Predictable Interrupt Management and Scheduling in the Composite Component-based System

Gabriel Parmer and Richard West

Computer Science Department
Boston University
Boston, MA 02215

{gabep1, richwest}@cs.bu.edu

December 2, 2008

Motivation, Goals, and Challenges

Customizable/extensible base system upon which wide range of systems can be built

- with application-specific services and abstractions

System dependability challenges

- code complexity
  - complicate testing and verification
  - focus on fault tolerance
- services provided by 3rd party, untrusted developers
  - schedulers, synchronization policies, ...

Can we provide a canonical base system that is predictable and dependable?

The Composite Component-Based OS

System policies and abstractions defined in separate components (schedulers, synchronization policies, etc. . .)

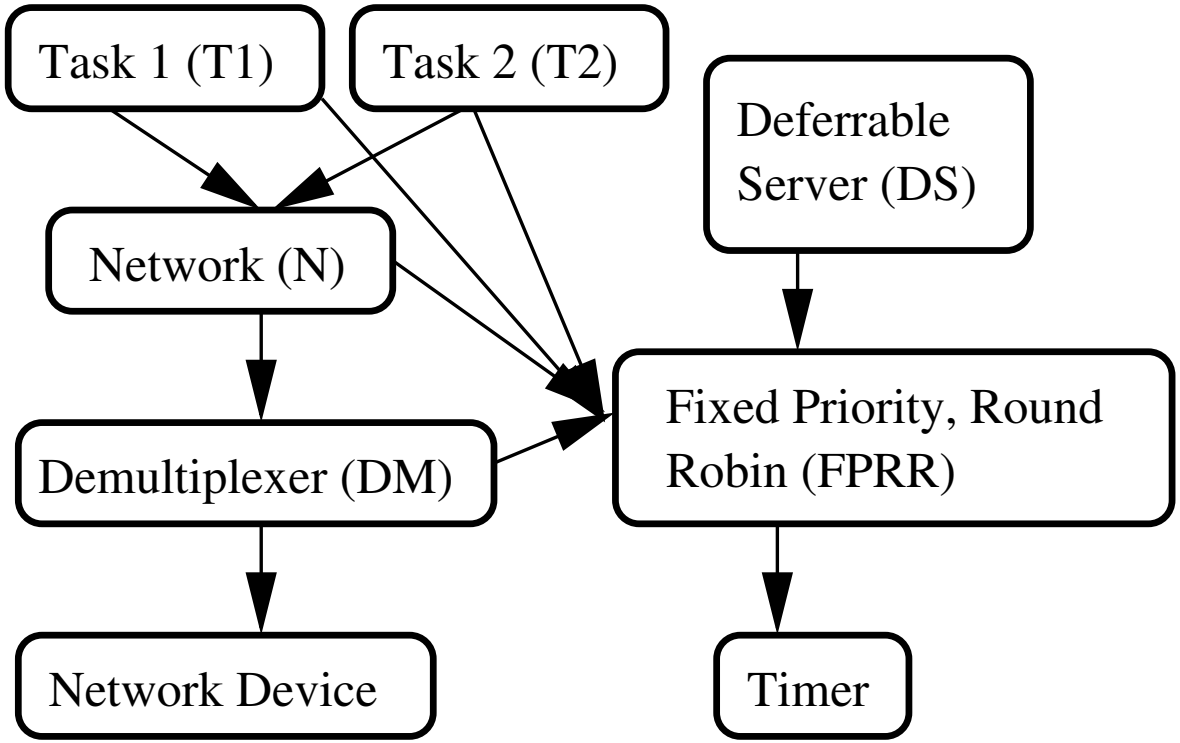
- at user-level in their own protection domains

Constrain scope of impact of component faults

Focus on

- application-specific system composition
- system-provided dependability

Simple Composite System



- components related via *dependency*

User-Level Component-Based Scheduling

Correctness of RT systems dependent on tasks' temporal behaviors

Goal: Define system scheduling policies in components

- control task and interrupt execution
- maintain accurate execution accountability
- must be efficient and predictable

Challenge: increased overhead of scheduler invocation

- inter-protection domain communication

Composite Scheduler Components

Composite kernel provides trusted communication between components

- scheduling external to kernel

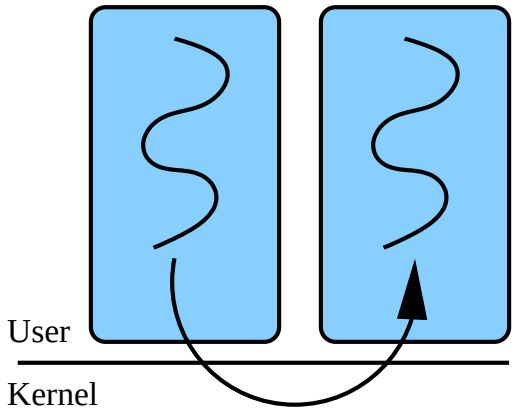
Composite includes functions to

- create a hierarchy of schedulers
- grant control of specific threads to certain schedulers

Schedulers multiplex CPU via

```
cos_switch_thread(thd_id, ...)
```

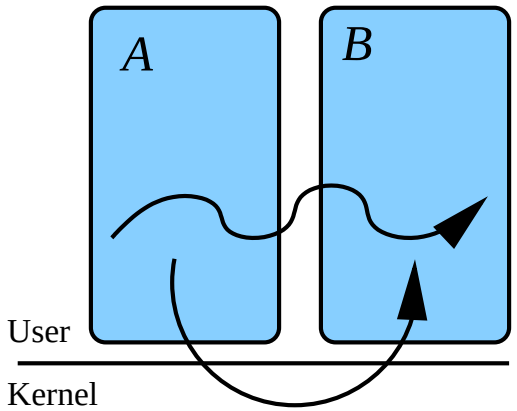
Component Invocations



Common design: threads in protection domains that communicate with each other via IPC

- each IPC includes scheduling decisions
- user-level component scheduler $\longrightarrow$ significant overhead, potentially increased WCET of a critical path

Component Invocations II



Composite: component invocation via migrating thread model

- a thread, τ executing in component A invokes B via the kernel and continues executing in B
  - τ is charged for execution in A and B
- IPC does not require scheduling decision by design

Interrupt Execution Management

Scheduling of interrupts

- interrupts promoted to threads
- run interrupt thread, or currently executing thread?
  - scheduling decision needed
- interrupt thread finishes execution
  - another scheduling decision needed

Possibility of significant overhead with user-level scheduling

- lessen effective utilization
- increase interrupt response time

Interrupt Management/Scheduling in Composite

Composite's mechanism for asynchronous "upwards" execution

- *brands*
  - a path of components to guide the asynchronous execution
  - a priority/urgency with which to schedule a corresponding upcall
- *upcalls*
  - thread, associated with a brand
  - conducts the asynchronous execution along path recorded in brand

Interrupts are *branded*, which associates them with a brand, and attempts to execute the brand's upcall

Avoiding Scheduler Invocations

Scheduling decision required when interrupt is branded

- shared data-structure between schedulers and kernel
  - scheduler posts urgency/importance of threads with kernel
  - kernel publishes CPU usage information to scheduler (cycles)
- when upcall attempted for a brand
  - kernel compares urgency/importance of current thread with brand
  - switch to upcall thread if higher urgency/importance

Avoiding Scheduler Invocations II

When upcall completes execution

- *common case:* immediately switch back to preempted thread
- *rare case:* possibly when a scheduling decision involving upcall has been made

User-Level Scheduler Synchronization

Schedulers must synchronize around critical sections

- mechanism must be efficient, predictable, policy neutral

Kernel-provided semaphores, mutexes, locks???

- but these primitives *rely* on a scheduler
  1. τ_1 acquires lock to shared run-queues $\rightarrow$
  2. τ_1 is preempted $\rightarrow$
  3. τ_2 attempts to take lock: contention $\rightarrow$
  4. kernel must know which thread to switch to $\rightarrow$
  5. invoke scheduler that synchronizes around run-queues $\rightarrow$
  6. **livelock**

Allow schedulers to disable/enable interrupts???

- not with untrusted/malicious/errant schedulers

User-Level Scheduler Synchronization II

Uncontested critical section access

- user-level solution using lock-free synchronization and *restartable atomic sections*
  - sections of assembly: if preempted in section, instruction pointer reset to start of section
  - can model atomic instructions

Contention for critical section

- library and kernel supplied wait-free synchronization
- higher-priority thread, H, "helps" lower priority thread through critical section, which then switches back to H

Microbenchmarks

- Thread migration efficient?
- Thread switching support efficient?
- Brands/upcall costs vs. scheduler invocations for interrupts

Operation	Cycles [1]
Hardware RPC Costs (U/K transitions, page tbls)	1110
Linux Pipe RPC	15367
Composite Component Invocation	1620
Linux Thread Switch	1903
Composite Base Thread Switch	529
Composite Thread Switch w/ FPRR	976
Composite Brand w/ Upcall Execution	3442
Composite Brand w/ Scheduler Invocations	9410

[1] 2.4 Ghz Pentium IV, Linux 2.6.22

Case Study: Predictable Interrupt Scheduling

Goal: predictable interrupt scheduling

Predictable task execution

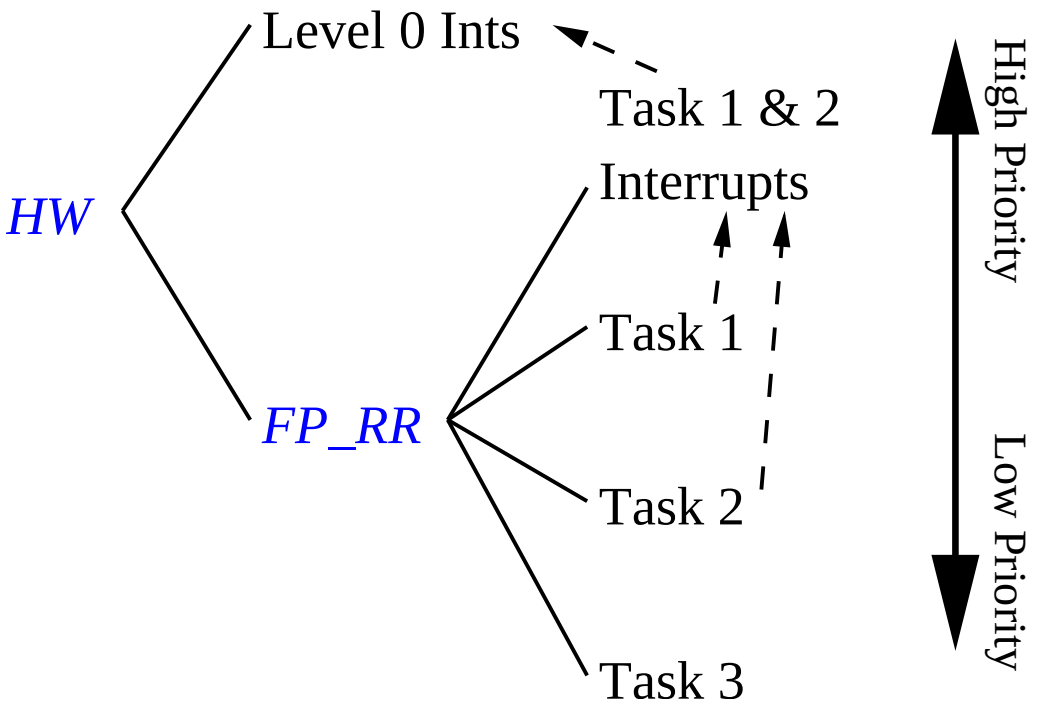
- control interrupt interference with tasks
- tasks have dependencies on interrupts!
  - a NIC interrupt executes on behalf of task that will receive that packet

Intelligent scheduling of interrupts

- demultiplexing component inspects packet contents
- brands specific to the contents of the packet
  - interrupts for different tasks are scheduled differently

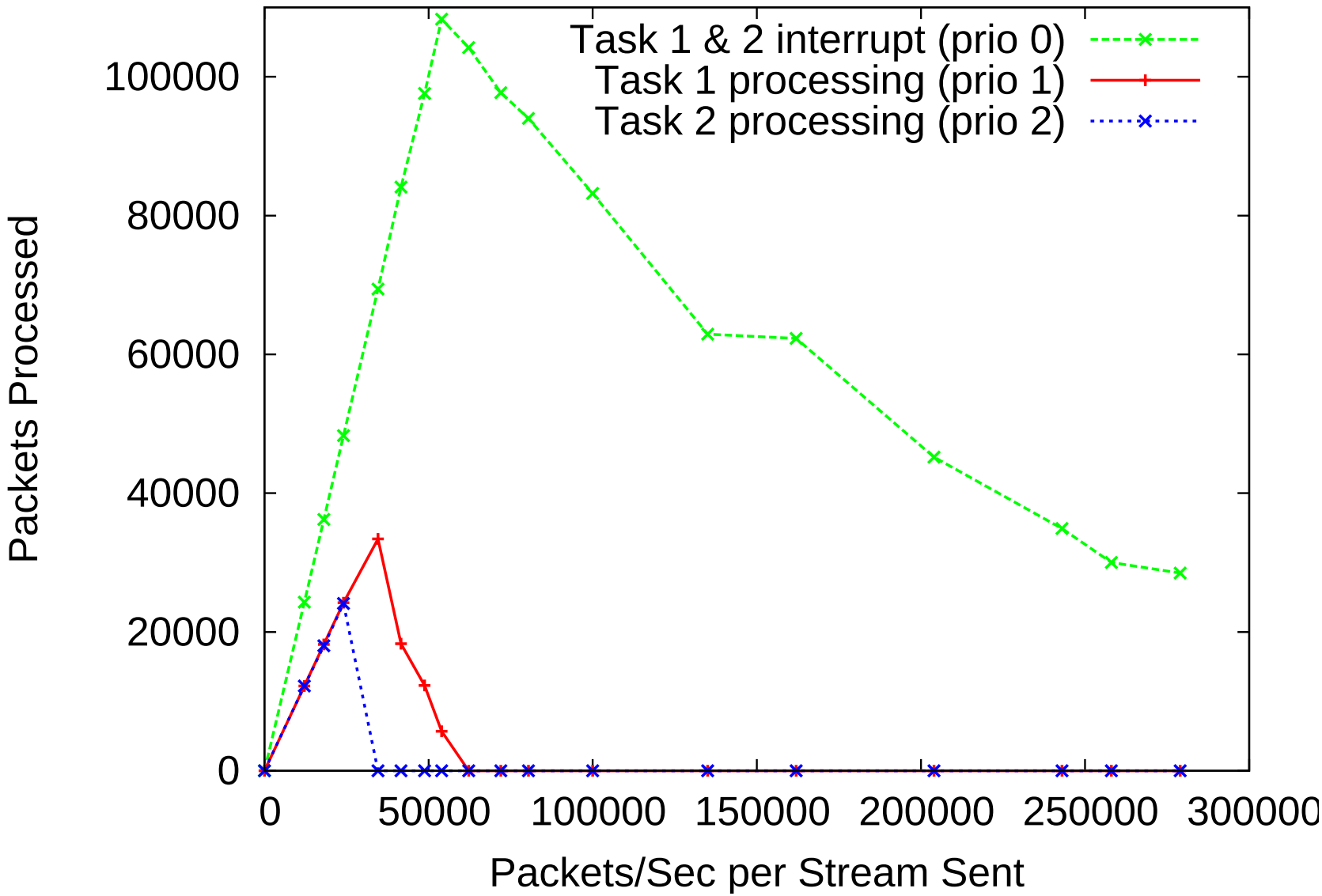
Linux-Style Interrupt Scheduling

All interrupts executed at highest priority



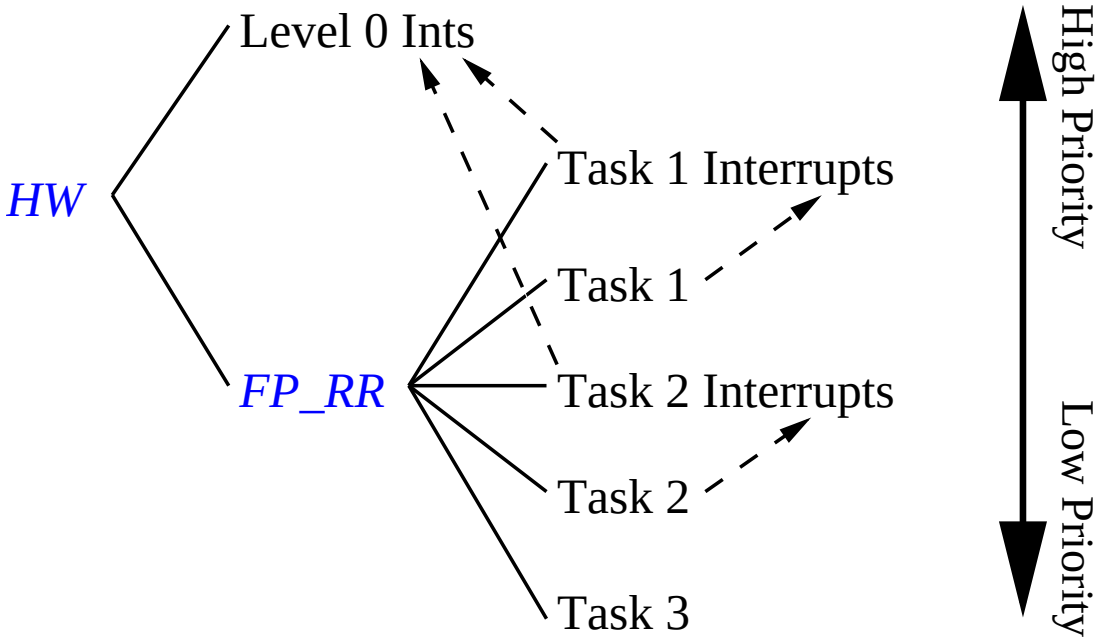
- *italic* nodes are schedulers
- dotted lines represent dependencies

Linux-Style Interrupt Scheduling II

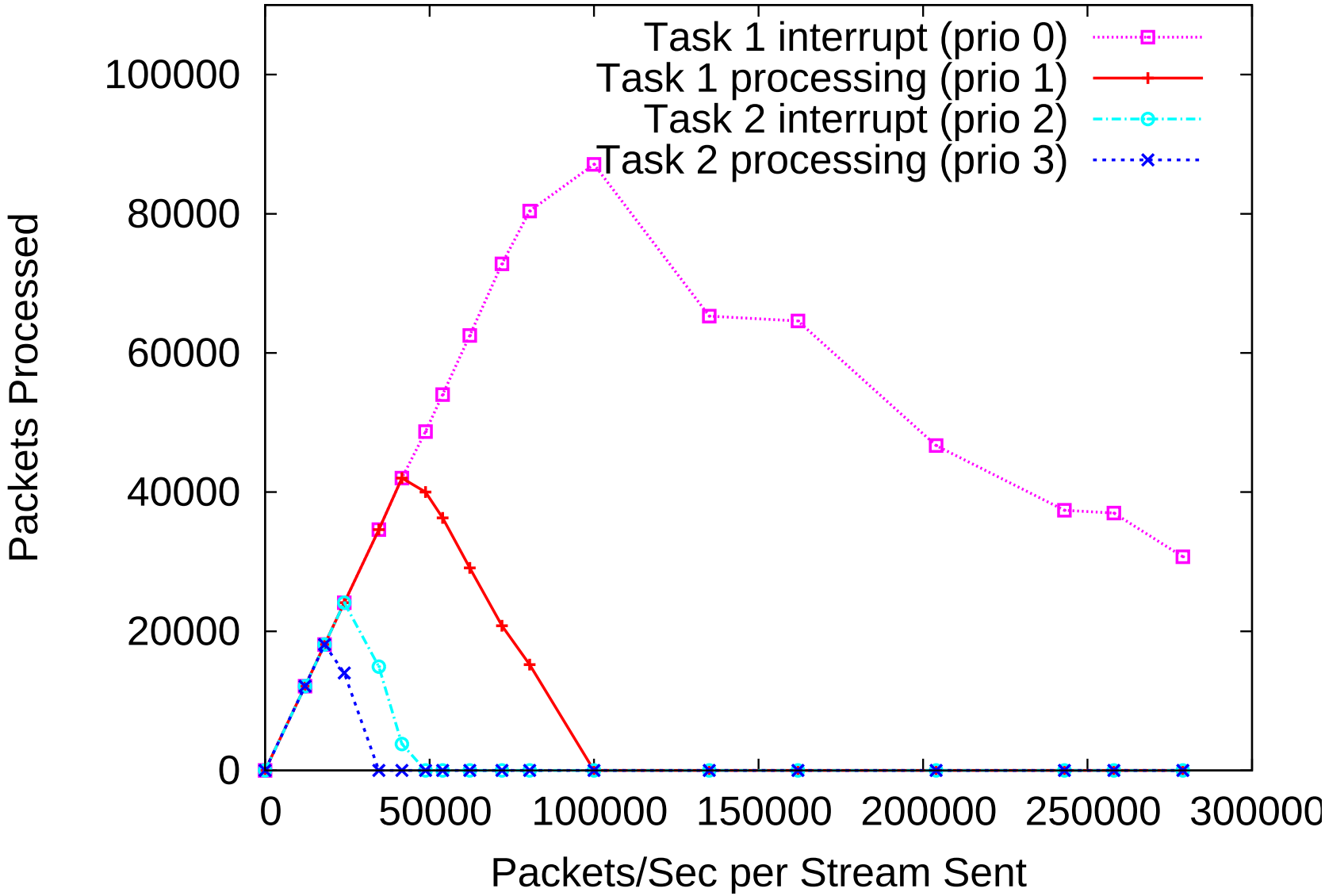


Priority Differentiation for Tasks and Interrupts I

Interrupts are executed at the priority above their associated task, but below other tasks of higher priority

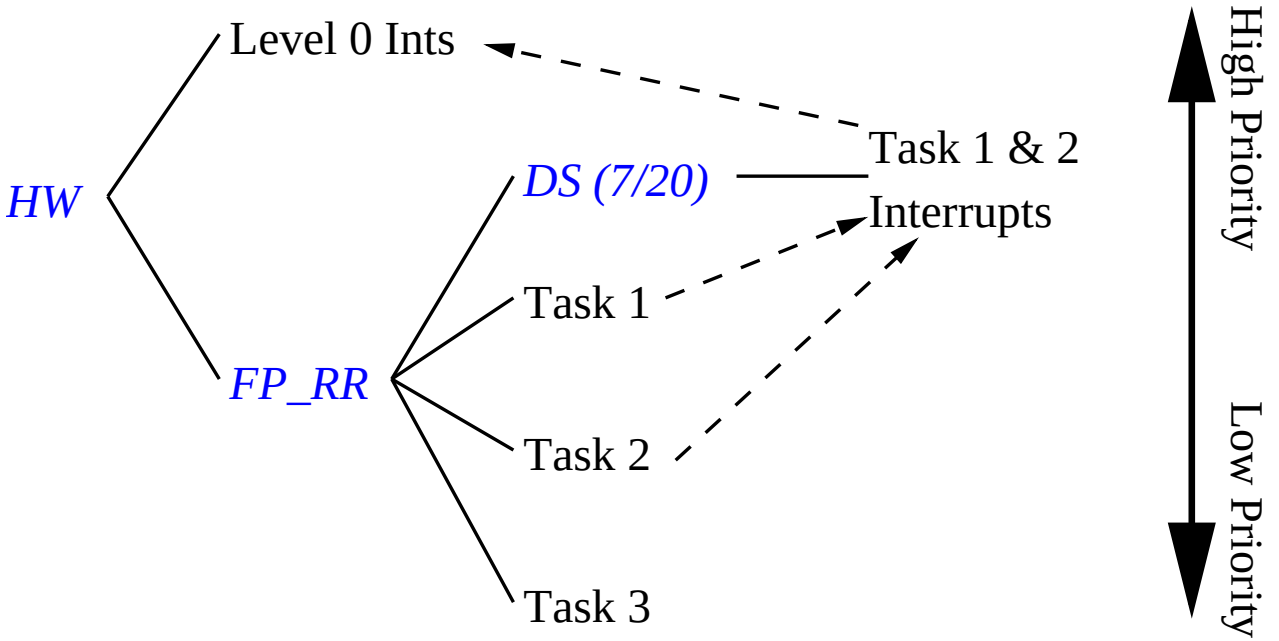


Priority Differentiation for Tasks and Interrupts II

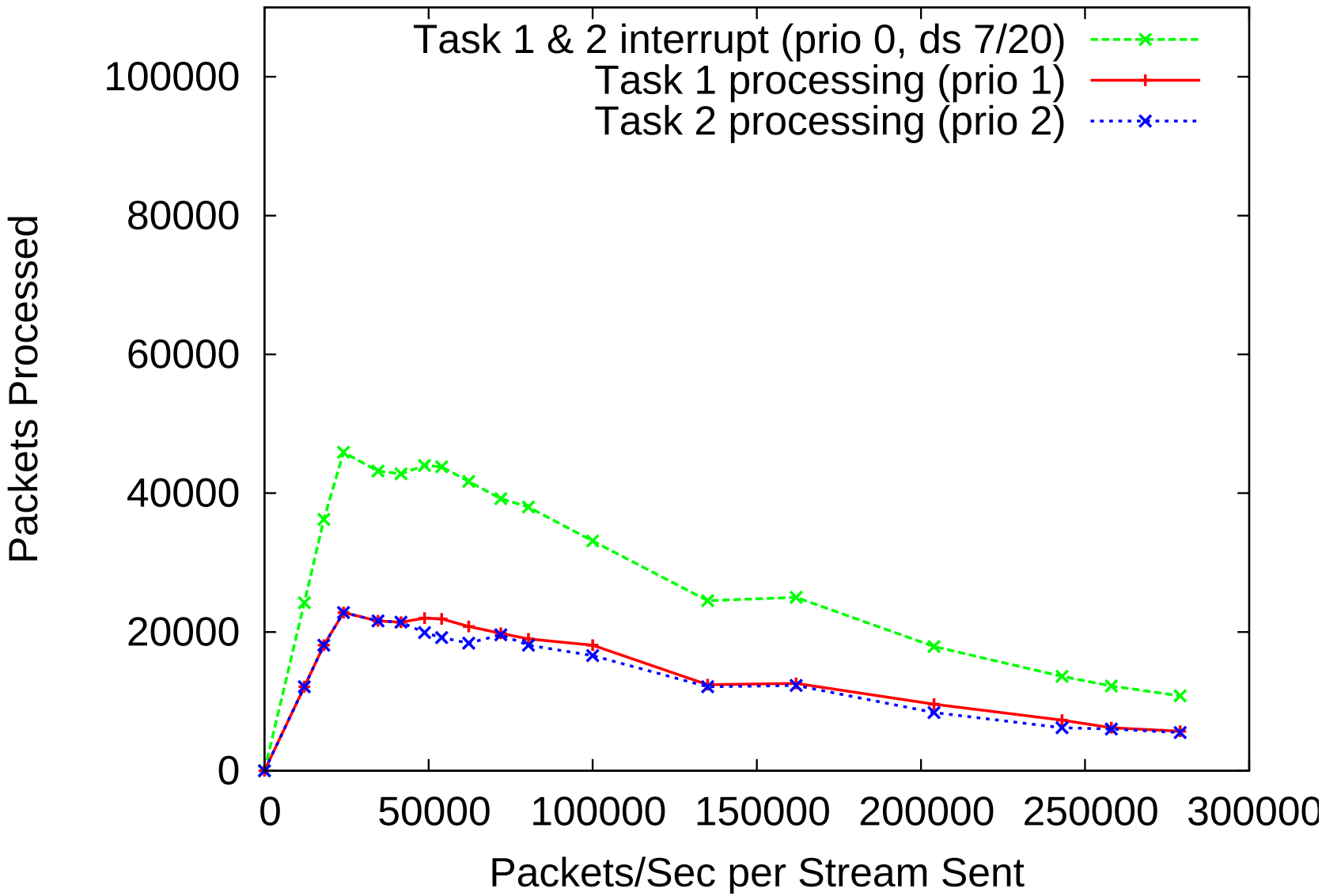


Threaded Interrupts with Deferrable Server

Interrupts are executed at a higher priority than tasks, but in an aperiodic (deferrable) server

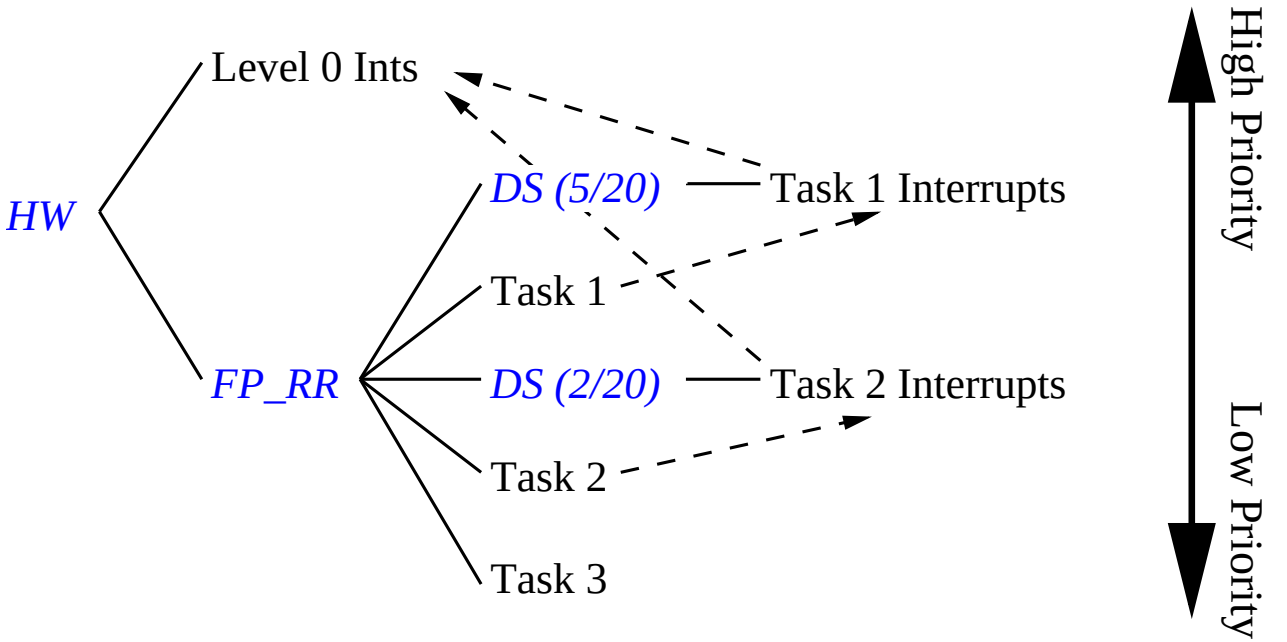


Threaded Interrupts with Deferrable Server II

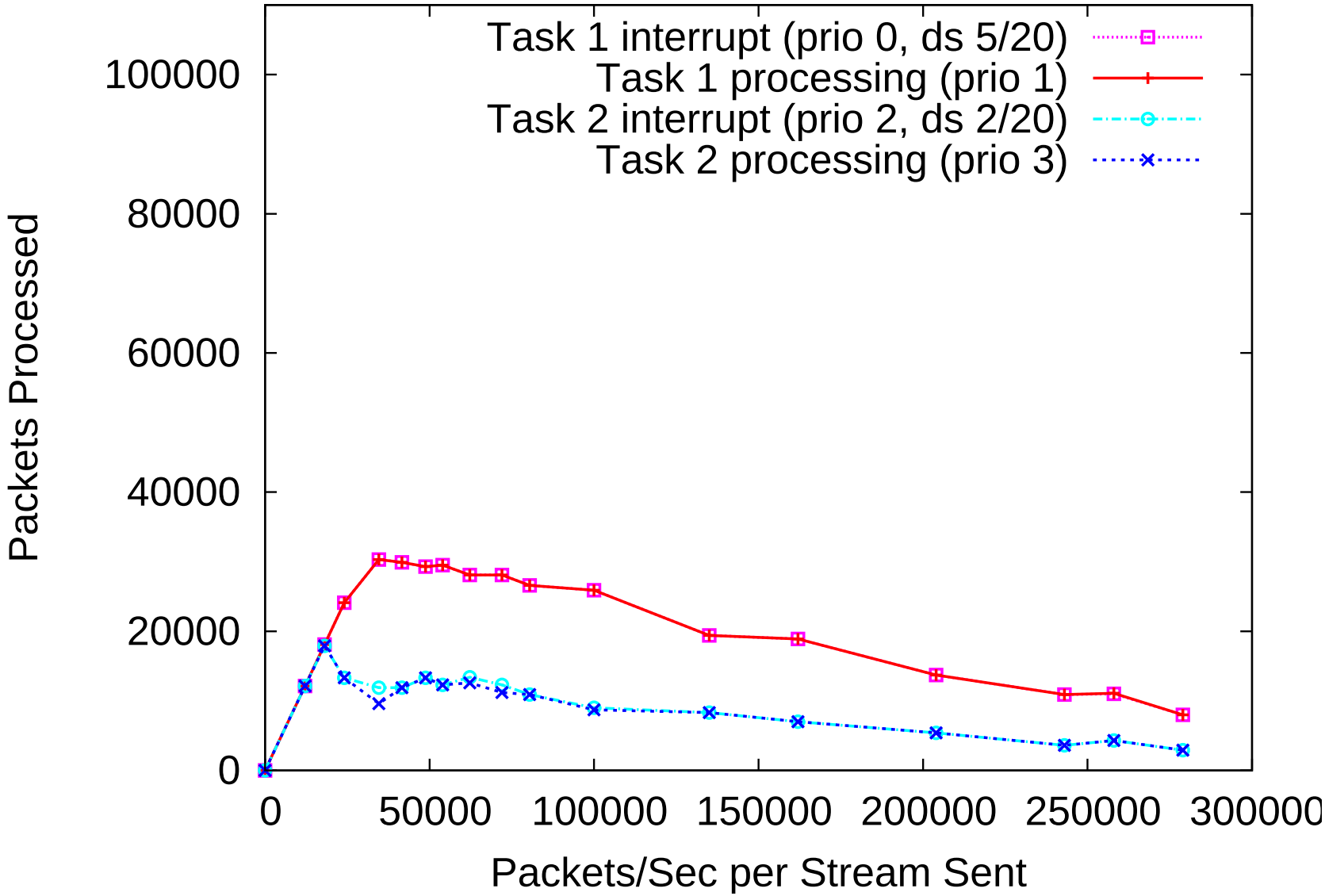


Priority Differentiation and Deferrable Servers

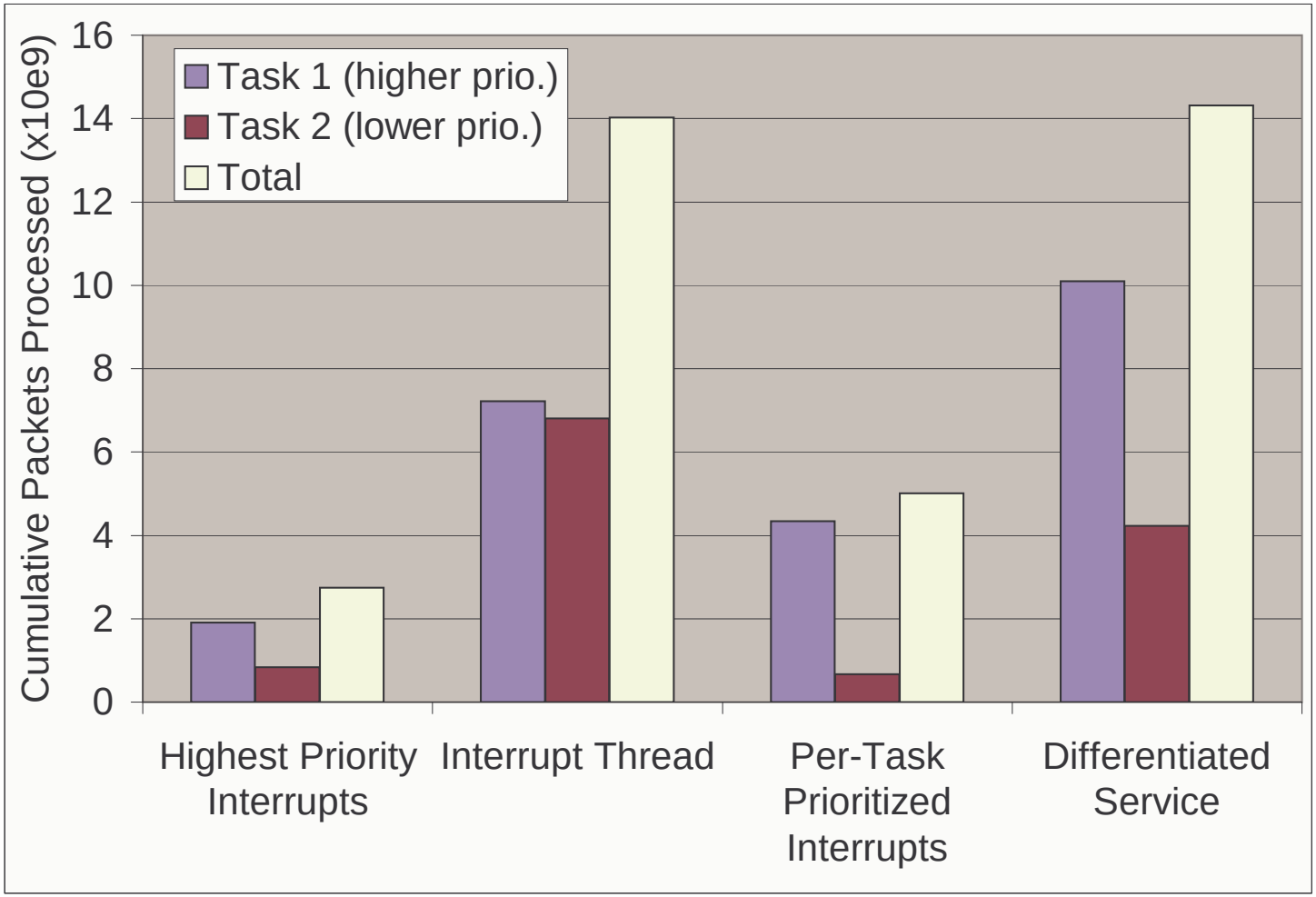
Interrupts are executed at the priority above their associated task, but below other tasks of higher priority, *and* are executed in deferrable servers with allocations commensurate with desired task progress



Priority Differentiation and Deferrable Servers II



Overall Comparison



Related Work

L4, Scheduler Activations (inc. K42), middleware-scheduling, CPU inheritance scheduling, ...

No previous work including *all* of the following

1. define complete scheduling behavior of all execution in system
2. schedulers don't have to be trusted
3. efficient even in presence of frequent interrupts

Conclusions

Component-based schedulers enable

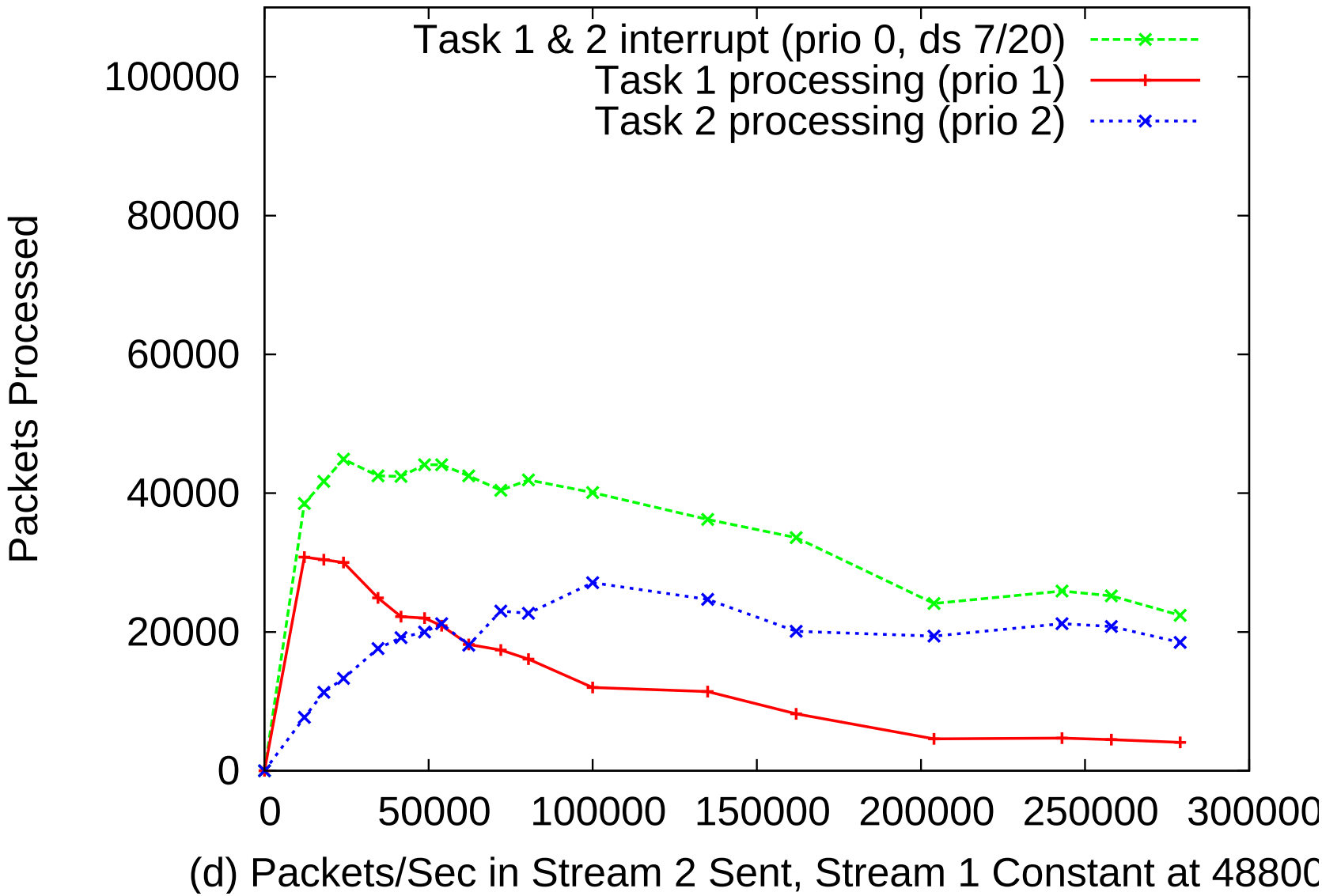
- application-specific system behavior
- high confidence system/fault tolerance

User-level component-defined scheduling policies

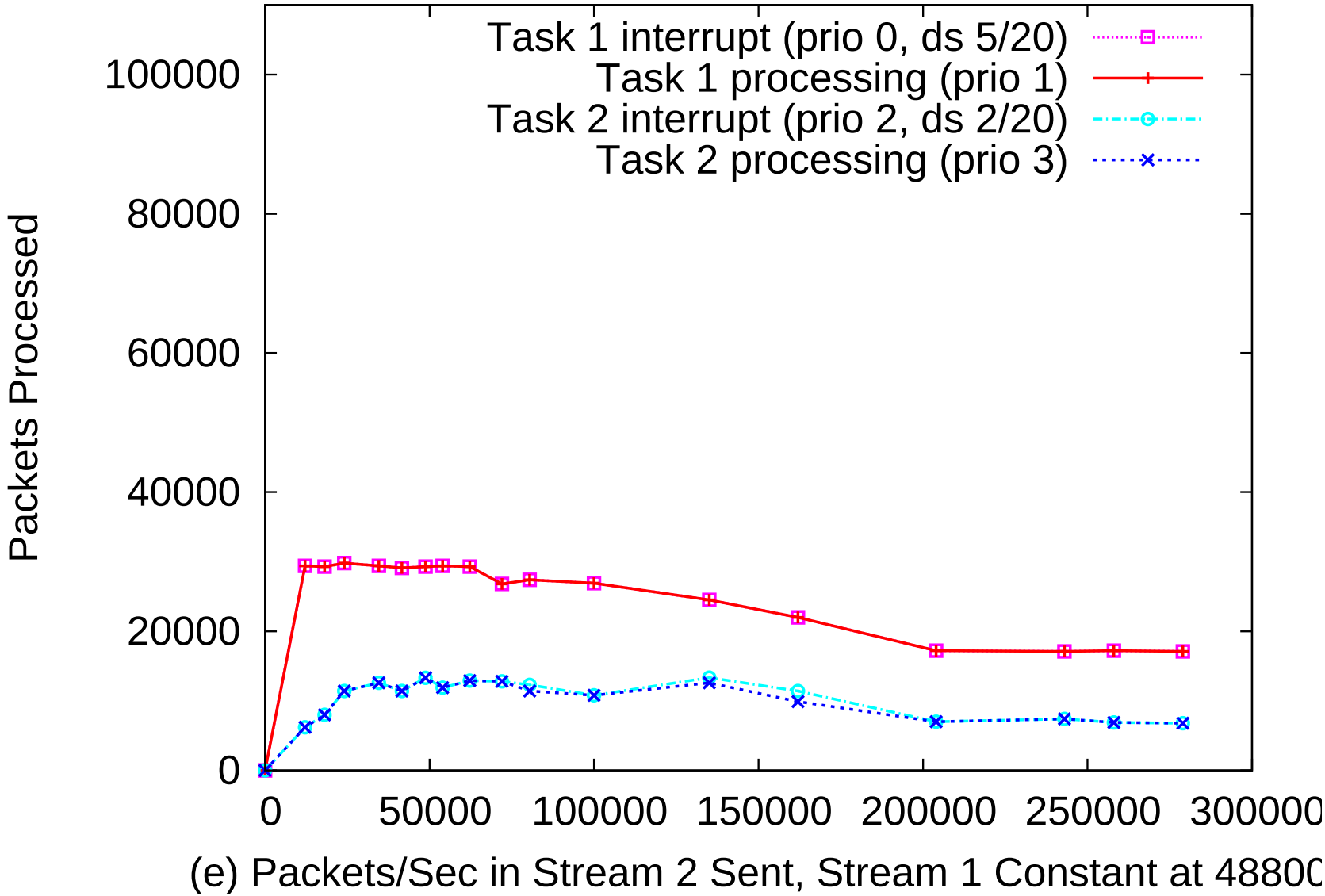
- can precisely control the system's temporal behavior
- can maintain accurate accounting information even for interrupt execution

Questions?

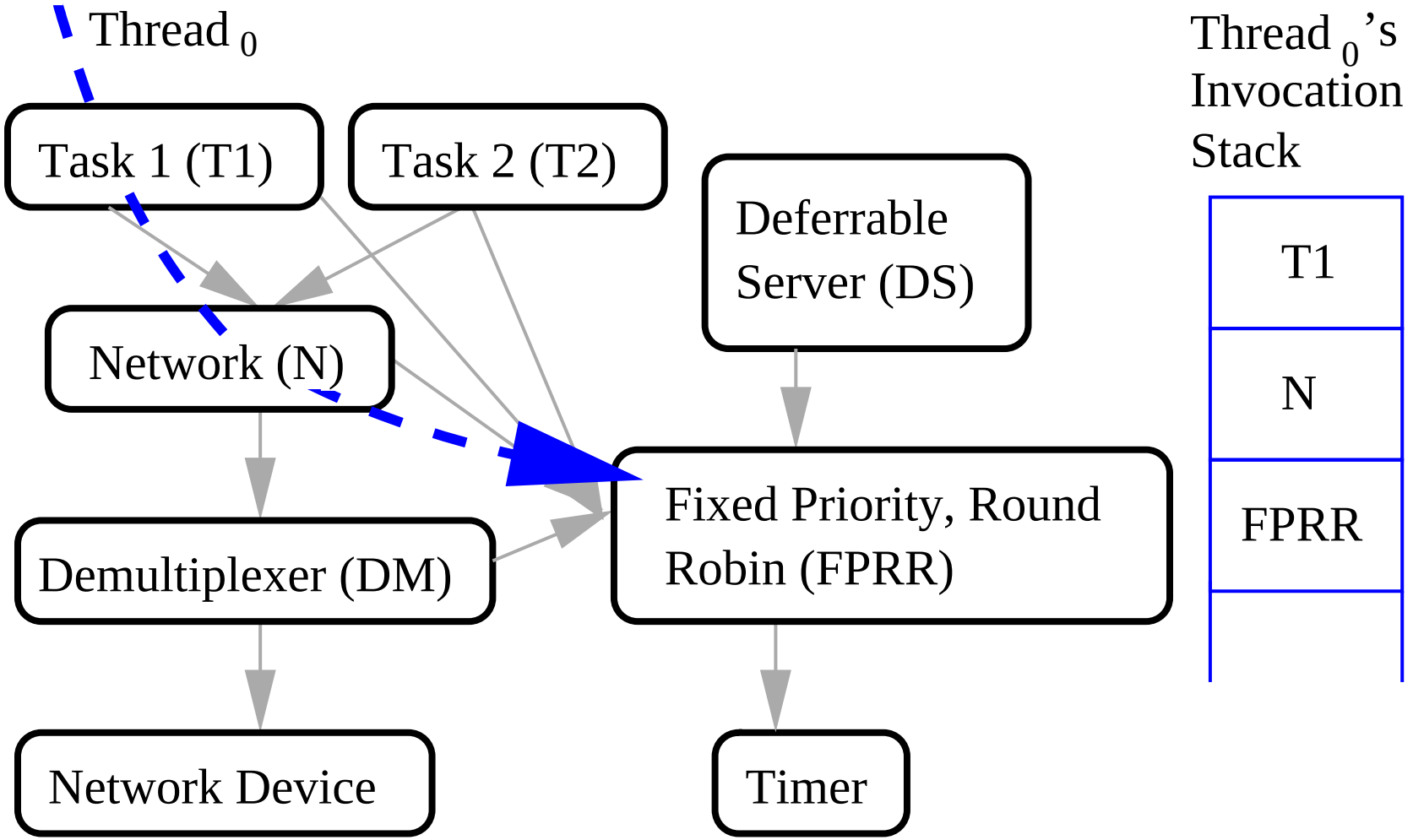
Priority Differentiation and Deferrable Servers III



Priority Differentiation and Deferrable Servers IV

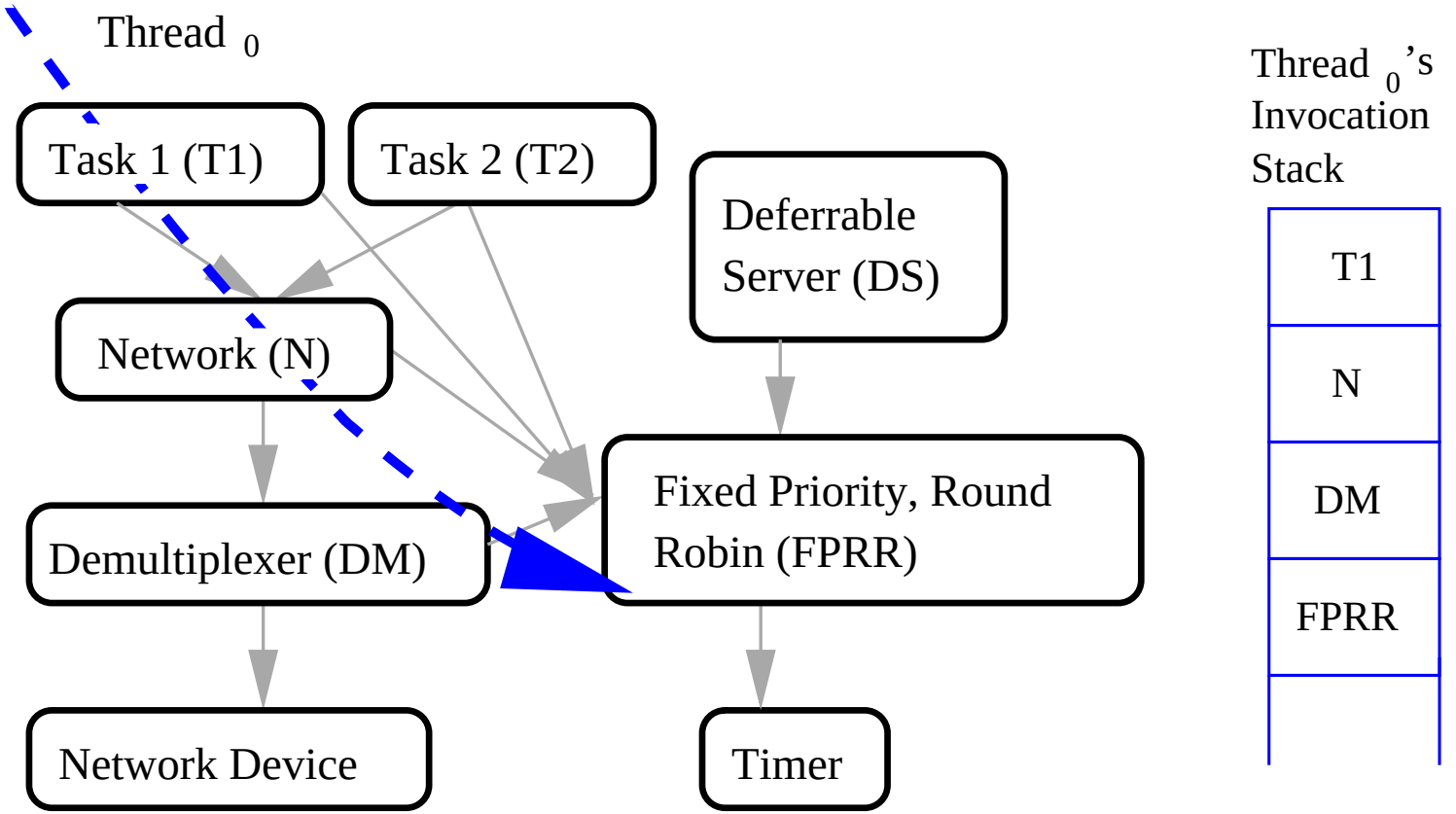


Composite Thread Migration



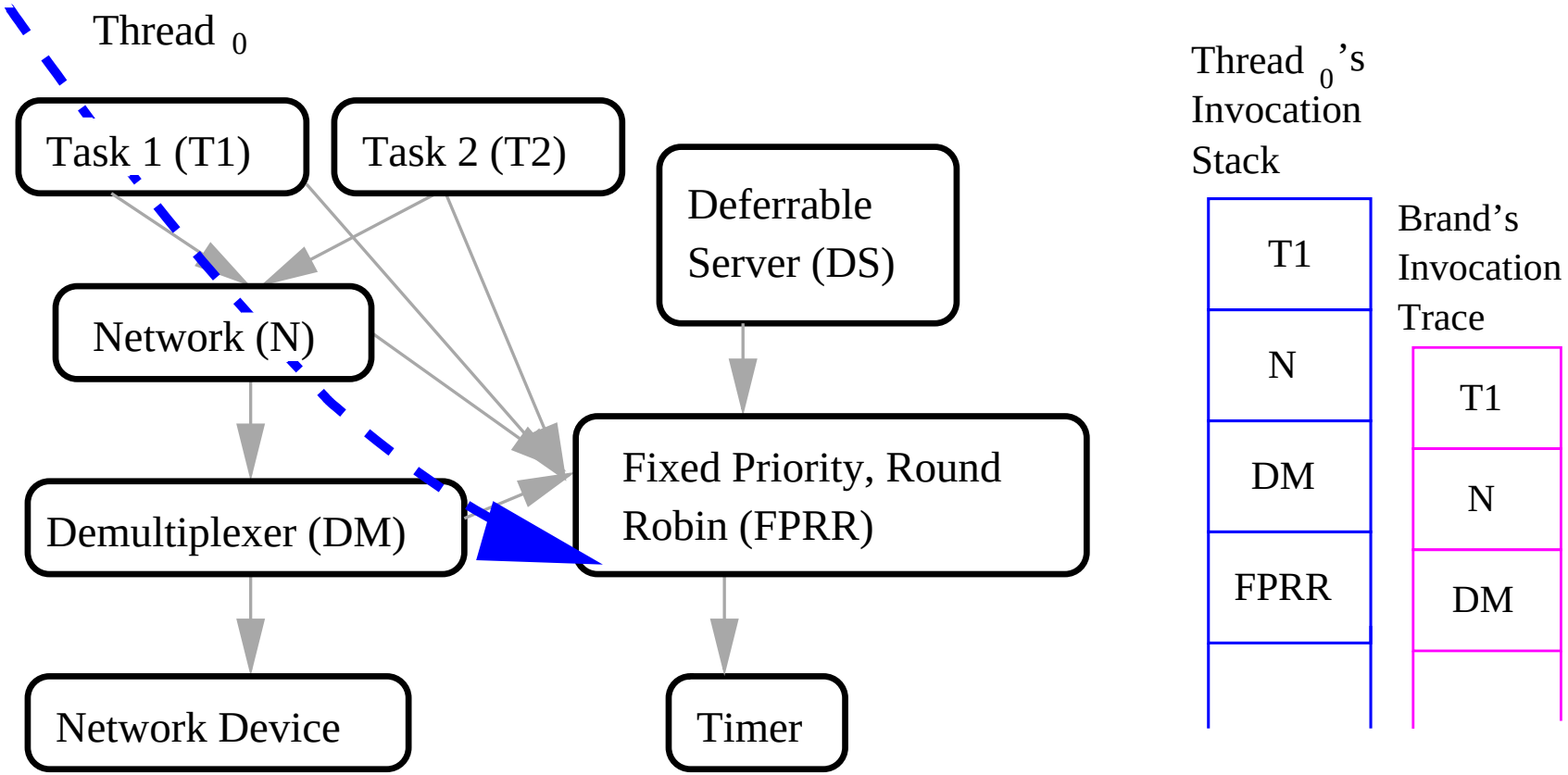
Brand Creation

1. Thread$_0$ calls scheduler (FPRR) to create a brand to be executed from DM



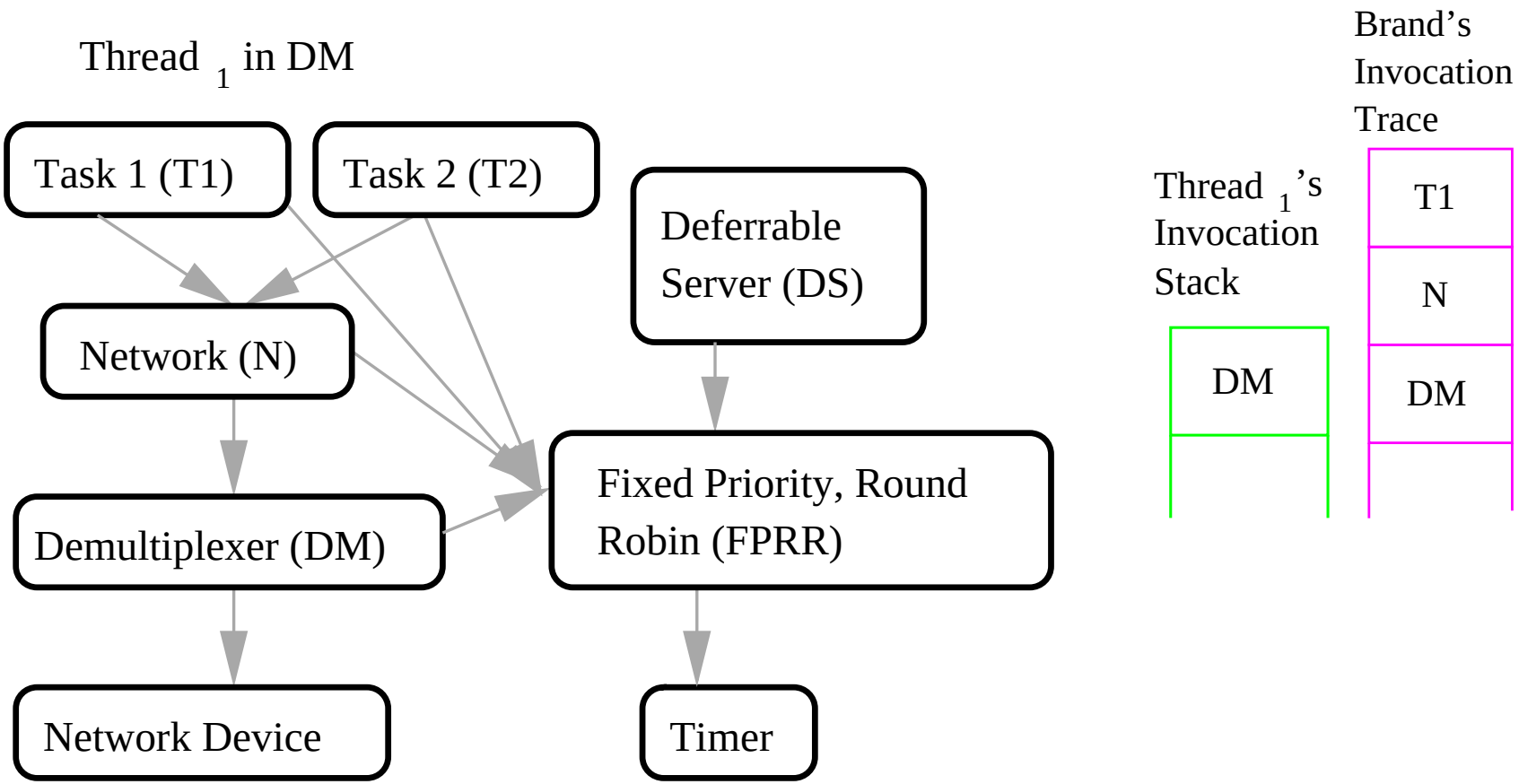
Brand Creation

1. Thread$_0$ calls scheduler (FPRR) to create a brand to be executed from DM
2. `cos_brand_cntl(BRAND_CREATE, DM)` in FPRR



Upcall Execution

1. Thread$_1$ in the Demultiplexer wishes to cause an asynchronous upcall



Upcall Execution

1. Thread$_1$ in the Demultiplexer wishes to cause an asynchronous upcall
2. Thread$_1$ executes `cos_brand_upcall(Brand, arg1, arg2)`