Idletime Scheduling with Preemption Intervals

20th ACM Symposium on Operating System Principles, Brighton 2005

Lars Eggert and Joseph D. Touch

Presented by
Gerd Dauenhauer

Software Systems Seminar 2007,
University of Salzburg,
Department for Computer Sciences

Motivation

Many computer system resources are often unused.

- Studies on CPU utilization: 70% of machines idle at any given time

We want to utilize this idle capacity for productive background work.

Examples:

- Maintenance tasks, such as virus checking or file system optimization
- Caches and pre-fetching systems, such as for likely future web requests
- Distributed computing, such as X@home

➔ Get something for nothing. Speed is secondary objective.

Problem: Foreground work should not be affected by background work.

Objectives

Presence of background work should be transparent to foreground work:

- Side effects should be hidden.
  - E.g. files created in the background should not be visible to regular tasks.
- Performance impact should be minimal.
  - Execution time of regular tasks should be equal with or without background tasks.
  - Background work should only fill the gaps.

➔ Idletime scheduling only aims at the performance.

Wide variety of applications should by supported:

- Idle time mechanism must be tunable.
- Existing applications must be supported without modifications.

Only small changes in operating systems and APIs should be necessary for deployment.

Key Concept: Preemption Interval

New low-priority service class for idletime background tasks.

Not enough. Problem is:

Idletime scheduling can still delay regular foreground execution.

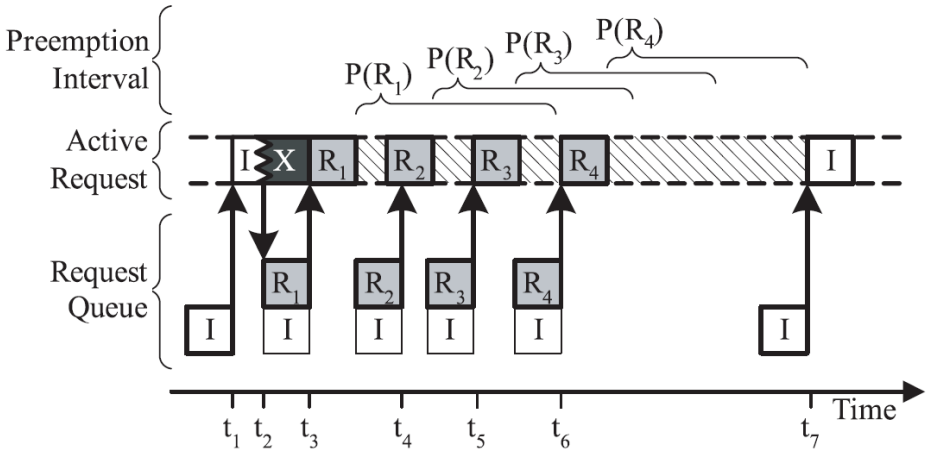
- Preemption cost for stopping the low priority background task and starting the high priority foreground task

➔ Minimizing the preemption costs is the key objective.

This is done by introducing the *Preemption Interval*:

- Intentional delay execution of background tasks to minimize preemptions

Key Concept: Preemption Interval II



Preemption Interval (PI): time period after service of a foreground request

During PI no background task requests are started

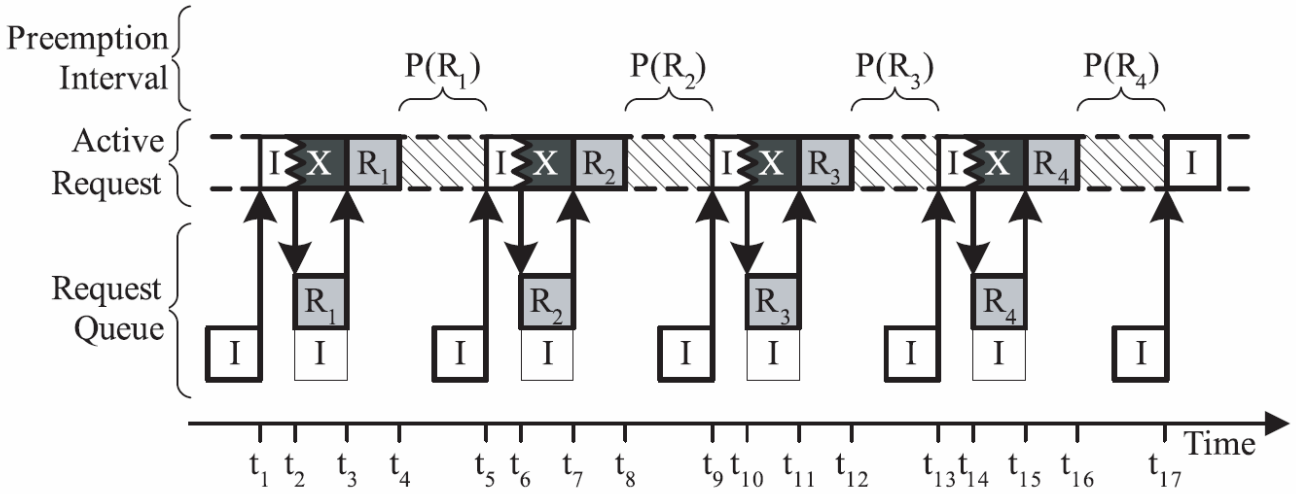
During PI new foreground tasks *do* start

Each new foreground task extends the PI

Sequences of foreground requests with PIs longer than their inter-arrival time form bursts.

➔ At most one preemption within a burst of foreground requests can happen.

Variations of the Preemption Interval



Preemption interval length can not be chosen arbitrarily:

To short PI (less than the inter-arrival gaps of foreground requests):

- Idle time scheduling mechanism is ineffective,
  - Background request is started and preempted

➔ Each foreground request has full preemption overhead.

This situation also occurs for very light foreground workloads.

Then it may be acceptable.

Variations of the Preemption Interval II

Indefinite PI inhibits any idletime requests from being served.

- Foreground performance is identical to a system without idletime scheduling.

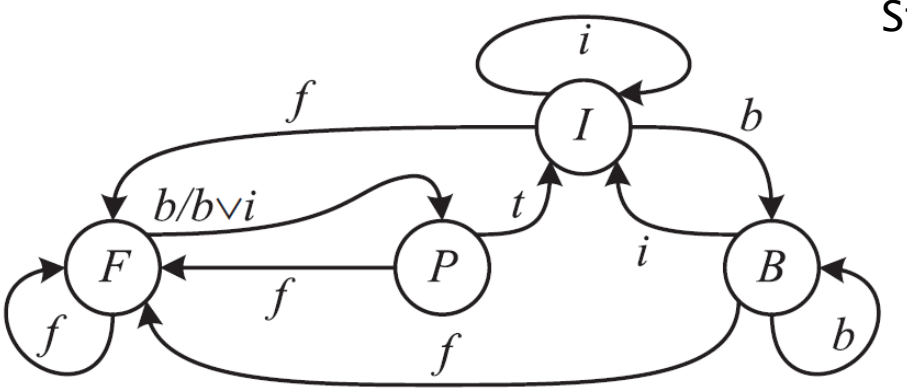
With zero length PIs, idletime scheduling is ineffective.

The system is identical to a system with traditional priority queues.

A useful upper bound for the PI length is the average foreground request inter-arrival time.

➔ Bursts of foreground requests with at most one preemption delay.

State Diagram



States:

- I – Idle (=start)
- F – Serving a foreground request
- P – Preempting Interval, not serving a request
- B Serving a background request

Events:

- f – Foreground request waiting at head of queue
- b – Background request waiting at head of queue (no foreground requests in queue due to priorities)
- t– Preemption interval expired
- i – Queue is empty

Event b is not consumed on F→P transition (signified by b/b): b is required by B

B→F transition causes a preemption

Implementation

Idletime scheduling is currently implemented on FreeBSD 4.7 for:

- Disk drive
- Network interface

Applications tag a resource as either regular or idletime through a new idletime option for file descriptors (including sockets).

- *fcntl()* call for files
- *setsockopt()* call for sockets

➔ The idletime schedulers use the tags to prioritize requests on these streams.

Other possible idletime APIs would be possible, such as

- CPU priorities: treat all resource requests as idletime if priority < limit
- Specific port ranges or IP addresses could indicate idletime use

Implemented through standard BSD timing facility.
Timer start at begin of request for implementation reasons.

Disk Scheduler

Operates at the border between buffer cache and block device driver.

Problem: Request priorities are not sufficient.

- Disk drives often re-order requests internally without priorities.

➔ Preemption intervals prevent idletime requests from entering the hardware queue.

Replaces the standard *disksort* algorithm (C-LOOK variant of the *elevator seek* algorithm).
Uses two C-LOOK queues for foreground and background requests.

Caching of background data could throw foreground data out of limited cache.

➔ Caching for background requests is disabled. Except disk hardware cache.

Speculative read-ahead (UFS performance enhancing feature) for background data would cause additional delays.

➔ Read-ahead is also disabled for background requests. Not useful anyway since no cache.

Network Scheduler

Operates at the border between network protocols and network interface driver.

Packets sent from a socket with idletime option set are tagged:

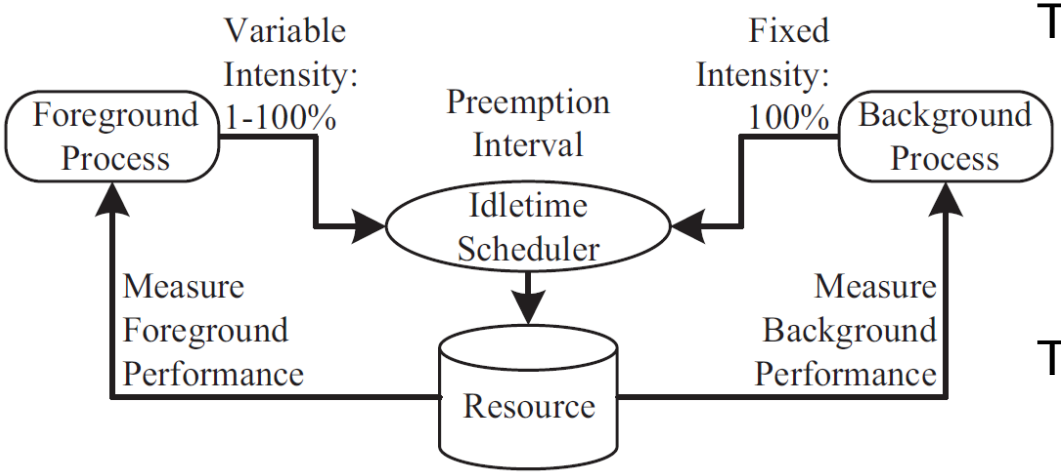
- IP header *type-of-service* field value 0x20 for IPv4
- Traffic class could be used for IPv6 (not implemented)

When receiving a packet with idletime tag, the idletime option is set for the socket. Sends on this socket will then also be idletime.

Replaces the standard FIFO queue for outbound traffic with queuing based on a modified ALTQ framework.

Currently, idletime scheduling is enabled only for only outbound traffic.

Experimental Set-Up



Two Processes on a machine, generating:

- Foreground request
- Background request

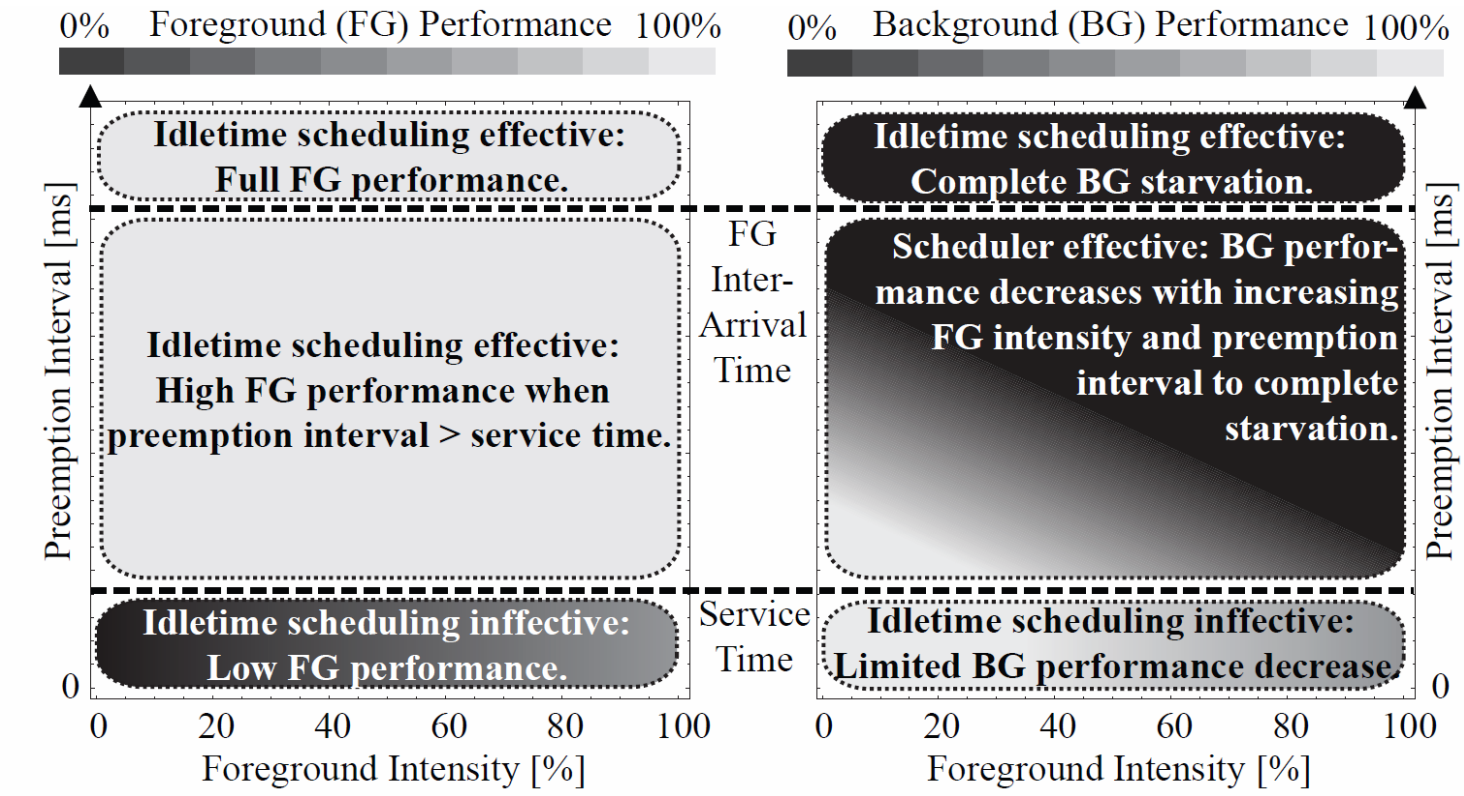
Two parameters:

- Intensity, i.e. fraction of CPU time used to generate requests (One request at least)
- Preemption interval length, interval > CPU quantum stalls idletime use

PC workstation running FreeBSD 4.7 with modified ALTQ framework

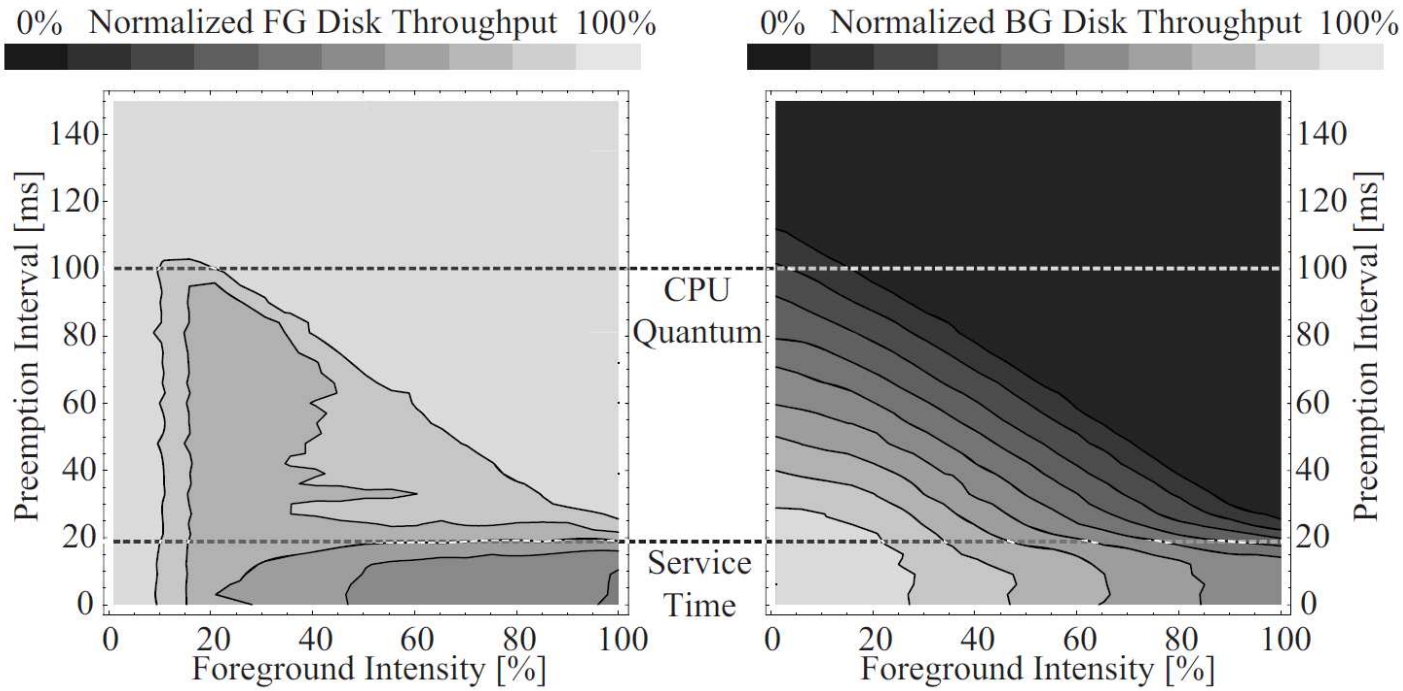
Dual CPU, eliminating contention between foreground and background processes

Expected Performance



Normalized against baseline without idletime scheduling

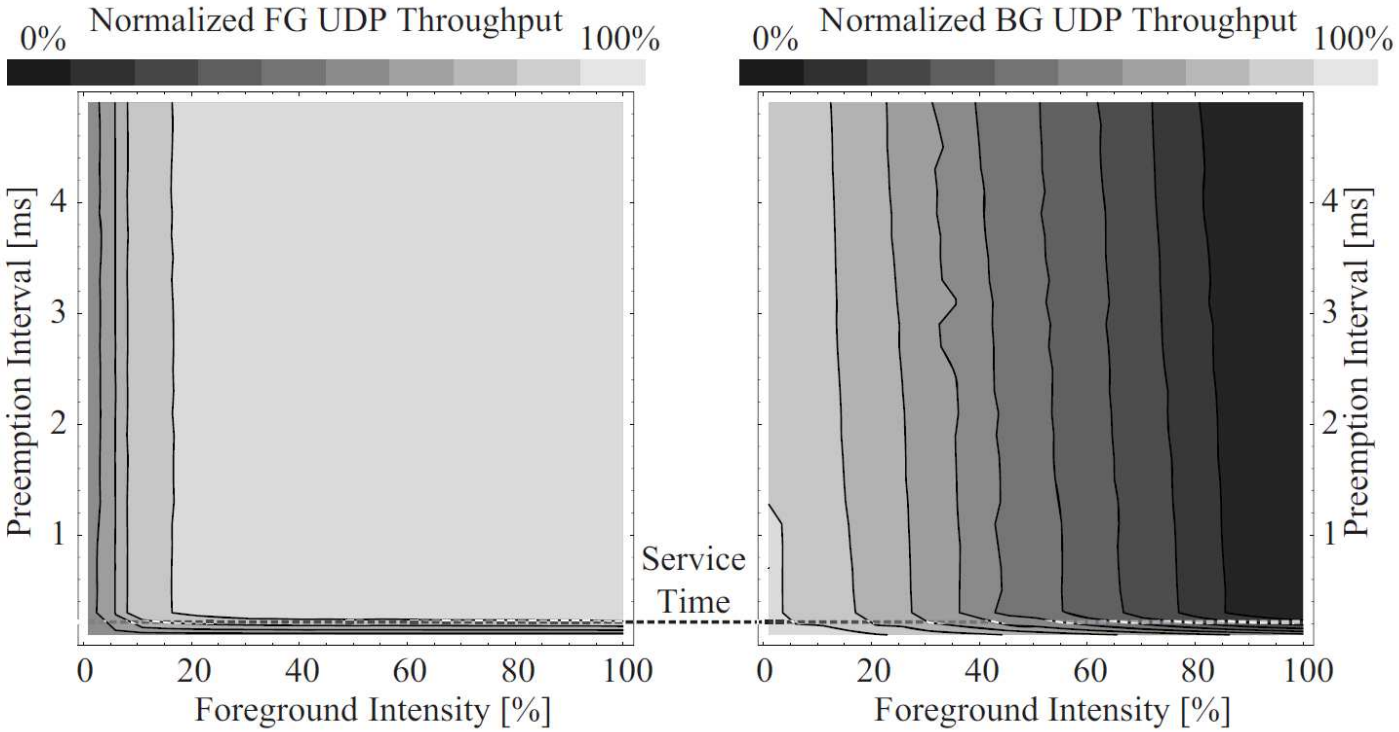
Disk Scheduler Results



Random access on 8.2GB ATA disk: seek time (15ms) + mean latency (5ms) = service time (20ms)
Reading 512 byte blocks

<20 ms: idletime scheduling ineffective
>100 ms: idletime scheduling suppresses background requests
<10% foreground intensity: priority queuing of request sufficient (delay not shown)

Network Scheduler Results

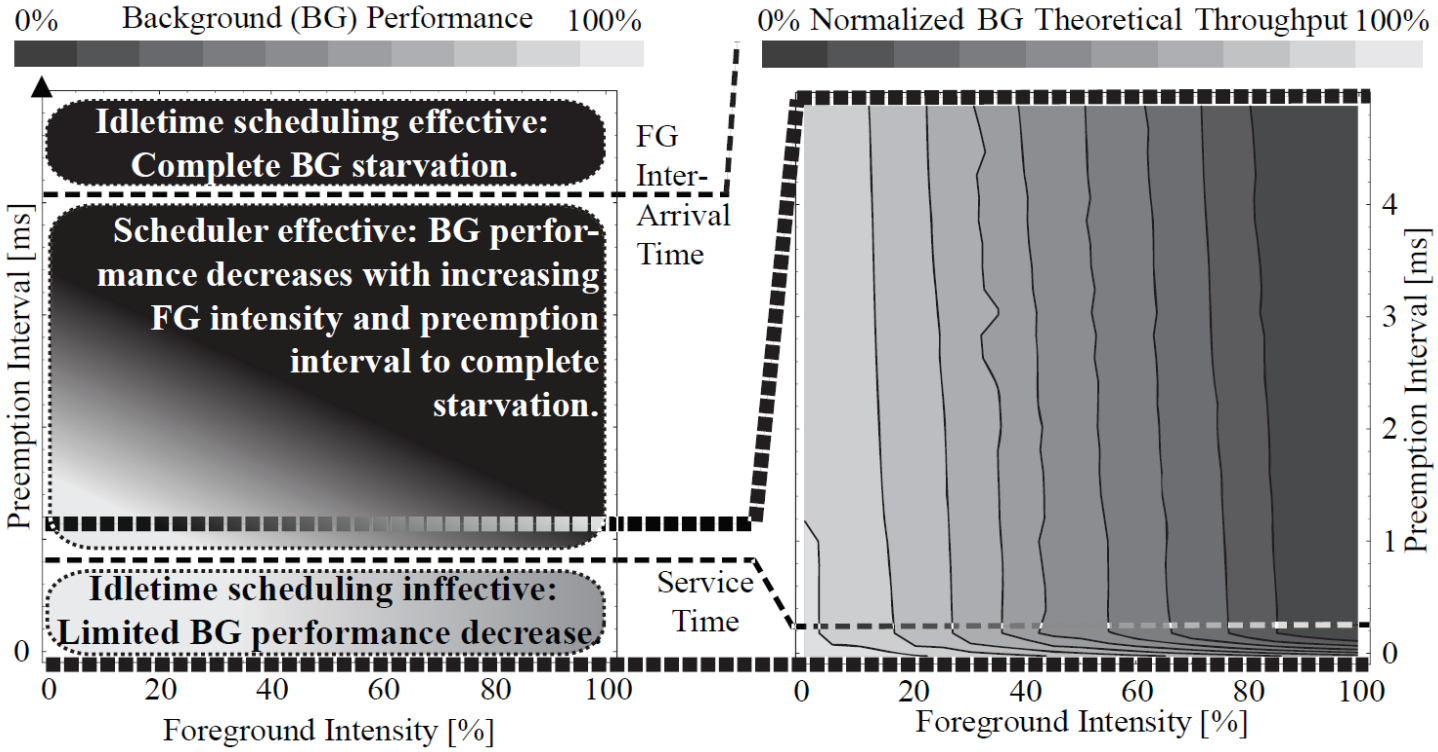


1 Gbit/s interface, UDP transmission to discard service (1400 Bytes/packet)

Service time 0.05 ms (measured),

orders of magnitudes less than inter-arrival time for foreground requests (100 ms)

Network Scheduler Results II



Future Work

Fixed overhead idletime scheduling:

- Present implementation is based strictly on current state and events.

  Simple operation and analysis, but optimization possible if policy for foreground tasks allows some performance reduction

➔ Immediately switch to background, if enough credits for foreground requests collected. E.g. 10 foreground → 1 background

(Long window: many credits, so limit the rate at which credits can be spent)

Automatic preemption interval adaption:

- Present implementation requires manual specification of interval length.

➔ Adapt length based on resource and work load observation.

(Event counters for states: increase on fast increase B[f], decrease on steady increase P[f], I[f])

Idletime use of storage capacity:

- Background tasks must not take away storage from foreground tasks.

➔ Kernel must be able to reclaim storage allocated to background task.

Applications using idletime service must be prepared for disappearing storage.

Conclusion

Common workloads on computer systems rarely utilize resources fully.

Idle capacity can be used for background work, if it does not interfere with foreground work.

Idle time scheduling – a generic, resource and workload independent kernel mechanism – addresses this demand.

Preemption intervals are used to minimize the cost of switching from background to foreground work.

Examples showed the feasibility of this approach.

Questions?