.017

[38] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proceedings of IEEE Symposium on Security and Privacy*. 317–331. https://doi.org/10.1109/SP.2010.26

[39] Angel Scull, Laurent Christophe, Jens Nicolay, Coen De Roover, and Elisa Gonzalez Boix. 2018. Practical Information Flow Control for Web Applications. In *Proceedings of the International Conference on Runtime Verification*. 372–388. https://doi.org/10.1007/978-3-030-03769-7_21

[40] Asia Slowinska and Herbert Bos. 2009. Pointless Tainting? Evaluating the Practicality of Pointer Tainting. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys '09)*.

[41] Geoffrey Smith. 2009. On the Foundations of Quantitative Information Flow. In *Foundations of Software Science and Computation Structure*. https://api.semanticscholar.org/CorpusID:6680444

[42] Cristian-Alexandru Staicu, Daniel Schoepe, Musard Balliu, Michael Pradel, and Andrei Sabelfeld. 2019. An Empirical Study of Information Flows in Real-World JavaScript. In *Proceedings of the 14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security (PLAS'19)*. New York, NY, USA, 45–59. https://doi.org/10.1145/3338504.3357339

[43] Elizabeth Stinson and John C. Mitchell. 2007. Characterizing Bots' Remote Control Behavior. In *Proceedings of 4th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '07, Vol. 4579)*. 89–108. https://doi.org/10.1007/978-3-540-73614-1_6

[44] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. 2004. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*.

[45] Toshinori Usui, Yuto Otsuki, Yuhei Kawakoya, Makoto Iwamura, and Kanta Matsuura. 2022. Script Tainting Was Doomed From The Start (By Type Conversion): Converting Script Engines into Dynamic Taint Analysis Frameworks. In *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*. 380–394.

[46] Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic. 2008. FlexiTaint: A programmable accelerator for dynamic taint propagation. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture (HPCA '08)*.

[47] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Krügel, and Giovanni Vigna. 2007. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis.

[48] Shiyi Wei and Barbara G. Ryder. 2013. Practical Blended Taint Analysis for JavaScript. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. 336–346. https://doi.org/10.1145/2483760.2483788

[49] Heng Yin, Zhenkai Liang, and Dawn Song. 2008. HookFinder: Identifying and Understanding Malware Hooking Behaviors. In *Proceedings of the Network and Distributed System Security (NDSS '08)*.

[50] Heng Yin and Dawn Song. 2010. *TEMU: Binary Code Analysis via Whole-System Layered Annotative Execution*. Technical Report UCB/EECS-2010-3. EECS Department, University of California, Berkeley.

[51] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. 2007. Panorama: Capturing System-Wide Information Flow for Malware Detection and Analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*. 116–127. https://doi.org/10.1145/1315245.1315261

[52] Man-Ki Yoon, Negin Salajegheh, Yin Chen, and Mihai Christodorescu. 2016. PIFT: Predictive Information-Flow Tracking. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*.


Received 2024-05-25; accepted 2024-06-24

# The Cost of Profiling in the HotSpot Virtual Machine

Rene Mueller
Huawei Zurich Research Center
Zurich, Switzerland
rene.mueller@huawei.com

Maria Carpen-Amarie
Huawei Zurich Research Center
Zurich, Switzerland
maria.carpen.amarie@huawei.com

Matvii Aslandukov*
Kharkiv National University of Radio Electronics
Kharkiv, Ukraine
matvii.aslandukov@nure.ua

Konstantinos Tovletoglou*
Independent Researcher
Zurich, Switzerland
ktovletoglou01@qub.ac.uk

## Abstract

Modern language runtimes use just-in-time compilation to execute applications natively. Typically, multiple compiler tiers cooperate so that compilation at a later stage can leverage profiling information generated by earlier tiers. This allows for machine code that is optimized to the actual workload and hardware. In this work, we study the profiling overhead caused by code instrumentation in the HotSpot Java virtual machine for 23 applications from the Renaissance suite and five additional benchmarks.

Our study confirms two common assumptions. First, most applications move quickly through the profiling phase. However, we also show applications that tier up surprisingly slowly and, thus, are more affected by profiling overheads. We find that the instrumentation needed for profiling can slow application execution down by up to 35×. A key factor is the memory contention on the shared profiling data structures in multi-threaded applications. Second, most virtual call sites are monomorphic, i.e., they only have a single receiver type. This can reduce the run-time cost of otherwise expensive receiver type profiling at virtual call sites.

Our analysis suggests that, for the most part, profiling overhead in language runtimes is not a cause for concern. However, we show that there are situations, e.g., in multi-threaded applications, where profiling impact can be consequential.

***CCS Concepts:*** • **Software and its engineering → Just-in-time compilers**; **Runtime environments**.

*Work done while at Huawei Zurich Research Center, Switzerland.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*MPLR '24, September 19, 2024, Vienna, Austria*
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1118-3/24/09
https://doi.org/10.1145/3679007.3685055

***Keywords:*** Java virtual machine, profiling, JIT compiler

**ACM Reference Format:**
Rene Mueller, Maria Carpen-Amarie, Matvii Aslandukov, and Konstantinos Tovletoglou. 2024. The Cost of Profiling in the HotSpot Virtual Machine. In *Proceedings of the 21st ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '24), September 19, 2024, Vienna, Austria.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3679007.3685055

## 1 Introduction

Modern language runtimes with just-in-time (JIT) compilers are very popular for managed languages such as Java, C#, JavaScript, etc. Their advantage over ahead-of-time (AOT) compilation is their ability to leverage run-time information about the actual workload for additional compiler optimizations. The downside of the JIT compilation, however, is not only the delay of the compilation effort into the execution phase, but also the overhead from collecting the profiling information.

In this work, we quantify the profiling overhead of the HotSpot Java virtual machine (JVM) in the OpenJDK. We observe that the profiling overhead introduced by code instrumentation can cause an up to 35-fold application slowdown. Fortunately, applications with a pronounced hotspot move quickly through the profiling phase. This reduces the observable impact of profiling on the run-time or throughput of applications. HotSpot, thus, does live up to its name. However, we also find application code from popular benchmarks that tiers up more slowly and remains longer in the profiling phase. This poses the question: why do some application move faster through the profiling phase than others?

***Contributions.*** We describe the code instrumentation used in HotSpot to collect profiling information and evaluate the resulting run-time overhead across the Renaissance suite [19] and five of our own benchmarks. Specifically, we measure the run-time cost from the code instrumentation used to collect the profiling data and find that profiling can slow down applications by up to 35×. We show that this slowdown can be caused by application threads that concurrently update shared data structures that hold the profiling information.

Of all profiling instrumentation, the code used to collect receiver type profiles, e.g., the different receiver types encountered in virtual method calls, has the largest footprint and the largest number of branches. The effective run-time overhead depends on the number of encountered types (degree of polymorphism). Based on the analysis of the aforementioned workloads, we confirm the common assumption for object-oriented programs that most virtual call sites are monomorphic. This can reduce the effective cost in 80 % of the cases in which receiver type profiles are collected.

## 2 Background

The HotSpot JVM executes Java bytecode, which is the instruction set for the abstract stack machine defined by the Java Virtual Machine specification. The instruction set and its execution engine were originally designed to execute Java programs. Today, however, the Java bytecode is used as an intermediate representation format for many other languages besides Java, such as Scala, Kotlin, Ruby (JRuby), Groovy, Clojure etc. Therefore, we do not limit this study to Java applications and, instead, consider Java bytecode as the workload for the JVM. The Java bytecode and JIT compilation in HotSpot JVM are designed to support the *open-world assumption* of Java in which new classes can be loaded or existing classes redefined at run-time [14], e.g., by modifying or adding methods. After such a bytecode change, the JIT compiler can perform on-the-fly optimizations across methods and classes that go beyond the traditional dynamic link/loading and which would be difficult to implement using ahead-of-time compilation approaches.

***Multi-tier Compilation.*** HotSpot is a complex language runtime that consists of an interpreter and two method-based JIT compilers, called **C1** and **C2** (see Figure 1), which are further organized in four compilation tiers. C1 is the simpler of the two compilers and generates code more quickly, albeit with fewer optimizations. In addition, the application start-up time is further reduced by not compiling all the bytecode at once. Instead, methods are first executed by the bytecode interpreter. They subsequently move through the compiler tiers depending on their "hotness", essentially, the time application threads spend in these methods. Hotness is quantified by two metrics: (1) the number of times a given method is invoked and (2) how often the loops in the method are executed, i.e., the number of times a back-edge of any loop in the method is taken.

***Bytecode Profiling.*** As the application's bytecode code moves through the compiler tiers, i.e., the code tiers up, profiling data is collected through code instrumentation added by the C1 compiler (in Tiers 2 and 3) or during execution in the interpreter (Tier 0). The C1 compiler can add code instrumentation to different degrees, ranging from no-instrumentation (Tier 1), only counting method invocations and loop trips in Tier 2, to full profiling of a number of selected bytecode instructions (listed in Table 1) in Tier 3. When the instrumented code is executed or the application runs sufficiently long in the interpreter, profiling information is collected and written into *Method Data Objects (MDOs)*. These objects are maintained by the JVM, each containing profiling data for a specific method.

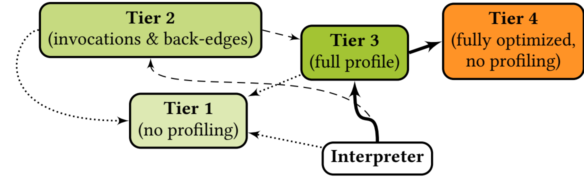



**Figure 1.** Multi-tiered compilation in HotSpot

The collected profiling information is made available to the C2 compiler (Tier 4) and enables it to generate more efficient code. For example, the C2 compiler uses the frequency of taken and non-taken paths of branches to determine the layout of basic blocks, makes inlining decisions for virtual method calls based on the actually encountered receiver types, or omits the generation of machine code for paths that were never taken during profiling. Many of these optimizations are speculative. Thus, in order to guarantee correct execution, the C2 compiler emits run-time checks that guard the speculatively generated code. If such a check fails, the execution resumes in the interpreter while the bytecode is recompiled with the earlier-made speculation adjusted accordingly. Such an occurrence is called a *deoptimization event*. The execution exits the interpreter and resumes in compiled code once the JIT compilers make the new code available. The application code will eventually stabilize without further deoptimizations in the highest tier (Tier 4). Once fully tiered up, no further profiling data is collected in order to eliminate the run-time overhead from code instrumentation.

Figure 1 shows different possible paths a method can take through the tiers. The common path for a method (highlighted in bold in the figure) is to move from the interpreter to the fully-profiling Tier 3 and subsequently to Tier 4. Trivial methods that are eventually inlined by C2, e.g., getters and setters, take a different path (shown as dotted lines in Figure 1). Furthermore, if the load on C2 compiler threads is high and the queue of compilation jobs for C2 becomes long, a back-pressure path is introduced (shown as dashed lines) that temporarily slows down the tiering-up until the pressure on the C2 compiler threads has eased up.

In this study, we analyze the overhead of the different types of instrumentation that can be added by the C1 compiler. The instrumentation covers the bytecode instructions listed in Table 1 but also includes method-level information such as the number of method invocations and loop back-edges taken in any loop of the method. This data is written into the MDO, which is organized as an array of 8-byte slots.

```
1 ; e.g., invokestatic
2 movabs MDO_addr,%rsi
3 addq   $1,count_off(%rsi)
4 callq  static_target_method
5 ...
```

**(a)** Count Profile

```
1 ; e.g., ifeq
2      movabs MDO_addr,%rax
3      movabs taken_off,%rsi
4      je     .l1
5      movabs not_taken_off,%rsi
6 .l1: mov    (%rax,%rsi,1),%rdi
7      lea    1(%rdi),%rdi
8      mov    %rdi,(%rax,%rsi,1)
9      je     .l2
10     ...
11 .l2: ; actual target
```

**(b)** Branch Profile

```
1 ; in method preamble
2 movabs MDO_addr,%rsi
3 mov    invoc_cnt_off(%rsi),%edi
4 add    $2,%edi
5 mov    %edi,invoc_cnt_off(%rsi)
6 and    overflow_mask,%edi
7 cmp    $0,%edi
8 je     counter_overflow
9 ...
```

**(c)** Invocation Count

**Figure 2.** x64 assembly code for count and branch profiles, and the invocation counter in the method preamble

**Table 1.** Profile types in Method Data Objects (MDO) and profiled bytecode instructions

| Profile Type | Bytecode Instructions |
|---:|---|
| Counter Profile | invokestatic, invokespecial, invokedynamic |
| Jump Profile | goto, goto_w |
| Branch Profile | ifeq, ifge, ifle, ifne, ifnonnull, ... |
| Multi-branch Profile | tableswitch, lookupswitch |
| Receiver Type Profile | aastore, checkcast, instanceof |
| Virtual Call Profile | invokeinterface, invokevirtual |
| Invocation Count | *per-method counter* |
| Back-edge Count | *per-method counter* |

***Profiling Instrumentation in x64.*** Figure 2 shows three examples of the code instrumentation created by the C1 compiler in x64 assembly. In each of the examples, one or more slots in the MDO are updated. Each slot is referenced through an offset from the MDO's base address. Observe the absence of atomic instructions. Thus, the counter updates are not thread-safe. Updates can be lost if counters are updated by multiple threads concurrently.

Figure 2a shows the *Count Profile* used in invokestatic (for static methods) or invokespecial (for Java constructors). It simply increments the count slot in the MDO that corresponds to the profiled bytecode instruction. The code template to collect *Branch Profiles* is shown in Figure 2b. A branch profile contains two slots, which count how often the branch was taken or not taken. The *Invocation Count* of a method is incremented in the method preamble (see Figure 2c). The snippet does not only increment the counter[1] but also checks for overflows using the overflow_mask. This masks out the least significant ten bits of the counter, causing an "overflow" every 1,024 invocations. Each time such an overflow occurs, a call into the runtime is made to notify it that the method may move up to the next tier. The *Receiver Type Profile*, e.g., used in virtual calls (invokevirtual), has one of the most complex templates. It counts the occurrence of different data types of receiver objects for a method call at the call-site. On x64, the corresponding code template takes 150 bytes and 23 CPU instructions.

[1]The add 2 is correct for the increment operation because the least significant bit in the count is used as a "sticky" carry bit.

```
1 public void render(int[] image) {
2   int width = camera.getWidthPixels();
3   int height = camera.getHeightPixels();
4   for (int y = 0; y < height; y++) {
5     for (int x = 0; x < width; x++) {
6       Ray ray = camera.rayThroughPixel(x, y);
7       Object3D closest = null;
8       double min = Double.POSITIVE_INFINITY;
9       for (Object3D o : objects) {
10        double dist = o.intersect(ray);
11        if ((dist > 0) && (dist < min)) {
12          min = dist; closest = o; }
13      }
14      Color pixel = backgroundColor;
15      if (closest != null)
16        for (Light l : lights)
17          pixel = pixel.blend(
18            closest.computeColor(camera, l));
19      image[y * width + x] = pixel.asRGBInt();
20    }
21  }
22 }
```

**Listing 1.** render method of object-oriented Java raytracer

## 3 Motivation

The instrumentation by C1 increases the size of the generated machine code by a non-trivial amount. We illustrate this on a *raytracer* application.

### 3.1 Raytracer Application

Listing 1 shows the Java source code of the raytracer's render method that iterates over pixels of the image plane, shoots rays from the camera location through the pixels into

**Table 2.** Code footprint of the render method in number of x64 instructions and the duration of the method call

| | Profiling | | x64 | |
|---|---|---|---|---|
| **Tier (Compiler)** | Invocations, Back-edges | Bytecode Instr. | # CPU Instr. | Run Time |
| 1 (C1) | | | 306 | 2.4 s |
| 2 (C1) | ✓ | | 380 | 2.6 s |
| 3 (C1) | ✓ | ✓ | 709 | 3.8 s |
| no inline 4 (C2) | | | 550 | 1.0 s |
| 4 (C2) | | | 1,255 | 0.9 s |

the scene, checks for intersections of the rays with objects, and determines the color of the pixels. We choose this application for illustration because it has a number of interesting properties. First, it uses subtype polymorphism, which allows optimization for Java's virtual method dispatch at run-time (`invokevirtual`). Every scene object type, such as `Sphere`, `Box`, `Triangle`, etc., is a subclass of the abstract base class `Object3D` and comes with its own implementation of an `intersect(Ray)` method that determines whether a given ray intersects with an object of that particular type. Second, the `render` method uses Java `Iterable` and `Iterator` objects in for-each loops to iterate over the collection objects and lights in the scene. These iterators are controlled through `invokeinterface` bytecode instructions. For both invocations, the receiver types are profiled in Tier 3. Third, `render` also has several nested loops, allowing it to tier up quickly, which simplifies the analysis. Finally, the method also permits speculative optimizations on the control flow, e.g., omitting the then-branches (lines 12 and 16) if the rays did not intersect with any object during profiling.

Table 2 shows the size of full-method compilations of the `render` method by each of the four compilation tiers as the number of generated machine instructions for x64. Note that two different compilers are used: the base compiler C1 and the optimizing compiler C2. The C1 compilations shown in Table 2 differ only in the number of instructions emitted for profiling. Counting method invocations and back-edges in Tier 2 increases the code footprint from 306 in Tier 1 to 380 instructions (+24 %). The profiling of bytecode instructions in Tier 3 further increases code footprint to 709 instructions (2.3× baseline C1). For C2 (Tier 4), we show two variants in Table 2. For the first, we disable the aggressive inlining of method calls by the C2 compiler. This permits a direct comparison with C1 (Tier 1) as in both cases only the trivial getter methods in lines 2 and 3 (Listing 1) as well as the call to `iterator` in the for-each statements in lines 9 and 16 are inlined. The second variant shows the default behavior of HotSpot in Tier 4 with full inlining allowed. The scene on which we invoke `render`, consists of only one concrete type of `Object3D`, specifically, `Sphere`. The call sites in `render` are therefore *monomorphic* and, if permitted, C2 then not only inlines *all* calls in `render` but also the nested calls. The inlining of calls explains the significant increase of the method's footprint by 4.1×.

Table 2 also shows duration of the call to the `render` method after it is fully compiled to machine code. The table shows that C1 code with invocation and back-edge counting (Tier 2) takes 8 % longer than without any profiling, increasing the call duration from 2.4 s to 2.6 s. Full instruction profiling increases the duration of the `render` call further to 3.8 s. Thus, *full profiling for the raytracer increases the run-time by 58 %* compared to code created by the same compiler without any profiling instrumentation. Switching from C1 to the optimizing C2 compiler and using the collected profiling information results in a speedup of 2.4×. Allowing aggressive inlining in C2 increases the speedup over Tier 1 to 2.6×.

### 3.2 Slow Tier-Up

The run-time overhead for the application that is caused by code running in the profiling tiers can be substantial, as shown in Table 2 for the raytracer application (+58 %). However, it is often argued that this overhead is negligible for applications that spend most of the time in a small well-contained range of bytecode instructions, such as "hot" methods or "hot" loops, in particular, if they run sufficiently long to amortize the tier-up overhead. Such code hotspots tier up quickly and consequently spend little time in the full-profiling Tier 3 where they encounter the highest overhead.

Figure 3 (left) shows the tier-up behavior of the single-threaded raytracer application running on an Intel Xeon W-1270 system with eight cores. The area plot shows which type of code is executed by the application during the 3.3 s run-time. We sample the application at a fixed frequency using the Linux `perf` utility. For each sample reported by `perf`, we look up the type of code that was under the program counter (PC). Since all code besides the VM itself is dynamically generated[2], we implement a JVM Tooling Interface (JVMTI) agent that gets notified when the VM has generated new code. This allows us to map the PC from the samples to the symbol names of the generated codelets. We also extend the JVMTI API such that each notification of the agent also carries the number of compilation tier (1,...,4) that generated it, such that we can map PC samples from `perf` back to compiler tiers. The tier breakdown shows how the applications move through the compilation tiers.

The raytracer, with its well-defined hotspot comprised of the loop-nest in the `render` method, tiers up quickly. Most of the application time is spent executing code created by the highest compilation tier (Tier 4). The fraction spent in the fully-profiling Tier 3, which experiences the highest run-time overhead, is shown as the red area in Figure 3. It only

[2]Dynamically generated code also includes the interpreter. The codelets that correspond to the bytecode instructions executed in the interpreter are generated during the start-up of the HotSpot JVM.

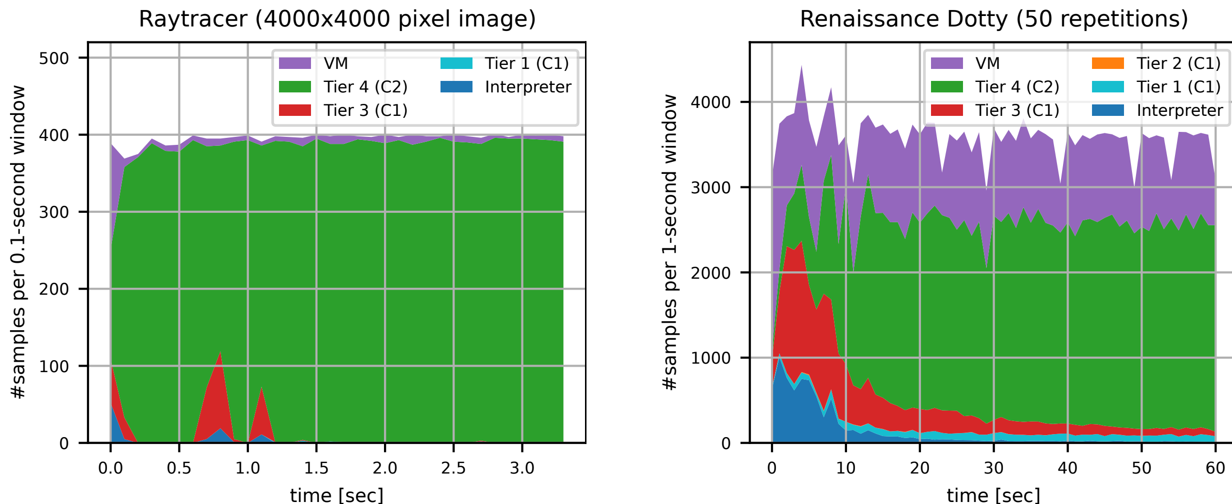



**Figure 3.** Two applications with different tier-up behavior. Left: raytracer tiering up quickly, Right: dotty from Renaissance with a slow tier-up over the 50 default benchmark repetitions

appears at the beginning of the application. The "spikes" in Tier 3 activity are due to exits from Tier 4 code, either when a loop of a partially compiled method ends or a deoptimization event occurs that is caused by a failed speculative optimization. The figure also shows that the application spends a short time in the interpreter (blue).

However, not all applications have a pronounced hotspot like the raytracer which would allow them to tier up quickly. In Figure 3 (right) we show the tier-up behavior of one such application from the Renaissance suite [19]. The benchmark is *dotty*, which runs the Dotty compiler to compile a number of Scala source files for the `scalap` utility. The area plot shows the tier distribution of the application thread for 50 repetitions of the dotty benchmark, the default number for this benchmark. The red Tier 3 area, although gradually shrinking over time, does not reach zero even after a minute of repetitions of the benchmark. The reason is the large code base with over 20,000 touched unique methods and over 27,000 compilations during this one-minute run.

The large number of compilations also strain the C1 and C2 compiler threads which in turn compete with application threads for system resources. However, the cost of the dynamic compilation itself is outside of the scope of this study as it is orthogonal to the run-time overhead that is incurred from the profiling instrumentation. In this work, we quantify the impact this profiling has on the observable application performance. The effective impact is determined by two factors. The first factor is the slowdown of the instrumented machine code that is generated by C1 in Tiers 2 and 3 over non-instrumented C1 code (Tier 1). The second factor is represented by the fraction of time the application spends executing instrumented code relative to the fraction it executes non-instrumented code (Tier 1 and Tier 4). In the next sections, we provide an analysis of both factors on a standard JVM benchmark suite and a number of other applications.

**Table 3.** Additional applications in the workload study

| Application | Lang. | Description |
|---:|:---:|---|
| Raytracer-Java | Java | Raytracer, 4-object scene |
| Raytracer-JRuby | Ruby | Raytracer, 4-object scene |
| Raytracer-Clojure | Clojure | Raytracer, 4-object scene |
| SparkSQL-TPC-H | Scala | 22 queries at scale factor 1 |
| H2O-GBM | Java | H2O inference on a GBM |

## 4 Profiling Overheads

In this section, we present a quantitative analysis of profiling overheads in Renaissance [19], which is a modern benchmark suite for the JVM. We also include the five additional applications listed in Table 3 in our analysis.

***Raytracer-*.*** We implement the simple raytracer from Section 3.1 in three languages that can be executed on the JVM: Java running directly on the JVM, Ruby on JRuby, and Clojure. The additional implementations in JRuby and Clojure allow us to study the effect of stacking language runtimes on top of the HotSpot JVM, specifically, running a dynamically typed code on top of a runtime that uses statically typed bytecode. All raytracers render the same scene that contains four objects, each of a different type. The `render` method of the Java raytracer is shown in Listing 1. By using different object type, the call sites on lines 10 and 18 are polymorphic and the receiver types have to be profiled. We compile the Clojure code ahead-of-time to Java bytecode in order to avoid the start-up overhead of the on-the-fly generation of bytecode by the Clojure runtime. JRuby uses its own interpreter and JIT compiler that creates Java bytecode

on-the-fly. For the JRuby JIT compiler, we enable the use of `invokedynamic` instructions for Ruby call-sites. This provides a 2.5× speedup when fully tiered up but adds to the profiling cost due the increased use of method handles that are translated in lambda-forms for which the JVM creates bytecode dynamically (JEP-160 [21]). This additional bytecode contains more profiled instructions and may further also collect argument and return types in calls (JSR-292 [9]).

***SparkSQL-TPC-H.*** We run the 22 SQL queries from the TPC-H decision support database benchmark on top of SparkSQL (Spark 3.3) with a Parquet data set at scale factor 1. We do not use any explicit data cache except the buffer cache of the file system. We execute the queries back to back in a single benchmark run. We discard the measurements of the first run when the buffer cache is cold. We include this application because of the dynamically generated Java code from SparkSQL's full-stage code generation and its large code footprint overall.

***H2O-GBM.*** We train a gradient-boosted decision tree model (GBM) on the airline dataset [25] using H2O-2, an in-memory Java machine learning platform. We export the trained model as Java code. Each decision tree is implemented as a method with one or more conditional branches on the selected attributes. We use this trained model in the benchmark in an inference configuration on a subset of the training set. We add this benchmark due to its large number of branches.

### 4.1 System Setup

We execute all benchmarks on an Intel Xeon W-1270 system with eight cores and 64 GiB of DDR4 memory (we provide an analysis for an Arm system in Appendix A). The operating system is Ubuntu 22.04.2 LTS. We run a release build of the OpenJDK 17.0.5 and unless otherwise noted, we use the configuration set by JVM Ergonomics[3], e.g., heap size, garbage collector, and number of threads for the garbage collector and C1 and C2 compilers. On our system, JVM Ergonomics limits the threads used for the C1 and C2 compilers to four threads each. This thread count is only used during times of the highest compiler load. We use the G1 garbage collector, as chosen automatically by the JVM, for all experiments. We control the tier-up behavior shown in Figure 1 (common case: Interpreter → Tier 3 → Tier 4) with the `XX:TieredStopAtLevel=L` VM option which prevents HotSpot from tiering-up beyond level `L` $= 1, \ldots, 4$.

### 4.2 Slowdown from Profiling

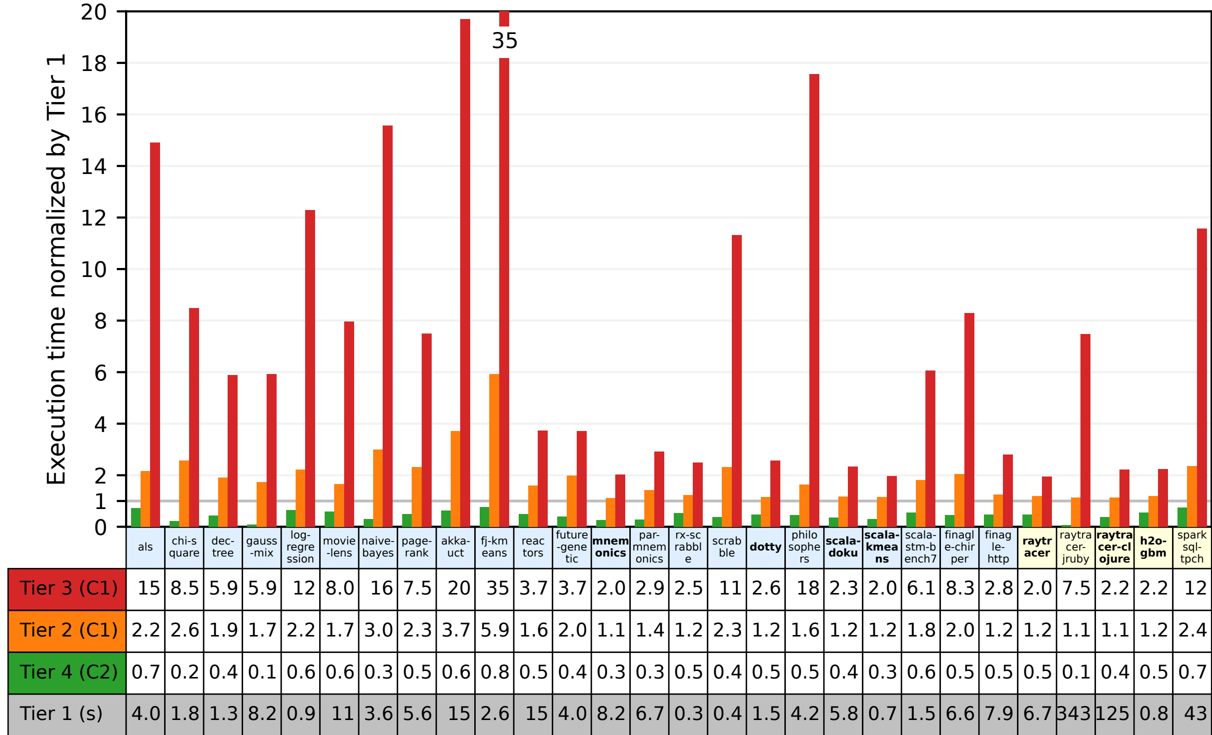
Figure 4 shows the increase of execution time of the 28 JVM applications caused by profiling overheads in Tiers 2 and 3. As the figure shows, full profiling can increase the execution time by up to 35× over C1 code with no profiling when we prevent the application to fully tier-up into Tier 4. The geometric mean of Tier 3 slowdown across the workloads is 5.7×. The profiling of method invocations and back-edges in Tier 2 can have a 6× overhead over Tier 1 (geometric mean 1.8×). For comparison, the fully optimizing C2 compiler provides a speedup of 2.4× (geometric mean) over non-instrumented C1 code, with the highest speedup of 10× for gauss-mix and raytracer-jruby.

### 4.3 Cache Contention Analysis

In the following, we analyze the behavior of the benchmark application with the highest profiling overhead, fj-kmeans (35×). This benchmark performs a parallel k-means clustering using Java's `ForkJoinPool`. On our 8-core system, this pool contains 16 worker threads. The program partitions 500,000 random data points into five clusters. This is achieved using the traditional k-means algorithm. The implementation consists of two parallel phases executed in fork-join manner, (1) assignment of each data point to the nearest cluster centroid, (2) updating the location of the centroids. The two phases are repeated 50 times. Even though the algorithm itself is simple, the implementation using the fork-join paradigm adds additional complexity due to the dynamic splitting of tasks into sub-tasks based on the amount of work of a task, i.e., number of data points and centroids. The number of data points and clusters is very small and, thus, the two parallel phases during which the worker threads run independently is relatively short. The frequent synchronization between phases effectively causes the worker threads to execute the same code at the same time. This is not a problem for Tier 1 code. In Tier 3, however, the tightly synchronized worker threads update the same MDO slots nearly at the same time. The resulting contention on MDO slots has a significant performance impact. We note that this performance impact occurs even though HotSpot does not prevent data races during counter accesses, which could be prevented using more expensive atomic operations such as compare-and-swap instructions. The cause is the cache coherence protocol. The cache line that contains the updated MDO slot will be marked as modified in the core's private cache, which also invalidates any copies that other cores may have in their private caches. Thus, when one of the other cores then tries to access the same MDO slot, the access will miss in the core's local cache. The coherence protocol resolves this miss by moving the modified cache line to private cache of the requesting core.

We quantify the impact of the contention on the MDO by measuring coherence traffic using the Linux `perf` tool as we vary the number of threads used by fj-kmeans. We focus on the snoop messages, i.e., messages that are exchanged to probe the state of a cache line. Specifically, we count the number of snoop responses in which the line was found in

[3]For dotty and SparkSQL-TPC-H, we need to increase the size of the JVM's code cache to 512 MB in order to run the application exclusively in Tier 3, due to the increase of the code-footprint from the profiling instrumentation.




| | als | chi-square | dec-tree | gauss-mix | log-regression | movie-lens | naive-bayes | page-rank | akka-uct | fj-kmeans | reactors | future-genetic | **mnemonics** | par-mnemonics |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Tier 3 (C1) | 15 | 8.5 | 5.9 | 5.9 | 12 | 8.0 | 16 | 7.5 | 20 | 35 | 3.7 | 3.7 | 2.0 | 2.9 |
| Tier 2 (C1) | 2.2 | 2.6 | 1.9 | 1.7 | 2.2 | 1.7 | 3.0 | 2.3 | 3.7 | 5.9 | 1.6 | 2.0 | 1.1 | 1.4 |
| Tier 4 (C2) | 0.7 | 0.2 | 0.4 | 0.1 | 0.6 | 0.6 | 0.3 | 0.5 | 0.6 | 0.8 | 0.5 | 0.4 | 0.3 | 0.3 |
| Tier 1 (s) | 4.0 | 1.8 | 1.3 | 8.2 | 0.9 | 11 | 3.6 | 5.6 | 15 | 2.6 | 15 | 4.0 | 8.2 | 6.7 |

| | rx-scrabble | scrabble | **dotty** | philosophers | **scala-doku** | **scala-kmeans** | scala-stm-bench7 | finagle-chirper | finagle-http | **raytracer** | raytracer-jruby | **raytracer-clojure** | **h2o-gbm** | spark-sql-tpch |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Tier 3 (C1) | 2.5 | 11 | 2.6 | 18 | 2.3 | 2.0 | 6.1 | 8.3 | 2.8 | 2.0 | 7.5 | 2.2 | 2.2 | 12 |
| Tier 2 (C1) | 1.2 | 2.3 | 1.2 | 1.6 | 1.2 | 1.2 | 1.8 | 2.0 | 1.2 | 1.2 | 1.1 | 1.1 | 1.2 | 2.4 |
| Tier 4 (C2) | 0.5 | 0.4 | 0.5 | 0.5 | 0.4 | 0.3 | 0.6 | 0.5 | 0.5 | 0.5 | 0.1 | 0.4 | 0.5 | 0.7 |
| Tier 1 (s) | 0.3 | 0.4 | 1.5 | 4.2 | 5.8 | 0.7 | 1.5 | 6.6 | 7.9 | 6.7 | 343 | 125 | 0.8 | 43 |

**Figure 4.** Profiling overheads in Tier 2 and Tier 3 relative to Tier 1 for the Renaissance suite and our additional benchmarks. Benchmarks in **bold** are single-threaded.

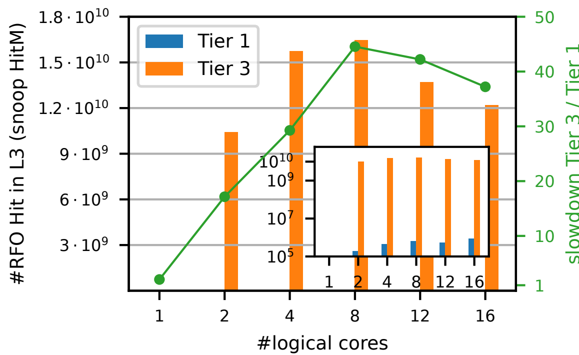



**Figure 5.** The number of read-for-ownership (RFO) requests that hit in L3 while a higher-level cache has the cache line in modified state correlates with the slowdown from Tier 3 profiling. Shown for fj-kmeans for different number of logical cores made available to the JVM.

modified state (snoop HitM). [4] We show the number of snoop HitM for both Tier 1 and Tier 3 in Figure 5 along with the resulting slowdown. We control the number of threads, thus, the degree of contention, by varying the number of logical cores available to the JVM. We restrict the JVM process to subsets of logical cores using the `taskset` utility. Our Intel system has 16 logical cores, i.e., eight physical cores each capable of running two hardware threads. We increase the number of logical cores by starting with the first hardware thread on the physical cores from 0 to 7. Logical cores 8 to 15 then use the second hardware thread of the physical cores. Figure 5 shows that the L3 snoop HitM counts for Tier 3 code are about four orders of magnitude higher than for Tier 1. We use a logarithmic scale for the inset figure for better comparison with the Tier 1 counts. The slowdown in execution time (right y-axis) follows roughly the number of snoop HitM. The worst-case slowdown from profiling is 45× if each of the eight physical core runs one application thread. The number of snoop HitM decreases once the second hardware thread of a physical core is used. The reason is that the threads that run simultaneously on the same core share private caches, in which case no snoop is triggered. This reduces the snoop traffic even though the work is distributed over more threads.

Figures 6 (left) and (right) compare the Tier 3/Tier 1 ratio for snoop HitM and the slowdown for all benchmarks. We distinguish between single- and multi-threaded benchmarks. The figures show that a high slowdown tends to correlate

[4] We measure the L3 snoop hits by counting the `offcore_response.demand_rfo.l3_hit.snoop_hitm` events.

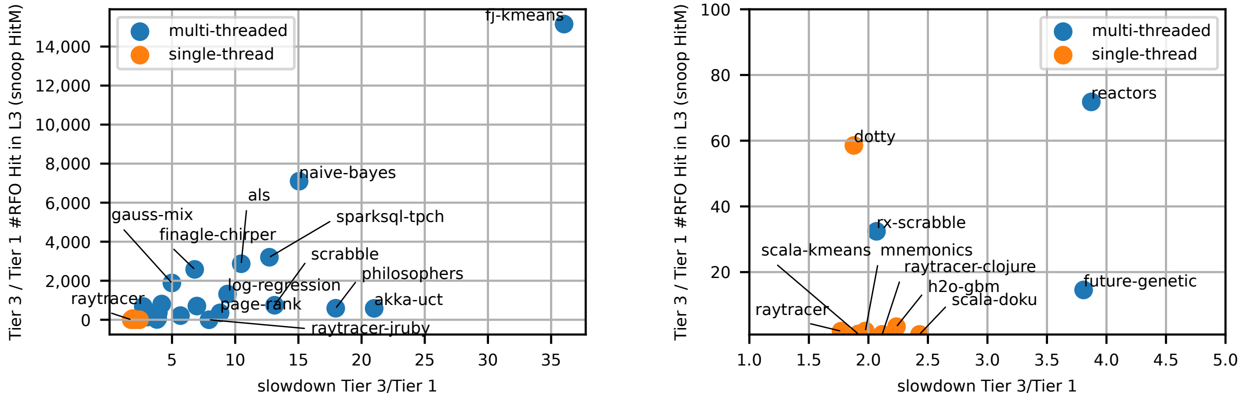



**Figure 6.** The number of read-for-ownership (RFO) requests that hit a cache line in the modified state in L3 correlates with the slowdown from Tier 3 profiling. Left: all benchmarks, right: zoomed-in.

with a high ratio of HitM snoops. Furthermore, the single-threaded benchmarks have an extremely low HitM ratio.

We can confirm this by restricting the number of cores made available to the JVM. Using the `taskset` utility, we limit and pin the JVM to one single logical core. This also determines the number of application threads spawned by the benchmarks. As expected, the Tier 3/Tier 1 slowdown is substantially reduced. When limited to one logical core, the profiling slowdown in akka-uct is reduced from 20× to 2.2×, philosophers from 18× down to 2.4×, naive-bayes from 15× down to 2.1×. In general, the slowdown when running on one core is between 1.4× and 2.4× for all benchmarks, except for raytracer-jruby, where it remains virtually unchanged at 7×. The high slowdown in the JRuby benchmark does not originate from MDO contention but rather from the code instrumentation for the profiling itself. This overhead is reduced from 7× to 4× when we disable the use of `invokedynamic` for Ruby call-sites in the JRuby JIT compiler. The reason is that fewer profiled instructions are generated as part of the translated lambda-forms, which reduces the native code footprint in Tier 3, e.g., the largest raytracer-jruby method in Tier 3 from 135 kB to 90 kB.

### 4.4 Increase of Code Footprint

In addition to the 5.7× slowdown (geomean), the Tier 3 profiling increases the mean size of the generated native code by 2.6× (geomean) over all benchmarks. Figure 7 shows the *mean sizes* of the native code that is generated by each C1 compilation for all benchmarks. The blue bar shows the mean sizes of the C1 code without instrumentation when we limit the VM to tier up beyond Tier 1, the orange bar when we stop at Tier 3. In the latter case, the C1 generated code contains instrumented as well non-instrumented code. The error bars in the figure show the 90 % percentile interval. We found that that the distribution of the compilation sizes has a long tail regardless whether instrumentation is present. In fact, the distributions tend to follow a Pareto-like distribution. As the figure shows, the 90 % percentile intervals are much larger for Tier 3 code. The size of the largest compilation in dotty is 221 kB for the `TreePickler::pickleTree` by Tier 3. The added instrumentation increases the size of generated native code by more than 6×. The mean size of the compiled code for dotty, however, only increases by 3.7× when instrumentation is added. h2o-gbm is the next largest benchmark in terms of average Tier 3 code size and 90 % percentile interval. 2/3 of the compilations in h2o-gbm that are larger than 8 kB are for the 50 methods that implement the 50 decision trees of the ensemble of gradient-boosted trees. The high number of branch profiles in these methods significantly increases the code footprint.

### 4.5 Microarchitectural Impact

One concern when running code with a large footprint on modern out-of-order processors is the resulting increase of front-end pressure. We examine this for the two benchmarks with the largest compilations using the *top-down microarchitecture analysis* method [24]. Table 4 shows the break-down of issue slots in the pipeline. In the ideal case, there is an instruction (micro-op) retiring in every slot, i.e., retiring = 100 %. In practice, however, slots may be unfilled. Unfilled pipeline slots are classified as due to front-end (FE) or back-end (BE) stalls, or incorrect speculation. The table shows that dotty and h2o-gbm are both dominated by front-end stalls. In both cases, the Tier 3 profiling increases the front-end stalls by 10 percent points.

A further analysis shows that the primary cause of the increase in front-end stalls are fetch-latency issues due to instruction cache misses (+7 % cycles in dotty, +2 % in h2o-gbm) and branch resteers (+3 % in dotty, +4 % in h2o-gbm). Instruction cache misses can be a direct result of the larger code footprint. Branch resteers are caused by branch mispredictions that cause a delay fetching instructions from the

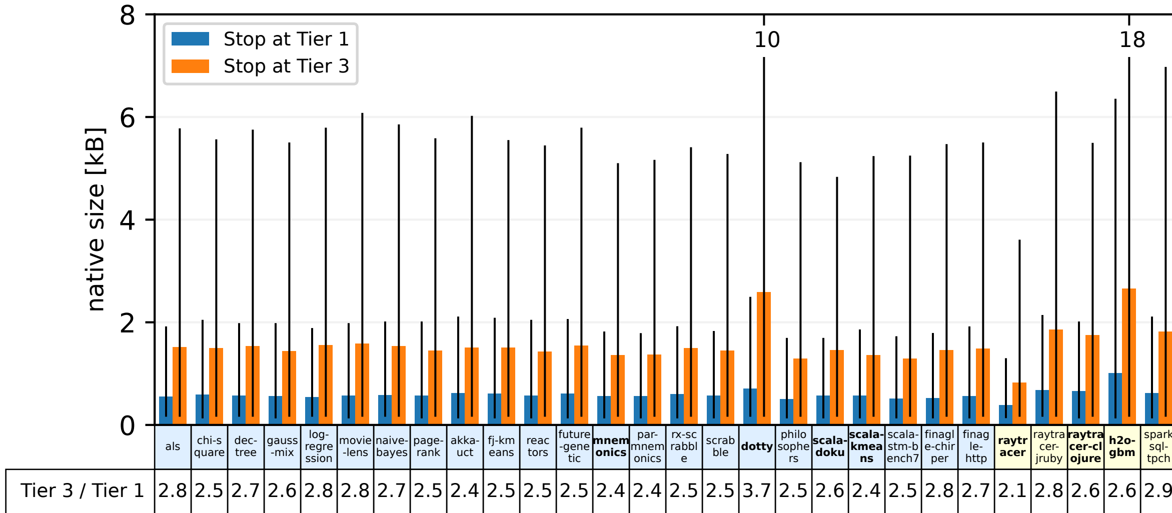



**Figure 7.** Machine code size when stopping in Tier 1 and the profiling Tier 3 for the Renaissance suite and our additional benchmarks. The bars show the mean values, the error lines indicate the 90 % percentile interval.

**Table 4.** Top-down microarchitectural analysis for dotty and h2o-gbm ( % of issue slots in the pipeline)

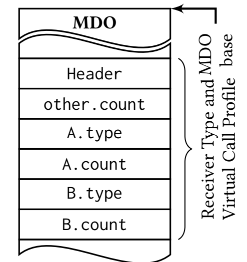
| | Tier | FE bound | BE bound | retiring | bad spec |
|---|---|---|---|---|---|
| dotty | 1 | 53 % | 10 % | 33 % | 4 % |
| | 3 | **63 %** | 7 % | 27 % | 3 % |
| h2o-gbm | 1 | 39 % | 21 % | 34 % | 7 % |
| | 3 | **49 %** | 11 % | 35 % | 4 % |

correct path. For h2o-gbm, we also observe issues in fetch-bandwidth. In Tier 3, the number of slots unfilled due to bandwidth limitations in the decode pipeline increases by 4 %. This seems to indicate that instruction streams containing profiling instrumentation are more difficult to decode for the processor. In all other single-threaded benchmarks, we do not observe such strong effects on the microarchitecture. In summary the added instrumentation by Tier 3, can increase the front-end pressure and affect applications that are already front-end-bound even stronger. This front-end bottleneck is a well-known problem in server workloads today. A lot of resources are invested on addressing this problem in modern processors, i.e., increasing instruction cache sizes, better branch predictors and instruction prefetchers, etc.

In this section, we showed the worst-case performance overhead by *restricting* HotSpot from moving beyond the profiling phase, which is an artificial setup not encountered in real-world deployments. Nevertheless, it reveals a potential performance issue in the current HotSpot JVM that is due to the contention on the data structures that hold the profiling data.

## 5 Case Study: Receiver Type Profile

In this section, we discuss one profile type in more detail: the *Receiver Type Profile*. It has the largest code footprint and contains multiple conditional branches. It is emitted by the C1 compiler to capture the type of the receiver object. The profile is collected for `invokevirtual`, `invokeinterface`, `instanceof`, as well as `checkcast`, and `aastore` bytecode instructions. The figure on the right shows the data layout of the receiver type profile in the MDO. The *Receiver Type Profile* consists of the two key-value pairs `A` and `B`, each storing the type (i.e., the class of the receiver) and the number of times (count) this particular type was encountered at this bytecode location. Thus, the profile remembers the first two distinct receiver types encountered and counts the number of occurrences. If more than two types are encountered, `other.count` is incremented.




The first slot in the receiver type profile, the `Header` field, contains the bytecode index, i.e., the location in the bytecode program, along with the type of the profile data and a number of flags. The flags indicate whether the instruction has caused a deoptimization event. We show the full code template of the receiver type profile in Listing 2. The code first checks whether the type of the current receiver object matches that in the `A` or `B` slots. If so, the corresponding counter is incremented. If not, it checks whether `A.type` or `B.type` do not yet have an assigned type. If a free slot is found, it is assigned to the

```
 1 ; rcx: pointer to the receiver object (rcvr)
 2   movabs MDO_base_addr,%r8 ;r8 <- MDO base
 3   mov    0x8(%rcx),%ecx    ;ecx <- klass ptr
 4   movabs $0x800000000,%r10
 5   add    %r10,%rcx  ; decompressed klass ptr
 6   cmp    a_type(%r8),%rcx  ;rcvr is A.type?
 7   jne    .l1               ;no -> test B.type
 8   addq   $0x1,a_count(%r8) ;A.count += 1
 9   jmpq   .l5               ;jump to end
10 .l1:
11   cmp    b_type(%r8),%rcx  ;rcvr is B.type?
12   jne    .l2               ;no -> is A empty?
13   addq   $0x1,b_count(%r8) ;B.count += 1
14   jmpq   .l5               ;jump to end
15 .l2:
16   cmpq   $0x0,a_type(%r8)  ;A.type is empty?
17   jne    .l3               ;no -> is B empty?
18   mov    %rcx,a_type(%r8)  ;A.type = rcvr
19   movq   $0x1,a_count(%r8) ;A.count = 1
20   jmpq   .l5               ;jump to end
21 .l3:
22   cmpq   $0x0,b_type(%r8)  ;B.type is empty?
23   jne    .l4        ;no -> increment 'other'
24   mov    %rcx,b_type(%r8)  ;B.type = rcvr
25   movq   $0x1,b_count(%r8) ;B.count = 1
26   jmpq   .l5               ;jump to end
27 .l4:
28   addq   $0x1,other_count(%r8) ;other.count+=1
29 .l5: ...
```

**Listing 2.** x64 instructions used to profile the receiver type of an `invokevirtual` call

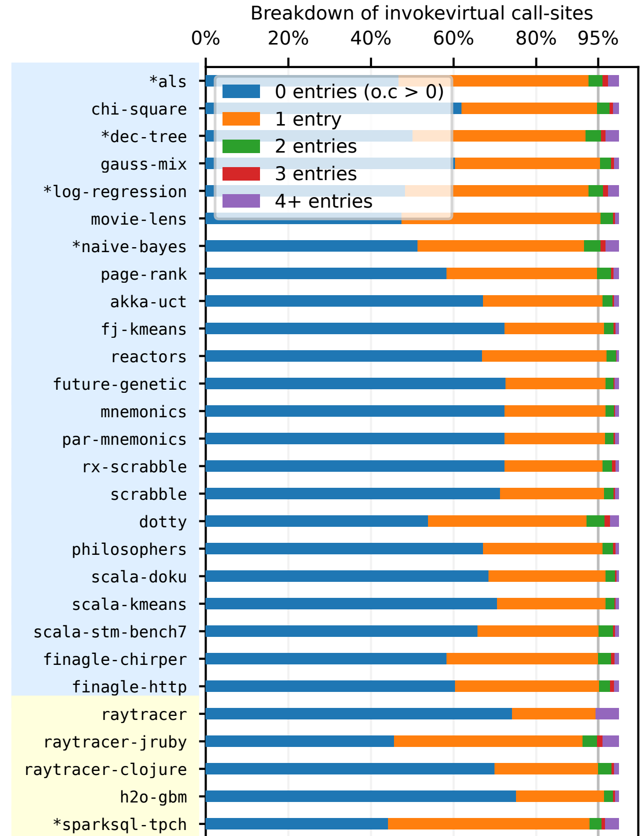



**Figure 8.** Polymorphism at `invokevirtual` call-sites measured as the number of entries in the receiver-type profile

object's receiver type and its count set to one. If no free slot is found, `other.count` is incremented.

Although C1 only leverages the first two types (in A and B), the number of profiled types can be increased to eight through the `XX:TypeProfileWidth` VM option. We use this feature to measure the degree of polymorphism for virtual calls in our benchmarks. A frequently cited observation is that the majority of calls in object-oriented programs with virtual method dispatch have only one single receiver type—[6] observes that a one-element in-line cache is effective about 95 % of the time. In other words, most call sites are monomorphic. Figure 8 shows the degree of polymorphism for our benchmarks. The figure shows the number of key-value entries used in the receiver type profile for which we increase the width from the two to eight (`XX:TypeProfileWidth=8`). The number of entries corresponds to the number of encountered types and, hence, the degree of polymorphism at the call site. The zero-entry `other.count > 0`, demands an explanation: C1 does not always need to emit the full profiling instrumentation snippet shown in Listing 2 for the receiver type profile. If it can determine through class hierarchy analysis that the virtual call site has a static binding, i.e., exactly one receiver, it replaces the complex code snippet shown in Listing 2 with that of a count data profile (Figure 2a) that simply counts the number of invocations by incrementing the `other.count` in the receiver profile type of the MDO. Sites with `other.count > 0`, thus, are deduced to be monomorphic and were reached at least once. In Figure 8, we omit call-sites that were never reached in the instrumented code, i.e., sites with zero-entries and `other.count = 0`. We consider the remaining zero-entries also as monomorphic.

The results in Figure 8 confirm earlier observations for the workloads we studied: the majority of virtual call sites are monomorphic. At least 95 % virtual calls are monomorphic for 17 out the 28 benchmarks. The exceptions in Renaissance are dotty (92 %) and all Spark benchmarks that internally use SparkSQL (marked with * in Figure 8). The JRuby raytracer has the lowest percentage of monomorphic call sites (91 %). This is not surprising since it runs inside JRuby, an interpreter for the dynamically typed language. Clojure is also dynamically typed but the Clojure raytracer has a larger fraction of monomorphic sites (95 %). The Java raytracer has only monomorphic sites except the three call-sites that refer to the four subtypes of `Object3D` used in the scene.

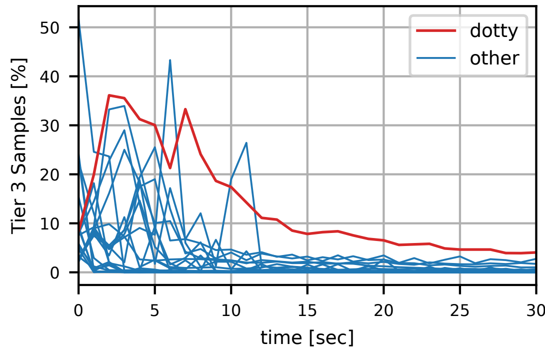



**Figure 9.** Fraction of Tier 3 samples during the first 30 s of the Renaissance benchmarks

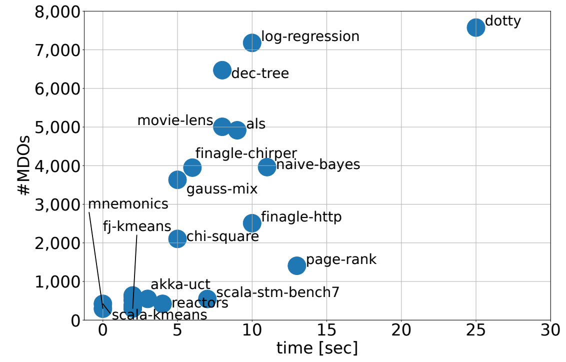



**Figure 10.** Tier-up times and method hotness for Renaissance benchmarks after a 30 s execution

From these results, we can make the following observations. First, at least 44 % of all virtual call-sites (shortest blue bar) have static binding and virtual dispatch is not needed. Second, in at least 21 % (shortest orange bar), inline-caching can optimize the virtual calls. A method dispatch through vtables is only needed in less than 9 % of the virtual calls (largest green + red + purple bars). From a profiling perspective, this means that in more than 44 % of the call sites, the expensive instrumentation for the receiver profile template (Listing 2) can be substituted with the simple counter in Figure 2a. Furthermore, in 80 % or more of call sites where the full receiver type profile is collected, the sites are monomorphic and the control-flow path (Listing 2) takes only one single conditional branch and one unconditional jump. The run-time overhead then is similar to that of a branch profile with a not-taken branch.

## 6 Tier-Up Behavior

The observable end-effect of the profile overhead depends on how long methods stay in the profiling phase in the unconstrained JVM. We quantify how fast an application tiers up by measuring the fraction of executed Tier 3 methods over time. We use Linux `perf` to sample the program and our JVMTI agent that annotates the compiled methods with the compiler tier that produced them. Figure 9 shows the fraction of samples in Tier 3 code for all Renaissance benchmarks during the first 30 s of their execution. As expected, all benchmarks start with a high fraction in Tier 3, which drops as methods get replaced by their optimized Tier 4 variants.

While most benchmarks do not spend any significant amount of time in Tier 3 after 12 seconds, we highlighted dotty as the outlier. It tiers up considerably slower than the other benchmarks. As shown in Section 4.2, the Tier 3 profiling instrumentation slows the single-threaded dotty benchmark down by 2.6×. Albeit this slowdown is smaller than for other benchmarks, profiling has a higher impact on the end-to-end performance in dotty.

The duration of the tier-up phase depends on the size of an application and the presence of pronounced hotspots. As a very coarse approximation, we quantify the size of the benchmarks by the number of allocated MDOs. This represents the number of unique methods that are invoked during execution since the VM creates an MDO for every profiled, compiled or inlined method. Similarly, we characterize the hotness of a method by its invocation count and number of loop back-edges taken in the method. Both counts are tracked in methods' data objects during Tier 2 and Tier 3 profiling. As in HotSpot's compiler policy, we use the sum of the two counts as a measure for the hotness of a method. Intuitively, an application tiers up faster if the invocation and back-edge counts are distributed over fewer MDOs. Consider dotty and dec-tree for example. For both benchmarks, HotSpot creates approximately 19,000 MDOs during the first 30 seconds. We find that 90 % of the invocation and back-edge counts fall on 7,575 MDOs in dotty, whereas in dec-tree 90 % of the counts occur in only 6,472 MDOs. This means that dec-tree has fewer hot methods (MDOs) and we would expect it to tier up faster than dotty.

In Figure 10, we show the tier-up time and the number of hot MDOs, i.e., the number of MDOs that account for over 90 % of invocation and back-edge counts, after the initial 30 s. We define the tier-up time as the time until the 95 % of application has tiered up such that the percentage of Tier 3 samples in a 1-second window (Figure 9) has dropped below 5 %. For example, the tier-up time for dotty is 25 s and for dec-tree is 8 s. The figure shows that benchmarks with a small number of top 90 %-MDOs, e.g., $<$ 1,000, tend to tier up quickly while dotty with the highest tier-up time also has the largest 90 % MDO count. For other benchmarks, the 90 %-MDO count has a weaker correlation with tier-up times.

## 7 Discussion and Related Work

We show that the performance overhead of profiling in the HotSpot JVM can be high. Furthermore, in the current implementation of HotSpot this overhead, including the one from

the bytecode compilation, is incurred every time the application starts. While this overhead is amortized in long-running applications, it poses a challenge when HotSpot is used in Function-as-a-Service (FaaS) scenarios since the user code has to move through the profiling tiers each time a function instance is started. AWS Lambda's *SnapStart* [2] improves the startup latency using checkpointing. However, as far as we understand, the checkpoint is taken *before* the function has handled the first request, i.e., before the application code has tiered up. Thus, this neither reduces profiling nor the compilation overhead. An interesting solution is *CRaC* (Coordinated Restore at Checkpoint)[20]. CRaC can create a full checkpoint of the JVM process executing a Java application when requested from within the application code, e.g., after a function in a FaaS setting has served enough requests that it has tiered up. CRaC can be used in combination with a snapshot orchestrator like Pronghorn [11], which automatically determines, stores and reuses the best checkpoint for each application. Similarly, Azul ReadyNow [3] persists the profiling logs and employs this information to accelerate warm-up in subsequent application runs.

A possibility is to switch from JIT to AOT compilation altogether, such as using Graal's *native-image* to create a machine binary, thereby closing Java's powerful open-world assumption. The CRaC approach, in contrast, still has a JIT compiler that could be used for run-time adaptation and leaves the open-world assumption open.

Once fully tiered up, no further profiling data is collected in the current HotSpot JVM. This lack of information limits later optimizations to adapt to changes in workload characteristics. Like other runtimes [8], HotSpot is known to suffer from stale or polluted profiles; the longer a program runs, the "more generic" the machine code becomes. Sporadic low-overhead updates of the profiling data while the code is fully tiered-up may help detect changes in the workload characteristics. Such a low-overhead profiling could enable more sophisticated optimizations.

Profiling is used as a means of optimization in static languages as well, through dynamic binary translation (DBT), dynamic binary optimization (DBO) and feedback-driven optimization (FDO). Large-scale studies coming from Google [5, 22] and Facebook/Meta [16, 17] give a sense of the gains obtained with FDO and post-link optimization. More precisely, they perform fleet-wide profiling of application binaries and install the new optimized code once it is generated. The two techniques show similar performance. Profile-guided optimization has also been shown to provide speedups in the context of memory-restricted applications, e.g., on mobile devices [12]. The critical difference between JIT profiling and DBO/FDO is that in JIT-compiled languages the profiling and feedback loop are readily included in the compilation mechanism. There is no need for back and forth translation, as the system naturally optimizes the code as it goes.

***Related Work.*** A large number of studies on JIT compilers for dynamic languages touch on profiling costs. Although missing a clear quantification of what makes profiling expensive, many works propose solutions to mitigate this overhead, e.g., through profile caching [13] and cross-run sharing [1, 4, 10, 15]. Our work complements these studies with a comprehensive characterization of profiling costs.

By contrast, Dot et al. [7] propose collecting *additional* profiling data in order to reduce the costs of the run-time checks specific to dynamic languages [18]. The additional profiling information is inexpensive to collect but allows for optimizations that lead to 7 % speedup on average.

Wade et al. [23] assess the impact of different profile types on the performance of generated code in JIT vs. AOT compilation. The authors use the DaCapo and SPECjvm2008 suites on OpenJDK9 and present a breakdown based on profile types and their contribution to the total impact. The authors conclude that even a small amount of profiling can significantly improve the generated code. Our work is orthogonal to this study, as it looks at the costs of profiling, provides a deep-dive into the receiver type profile and tier-up behavior of real-world applications running on OpenJDK 17.

## 8 Conclusions

We study the profiling overhead caused by code instrumentation in the C1 compiler of the HotSpot JVM. We report a slowdown of up to 35× over the code generated by the same compiler without profiling instrumentation. The largest overhead is observed in multi-threaded applications and as the main cause, we identify the memory contention on the profiling data structures. Although the impact of this overhead is not significant for applications with a pronounced hotspot and that quickly move through the profiling phase, we find one benchmark in the Renaissance suite that tiers up slowly and experience a higher impact.

The receiver type profile requires the most complex code instrumentation and is also among the most frequently collected profiles. However, its complexity often does not cause a significant run-time overhead as we found 95 % of the call sites to be monomorphic. Furthermore, in at least 44 % of the call sites, the C1 compiler can already deduce that they are monomorphic from the class hierarchy analysis and elide the type profile. Call sites without such a static binding are profiled. However, they end up being monomorphic in 80 % of the cases. Thus, only one conditional branch, which is predictable as always-taken, is executed at run-time, lowering the effective profiling costs.

A low-overhead, hardware-assisted profiling mechanism, would allow profiling in fully-tiered code and permit the JVM to apply speculative optimizations adapted to changes in the workload. We believe that pursuing this avenue is worthwhile as it may enable performance improvements in JIT runtimes beyond traditional native compilation.

# A Profiling Overheads on Arm64

We provide a brief analysis of the profiling overheard for the 64-bit Arm architecture. In general, we observe a similar behavior as on the Intel system. Figure 11 shows the arm64 assembly code for the count profile, branch profile, and the method-level invocation count. The shown profiling snippets require two to three instructions more on arm64 than on x64. The main reason is that setting the 48-bit MDO address in a register requires three instructions, each setting 16-bits of

```
1 ; x0: pointer to the receiver object (rcvr)
2   mov  x4,MDO_addrlo16       ; x4 <- MDO base
3   movk x4,MDO_addrmi16,lsl #16
4   movk x4,MDO_addrhi16,lsl #32
5   ldr  w0,[x0,8]             ; w0 <- klass ptr
6   eor  x0,x0,0x800000000   ; decompr. klass ptr
7   add  x9,x4,a_type_off
8   ldr  x8,[x9]               ; x8 <- A.type
9   cmp  x0,x8                 ; rcvr is A.type?
10  b.ne .l1                   ; no -> test B.type
11  ldr  x8,[x4,a_count_off]
12  add  x8,x8,1               ; A.count += 1
13  str  x8,[x4,a_count_off]
14  b    l5                    ; jump to end
15 .l1:
16  add  x9,x4,b_type_off
17  ldr  x8,[x9]               ; x8 <- B.type
18  cmp  x0,x8                 ; rcvr is B.type?
19  b.ne .l2                   ; no -> is A empty?
20  ldr  x8,[x4,b_count_off]
21  add  x8,x8,1               ; B.count += 1
22  str  x8,[x4,b_count_off]
23  b    .l5                   ; jump to end
24 .l2:
25  add  x9,x4,a_type_off
26  ldr  x8,[x9]               ; A.type is empty?
27  cbnz x8,.l3                ; no -> is B empty
28  str  x0,[x9]               ; A.type = rcvr
29  orr  x8,xzr,1
30  add  x9,x4,a_count_off
31  str  x8,[x9]               ; A.count = 1
32  b    .l5                   ; jump to end
33 .l3:
34  add  x9,x4,b_type_off
35  ldr  x8,[x9]               ; B.type is empty?
36  cbnz x8,.l4        ; no -> increment 'other'
37  str  x0,[x9]               ; B.type = rcvr
38  orr  x8,xzr,1
39  add  x9,x4,b_type_count
40  str  x8, [x9]              ; B.count = 1
41  b    .l5
42 .l4
43  ldr  x8,[x4,other_count_off]
44  add  x8,x8,1               ; other.count += 1
45  str  x8,[x4,other_count_off]
46 .l5: ...
```

**Listing 3.** arm64 instructions used to profile the receiver type of an `invokevirtual` call

**Table 5.** arm64: Top-down microarchitectural analysis for dotty and h2o-gbm (% of issue slots in the pipeline)

| | Tier | FE bound | BE bound | retiring | bad spec |
|---|---|---|---|---|---|
| dotty | 1 | 56 % | 15 % | 22 % | 7 % |
| | 3 | **68 %** | 9 % | 19 % | 3 % |
| h2o-gbm | 1 | 36 % | 24 % | 33 % | 6 % |
| | 3 | **49 %** | 16 % | 33 % | 3 % |

the address, instead of one single instruction on x64, which can contain the full MDO address in the immediate field of the move (`movabs`) instruction. The footprint of the three arm64 instructions is two bytes larger than that of `movabs`. The instruction count increase is further exacerbated by C1's suboptimal register use. It moves the MDO address into a register anew for every profiled bytecode instruction rather than retaining the value in a register for reuse. For example, in Listing 1, the MDO address of the `render` method is set more than 40 times, even though sufficient unused architecture registers are available. The assembly snippet for collecting the receiver type profile (Listing 3) is even larger. With 40 instructions, it is almost twice as large as on x64.

We repeat the overhead analysis from Section 4 on an Arm system. Because this is a dual-socket server-class system, we limit the number of CPU cores available to the JVM to 16, the same number as the logical core count on the Intel system, in order to obtain a comparable system configuration. Furthermore, we bind the JVM to use memory from the NUMA zone that is local to 16 CPU cores.

Figure 12 shows the profiling overheads in Tier 2 and Tier 3 for the Arm system. The overheads are similar to those of the Intel system. The geometric mean of the profiling slowdown is 5.0× compared to 5.7× on the Intel. We also identify fj-kmeans on arm64 as the benchmark that experiences the highest profiling overhead in Tier 3 with a 34× slow-down compared to Tier 1 (cf. 35× on x64 in Figure 4). The overheads in naive-bayes and akka are slightly higher for the Arm configuration, while philosophers shows a lower overhead. The Tier 3 profiling instrumentation increases the code footprint by 2.7× (geomean) compared to 2.6× on Intel. As expected, the average size of the code of a compilation when restricting the JVM to stop at Tier 3 is 8.5 % higher on arm64 than on x64. Since the Tier 1 code is only 2.4 % larger on arm64 than on x64, we conclude that C1's profiling instrumentation on Arm indeed causes an increase in code footprint size. However, despite this increase, only 1/3 of benchmarks experience a higher slow-down on arm64 and, hence, has a small performance impact.

We also repeat the top-down microarchitectural analysis (Table 5) and observe a similar behavior as on x64. Both benchmarks are also front-end bound on arm64. The profiling instrumentation further increases the percentage of pipeline slots that are empty due to front-end issues.

```
1 ; e.g., invokestatic
2 mov   x3,MDO_addrlo16
3 movk  x3,MDO_addrmi16,lsl 16
4 movk  x3,MDO_addrhi16,lsl 32
5 ldr   x8,[x3,count_off]
6 add   x8,x8,1
7 str   x8,[x3,count_off]
8 ...
```

**(a)** Count Profile

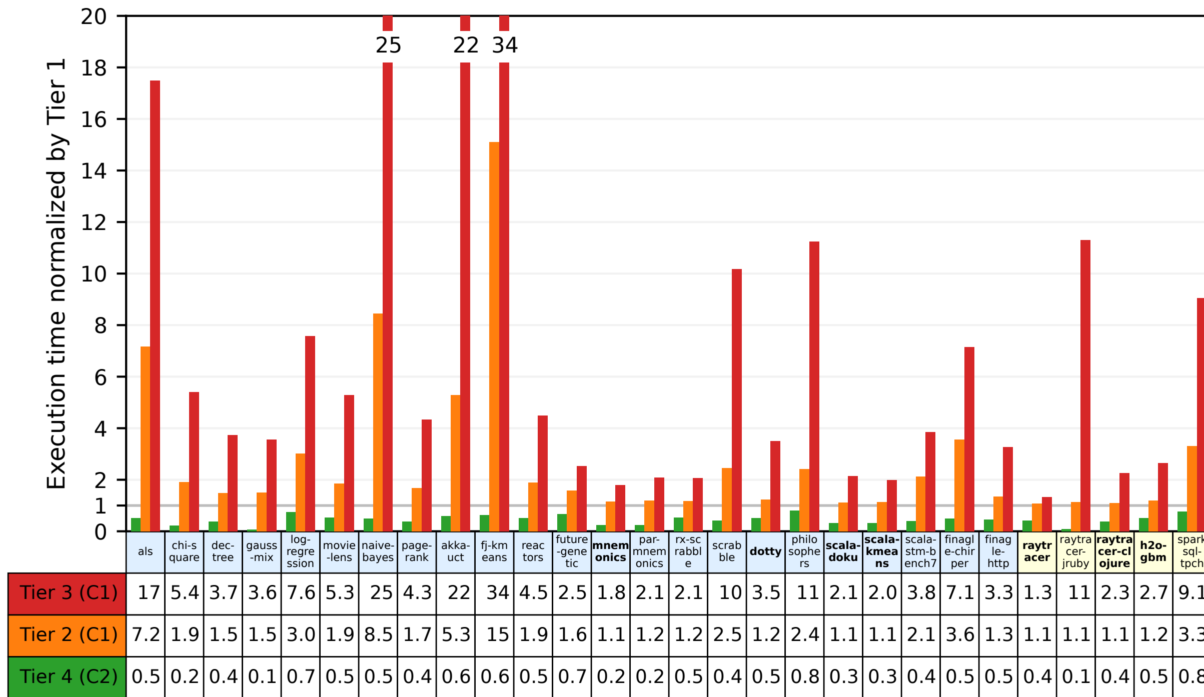
```
 1 ; e.g., ifeq
 2   mov   x0,MDO_addrlo16
 3   movk  x0,MDO_addrmi16,lsl 16
 4   movk  x0,MDO_addrhi16,lsl 32
 5   mov   x8,taken_off
 6   mov   x9,not_taken_off
 7   csel  x2,x8,x9,eq
 8   ldr   x3,[x0,x2]
 9   add   x3,x3,1
10   str   x3,[x0,x2]
11   b.eq  .l1
12   ...
13 .l1: ; actual target
```

**(b)** Branch Profile

```
 1 ; in method preamble
 2 mov   x0,MDO_addrlo16
 3 movk  x0,MDO_addrmi16,lsl 16
 4 movk  x0,MDO_addrhi16,lsl 32
 5 ldr   w2,[x0,invoc_cnt_off]
 6 add   w2,w2,2
 7 str   w2,[x0,invoc_cnt_off]
 8 and   w2,w2,overflow_mask
 9 cmp   w2,0
10 b.eq  counter_overflow
11 ...
```

**(c)** Invocation Count

**Figure 11.** arm64 assembly code for count and branch profiles, and the invocation counter in the method preamble




| | als | chi-square | dec-tree | gauss-mix | log-regression | movie-lens | naive-bayes | page-rank | akka-uct | fj-kmeans | reactors | future-genetic | **mnemonics** | par-mnemonics |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Tier 3 (C1) | 17 | 5.4 | 3.7 | 3.6 | 7.6 | 5.3 | 25 | 4.3 | 22 | 34 | 4.5 | 2.5 | 1.8 | 2.1 |
| Tier 2 (C1) | 7.2 | 1.9 | 1.5 | 1.5 | 3.0 | 1.9 | 8.5 | 1.7 | 5.3 | 15 | 1.9 | 1.6 | 1.1 | 1.2 |
| Tier 4 (C2) | 0.5 | 0.2 | 0.4 | 0.1 | 0.7 | 0.5 | 0.5 | 0.4 | 0.6 | 0.6 | 0.5 | 0.7 | 0.2 | 0.2 |

| | rx-scrabble | scrabble | **dotty** | philosophers | **scala-doku** | **scala-kmeans** | scala-stm-bench7 | finagle-chirper | finagle-http | **raytracer** | raytracer-jruby | **raytracer-clojure** | **h2o-gbm** | spark-sql-tpch |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Tier 3 (C1) | 2.1 | 10 | 3.5 | 11 | 2.1 | 2.0 | 3.8 | 7.1 | 3.3 | 1.3 | 11 | 2.3 | 2.7 | 9.1 |
| Tier 2 (C1) | 1.2 | 2.5 | 1.2 | 2.4 | 1.1 | 1.1 | 2.1 | 3.6 | 1.3 | 1.1 | 1.1 | 1.1 | 1.2 | 3.3 |
| Tier 4 (C2) | 0.5 | 0.4 | 0.5 | 0.8 | 0.3 | 0.3 | 0.4 | 0.5 | 0.5 | 0.4 | 0.1 | 0.4 | 0.5 | 0.8 |

**Figure 12.** arm64: Profiling overheads in Tier 2 and Tier 3 relative to Tier 1 for the Renaissance suite and our additional benchmarks. Benchmarks in **bold** are single-threaded.

## References

[1] Matthew Arnold, Adam Welc, and V. T. Rajan. 2005. Improving Virtual Machine Performance Using a Cross-Run Profile Repository. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA) *(OOPSLA '05)*. Association for Computing Machinery, New York, NY, USA, 297–311. https://doi.org/10.1145/1094811.1094835

[2] AWS 2022. *Improving startup performance with Lambda SnapStart.* AWS. https://docs.aws.amazon.com/lambda/latest/dg/snapstart.html

[3] Azul 2023. *Azul Platform Prime ReadyNow: Solving the Java Warmup Problem.* Azul. https://www.azul.com/wp-content/uploads/Prime-Data-Sheet-ReadyNow-Orchestrator.pdf

[4] Joao Carreira, Sumer Kohli, Rodrigo Bruno, and Pedro Fonseca. 2021. From Warm to Hot Starts: Leveraging Runtimes for the Serverless Era. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Ann Arbor, Michigan) *(HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 58–64. https://doi.org/10.1145/3458336.3465305

[5] Dehao Chen, David Xinliang Li, and Tipp Moseley. 2016. AutoFDO: Automatic Feedback-Directed Optimization for Warehouse-Scale Applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization* (Barcelona, Spain) *(CGO'16)*. Association for Computing Machinery, New York, NY, USA, 12–23. https://doi.org/10.1145/2854038.2854044

[6] L. Peter Deutsch and Allan M. Schiffman. 1984. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Salt Lake City, Utah, USA) *(POPL '84)*. 297–302. https://doi.org/10.1145/800017.800542

[7] Gem Dot, Alejandro Martínez, and Antonio González. 2017. Removing checks in dynamically typed languages through efficient profiling. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 257–268. https://doi.org/10.1109/CGO.2017.7863745

[8] Oliver Flückiger, Jan Ječmen, Sebastián Krynski, and Jan Vitek. 2022. Deoptless: Speculation with Dispatched on-Stack Replacement and Specialized Continuations. In *Proc. of PLDI'22 (PLDI 2022)*. 749–761. https://doi.org/10.1145/3519939.3523729

[9] Graham Hamilton, Gilad Bracha, and Danny Coward. 2011. *JSR 292: Supporting Dynamically Typed Languages on the Java Platform.* Java Specification Requests.

[10] Alexey Khrabrov, Marius Pirvu, Vijay Sundaresan, and Eyal de Lara. 2022. JITServer: Disaggregated Caching JIT Compiler for the JVM in the Cloud. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 869–884. https://www.usenix.org/conference/atc22/presentation/khrabrov

[11] Sumer Kohli, Shreyas Kharbanda, Rodrigo Bruno, Joao Carreira, and Pedro Fonseca. 2024. Pronghorn: Effective Checkpoint Orchestration for Serverless Hot-Starts. In *Proceedings of the 19th European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 298–316. https://doi.org/10.1145/3627703.3629556

[12] Kyungwoo Lee, Ellis Hoag, and Nikolai Tillmann. 2022. Efficient Profile-Guided Size Optimization for Native Mobile Applications. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction* (Seoul, South Korea) *(CC 2022)*. Association for Computing Machinery, New York, NY, USA, 243–253. https://doi.org/10.1145/3497776.3517764

[13] Zoltan Majo, Tobias Hartmann, Marcel Mohler, and Thomas R. Gross. 2017. Integrating Profile Caching into the HotSpot Multi-Tier Compilation System. In *Proceedings of the 14th International Conference on Managed Languages and Runtimes* (Prague, Czech Republic) *(ManLang 2017)*. Association for Computing Machinery, New York, NY, USA, 105–118. https://doi.org/10.1145/3132190.3132210

[14] Angela Nicoara, Gustavo Alonso, and Timothy Roscoe. 2008. Controlled, Systematic, and Efficient Code Replacement for Running Java Programs. In *Proc. of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008 (Eurosys '08)*. 233–246. https://doi.org/10.1145/1352592.1352617

[15] Guilherme Ottoni and Bin Liu. 2021. HHVM Jump-Start: Boosting Both Warmup and Steady-State Performance at Scale. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event, Republic of Korea) *(CGO '21)*. IEEE Press, 340–350. https://doi.org/10.1109/CGO51591.2021.9370314

[16] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. BOLT: A Practical Binary Optimizer for Data Centers and Beyond. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA) *(CGO'19)*. IEEE Press, 2–14.

[17] Maksim Panchenko, Rafael Auler, Laith Sakka, and Guilherme Ottoni. 2021. Lightning BOLT: Powerful, Fast, and Scalable Binary Optimization. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction* (Virtual, Republic of Korea) *(CC 2021)*. Association for Computing Machinery, New York, NY, USA, 119–130. https://doi.org/10.1145/3446804.3446843

[18] Alberto Parravicini and Rene Mueller. 2021. The Cost of Speculation: Revisiting Overheads in the V8 JavaScript Engine. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*. 13–23. https://doi.org/10.1109/IISWC53511.2021.00013

[19] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. In *Proc of PLDI*. 31–47. https://doi.org/10.1145/3314221.3314637

[20] Simon Ritter. 2022. Java on CRaC: Superfast JVM Application Startup. In *Devoxx Conference Belgium*.

[21] John Rose. 2012. *JEP 160: Lambda-Form Representation for Method Handles.* JDK Enhancement Proposals.

[22] Han Shen, Krzysztof Pszeniczny, Rahman Lavaee, Snehasish Kumar, Sriraman Tallam, and Xinliang David Li. 2023. Propeller: A Profile Guided, Relinking Optimizer for Warehouse-Scale Applications. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 617–631. https://doi.org/10.1145/3575693.3575727

[23] April W. Wade, Prasad A. Kulkarni, and Michael R. Jantz. 2017. AOT vs. JIT: Impact of Profile Data on Code Quality. In *Proceedings of the 18th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (Barcelona, Spain) *(LCTES 2017)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3078633.3081037

[24] Ahmad Yasin. 2014. A Top-Down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) (ISPASS 2014)*. 35–44.

[25] ZZZ-airline-data 2020. Airline Reporting Carrier On-Time Performance Dataset. https://dax-cdn.cdn.appdomain.cloud/dax-airline/1.0.1/airline.tar.gz Last accessed June 2023.


Received 2024-05-25; accepted 2024-06-24

# Author Index


Aigner, Christoph . . . . . . . . . . . . . . . . . . 2
Aslandukov, Matvii . . . . . . . . . . . . . 112

Barany, Gergö . . . . . . . . . . . . . . . . . . . . 2
Black-Schaffer, David . . . . . . . . . . . . 14
Bovel, Matthieu . . . . . . . . . . . . . . . . . 55
Burchell, Humphrey . . . . . . . . . . . . . 82

Carpen-Amarie, Maria . . . . . . . . . . 112
Chiba, Shigeru . . . . . . . . . . . . . . . . . . 28

Flesselle, Eugene . . . . . . . . . . . . . . . . 55

Goodacre, John . . . . . . . . . . . . . . . . . . 65

Larose, Octave . . . . . . . . . . . . . . . . . . 82

Lepper, Markus . . . . . . . . . . . . . . . . . . 48
Luján, Mikel . . . . . . . . . . . . . . . . . . . . 65

Marr, Stefan . . . . . . . . . . . . . . . . . . . . 82
Mochizuki, Fumika . . . . . . . . . . . . . . 28
Moriguchi, Sosuke . . . . . . . . . . . . . . . 41
Mössenböck, Hanspeter . . . . . . . . . . . 2
Mueller, Rene . . . . . . . . . . . . . . . . . . 112

Norlinder, Jonas . . . . . . . . . . . . . . . . . 14

Ogawa, Eri . . . . . . . . . . . . . . . . . . . . . . 98

Petoumenos, Pavlos . . . . . . . . . . . . . 65

Racordon, Dimitri . . . . . . . . . . . . . . . 55

Shioya, Ryota . . . . . . . . . . . . . . . . . . . 98
St. Amour, Leo . . . . . . . . . . . . . . . . . . 90
Suzuki, Go . . . . . . . . . . . . . . . . . . . . . . 41

Tilevich, Eli . . . . . . . . . . . . . . . . . . . . . 90
Titzer, Ben L. . . . . . . . . . . . . . . . . . . . . 1
Tovletoglou, Konstantinos . . . . . . . 112
Trancón y Widemann, Baltasar . . . 48

Watanabe, Takuo . . . . . . . . . . . . . . . . 41
Wright, Christopher John . . . . . . . . 65
Wrigstad, Tobias . . . . . . . . . . . . . . . . 14

Yamazaki, Tetsuro . . . . . . . . . . . . 28, 98
Yang, Albert Mingkun . . . . . . . . . . . . 14