NATIONAL ICT AUSTRALIA

THE UNIVERSITY OF NEW SOUTH WALES

Multiprocessor/Multithreaded Real-Time Systems

Stefan M. Petters

NICTA Ltd. and CSE UNSW

smp@cse.unsw.edu.au

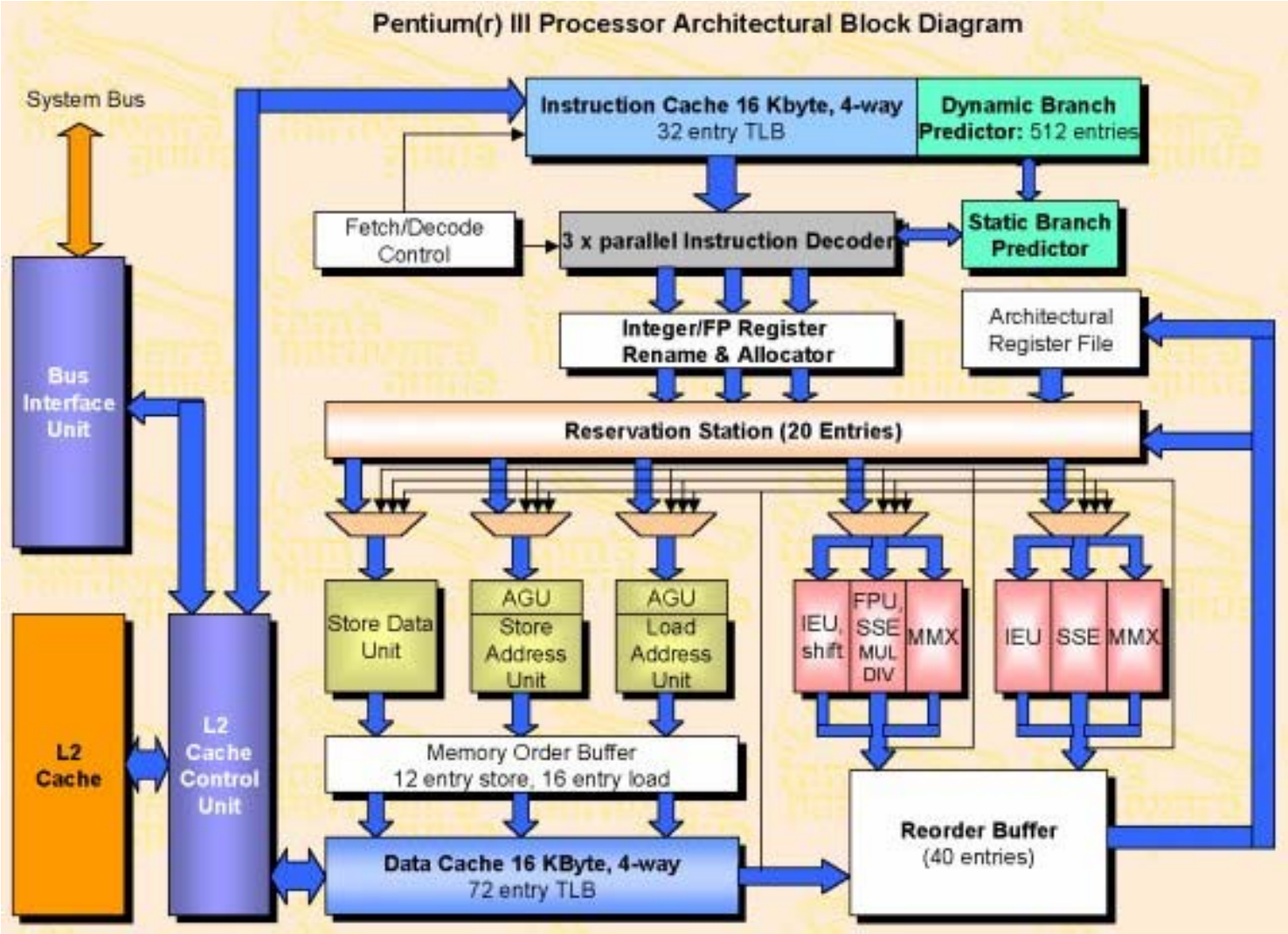
WHY

- Performance
  - Responsiveness in the presence of many external events
- Throughput
  - Managing continuous load
- Fault tolerance
  - Managing bugs, HW faults
- Reliability
  - Ensuring uptime, HW/SW upgrades …

Hardware

- Symmetric Multithreading (SMT)
  - Contention on execution units, caches, memory
- Symmetric Multiprocessor (SMP)
  - Contention on memory, cache coherency, eg NUMA
- Asymmetric Multiprocessor
  - Specialised units, coherency
- Distributed System
  - Latency in communication, loosely coupled

SMT



Almost an SMT:

Image taken from http://www.tommesani.com/images/P3Architecture.jpg

SMP

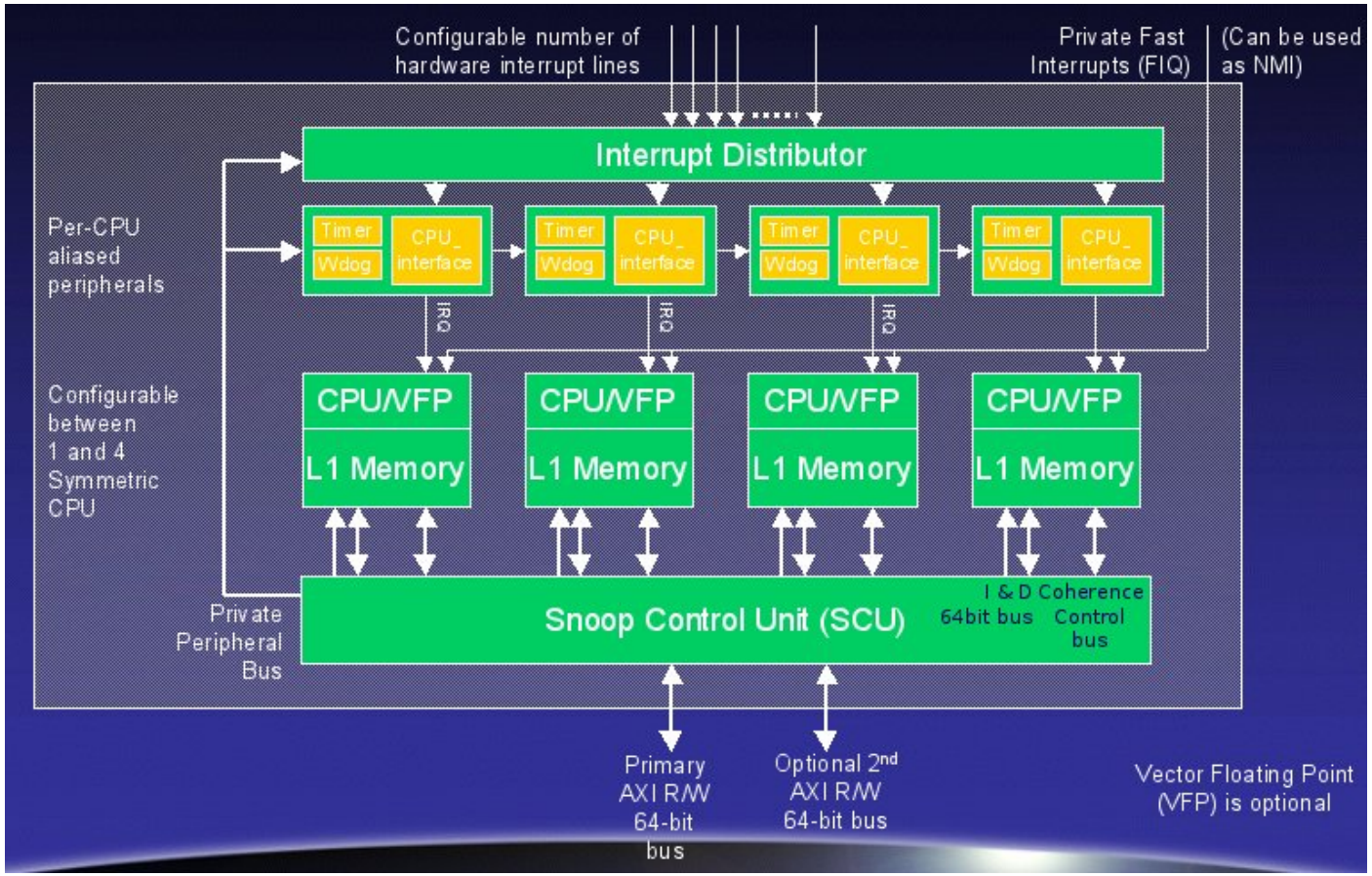
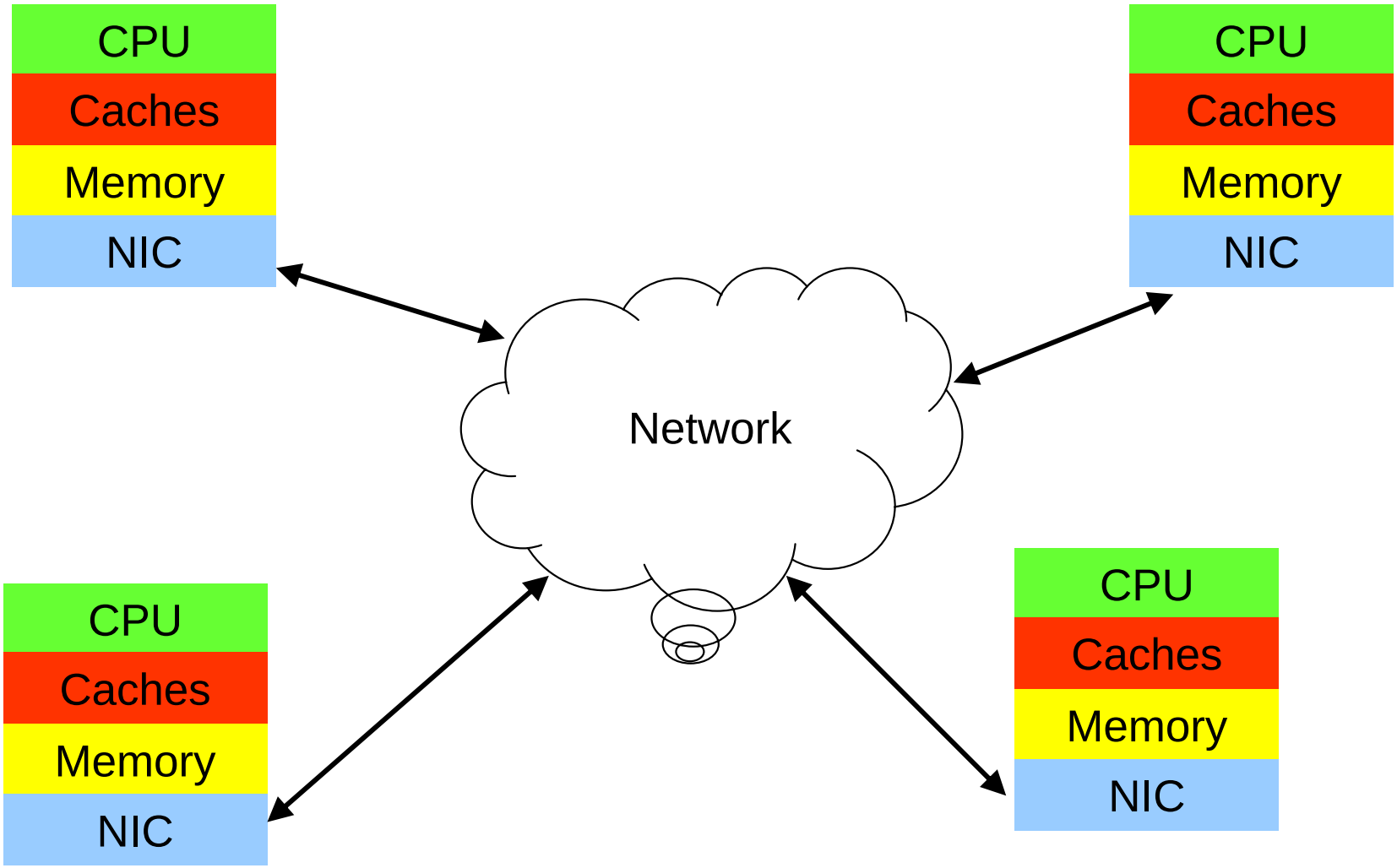


Image taken from http://linuxdevices.com/files/misc/arm-mpcore-architecture-big.jpg

Distributed System



Issues

- Resource contention
  - Execution units
  - Caches
  - Memory
  - Network
- Adding a CPU does not help
  - Example double the load, 2 instead of 1 CPU

Solutions??

- Partitioning
  - Resource contention still there!
  - Assignment using heuristics
- Non partitioning
  - mostly theoretical so far
  - Assumptions:
    - Zero preemption cost
    - Zero migration cost
    - Infinite time slicing
  - Don’t translate into reality
  - Acceptance test and no task migration a way to make it work

Solutions??

- Quite often non-preemptive
  - Fewer context switches
  - Reasoning is easy
    - IEEE Computer reference to insanity
  - Testing is easier??
  - Reduce need for blocking
- But!

Non-Preemptive

- But!!!
  - Less efficient processor use
  - Anomalies: response time can increase with
    - Changing the priority list
    - Increasing number of CPUs
    - Reducing execution times
    - Weakening the precedence constraints
  - Bin packing problem NP hard
  - Theoretically: time slicing into small quantums (PFAIR), but practically useless, as preemption and task migration overhead outweigh gains of Multiprocessors.

And now?

- No global solution.
- Partitioning and reducing it to single CPU problem good, but still contention of resources.
- Next step: After figuring out how to do the scheduling, what about preemption delay?
- Industry works with SMP/SMT, but most often on a very ad hoc basis.
- Active and unsolved research area
- Why does it work on non-RT?
  - Running the “wrong” task is not critical.

Integrating Real-Time and General-Purpose Computing

Many thanks to: Scott A. Brandt
University of California, Santa Cruz

Real-Time vs. General-Purpose OS

- Real-time and general-purpose operating systems implement many of the same basic operations
  - Process mgmt., memory mgmt, I/O mgmt, etc.
- They aim for fundamentally different goals
  - Real-time: Guaranteed performance, timeliness, reliability
  - General-purpose: Responsiveness, fairness, flexibility, graceful degradation, rich feature set
- They have largely evolved separately
  - Real-time system design lags general-purpose system design by decades
- They need to merge

Why?

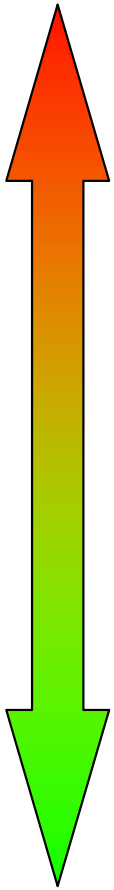
- We want both flexible general-purpose processing and robust real-time processing
  - Multimedia is ubiquitous in general-purpose systems
  - Real-time systems are growing in size and complexity
  - Look at the popularity of RTLinux
- Such systems are possible
  - GP hardware has grown powerful enough to support traditional hard real-time tasks (multimedia, soft modems, etc.)
  - Windows, MacOS, etc., are already headed in this direction
- Existing solutions are *ad hoc*
  - RTLinux, MacOS, Windows?
- The world is already headed that way
  - Microsoft, HP, Intel, Dell all want to develop integrated home systems
  - Complex distributed real-time systems do more than hard real-time
- We need to get out in front and lead the way

How?

- We need integrated solutions for each type of resource
  - CPU, storage, memory, network, …
- They must be hard real-time at their core
  - This is the only way to guarantee the hardest constraints
- They must provide native hard real-time, soft real-time, and best-effort support
  - SRT and BE support cannot be added as an afterthought
  - Neither can HRT
- We need an overall model for managing the separate resources
  - Each process must be able to specify it's per-resource constraints
  - Defaults should be reasonable, and helpful

Kinds of Timeliness Requirements

Constrained



Unconstrained

- Hard Real-Time (HRT) [e.g. flight control]
  - Hard deadlines, WCET
- Rate-Based (RB) [e.g. desktop audio]
  - Continuous processing requirements
- Soft Real-Time (SRT) [e.g. desktop video]
  - Non-critical deadlines and/or variable processing needs, worst-case, average-case, or no estimates
- Best Effort (BE) [e.g. editor or compiler]
  - Undefined timeliness requirements

- We want to run processes with different timeliness requirements in the same system
  - HRT, RB, SRT, and BE
- Existing schedulers largely provide point solutions:
  - HRT **or** RB **or** one flavor of SRT **or** BE
- Hierarchical scheduling is a partial solution
  - Allows apps with a variety of timeliness requirements, BUT
  - Static, inflexible hierarchies

- Goal: Uniform, fully dynamic integrated real-time scheduling
  - Same scheduler for all types of applications

Separate Resource Allocation and Dispatching

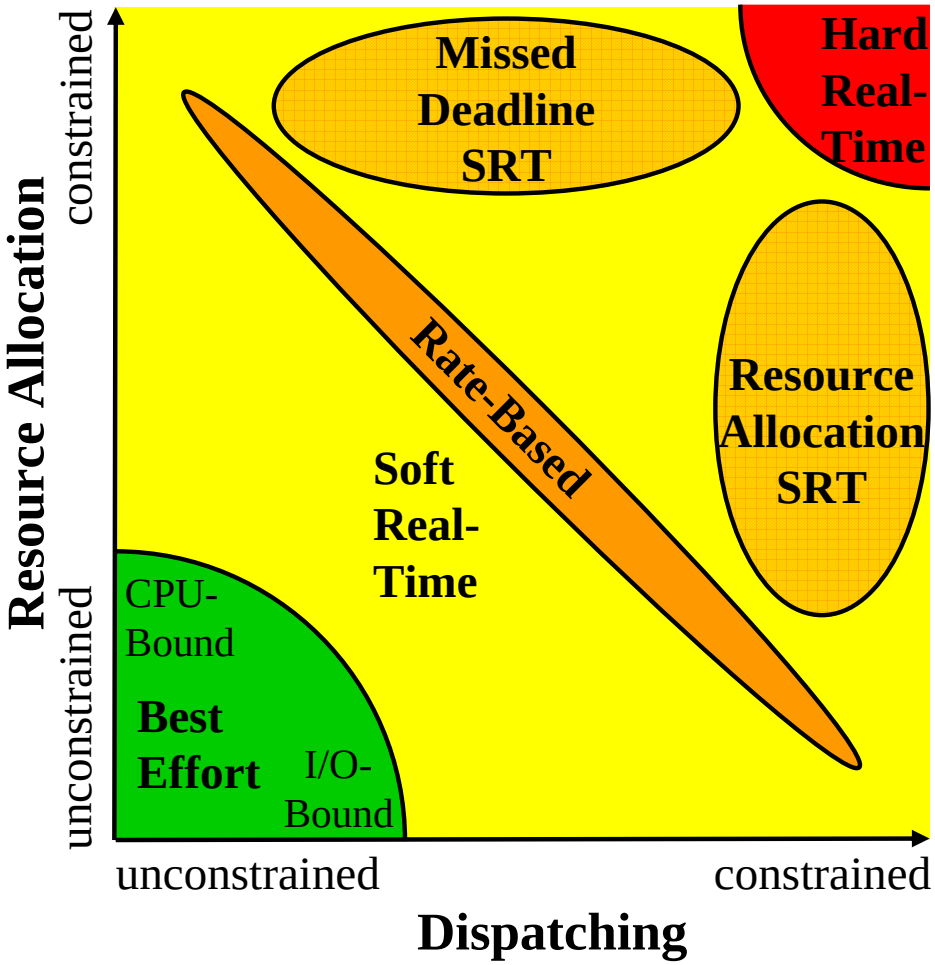
- Observation: Scheduling consists of two distinct questions:

  Resource allocation

  – *How much* resources to allocate to each process

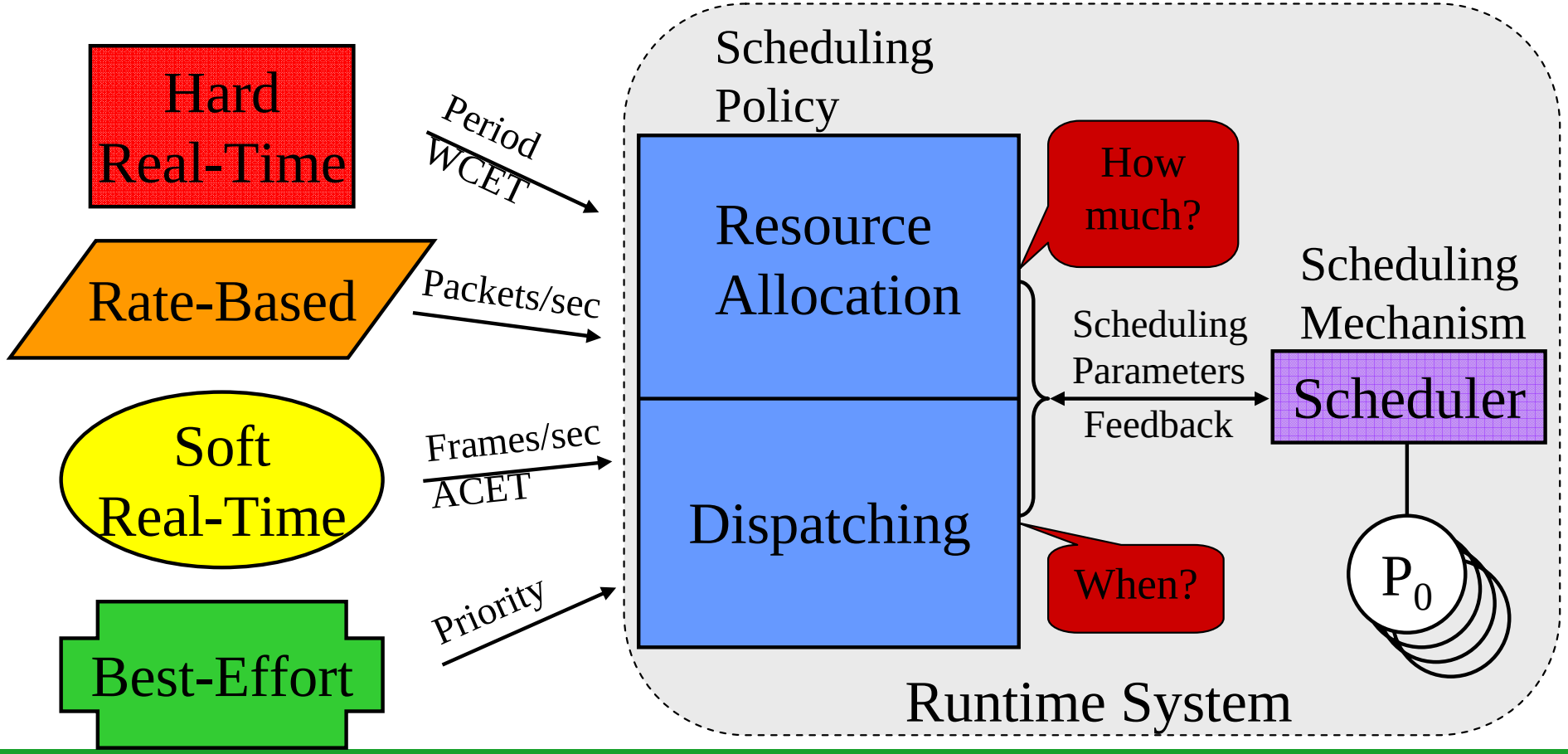
  Dispatching

  – *When* to give each process the resources it has been allocated
- Existing schedulers integrate their management
  – Real-time schedulers implicitly separate them somewhat via job admission



The (RAD) Scheduling Model

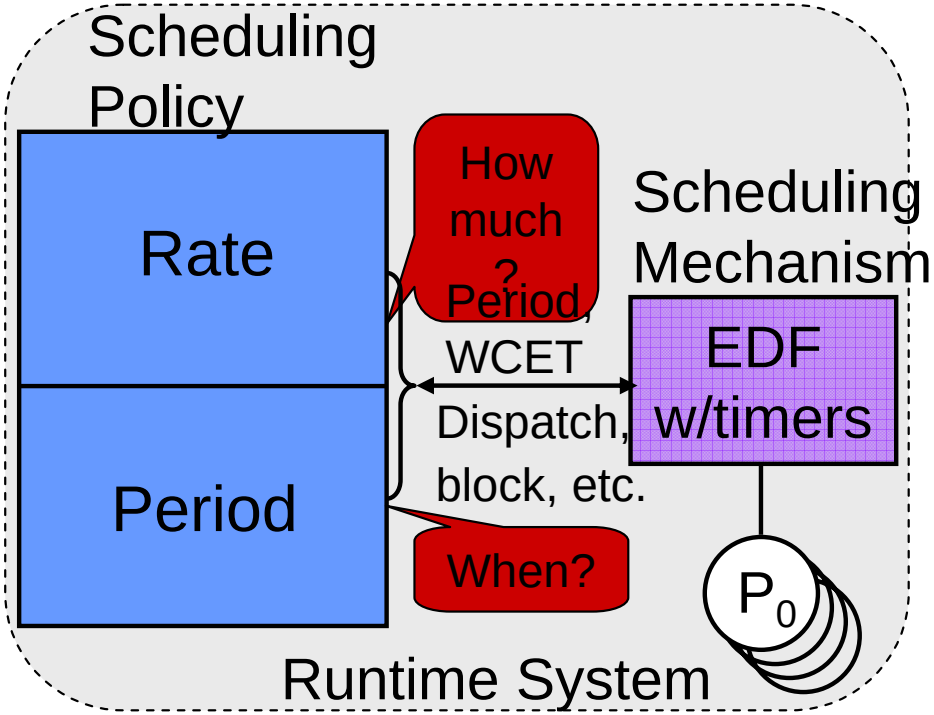
- Separate management of Resource Allocation and Dispatching
  - and separate policy and mechanism



Rate-Based Earliest Deadline Scheduler

- Basic Idea
  - EDF provides hard guarantees
  - Varying rates and periods provide flexibility
  - Programmable timer interrupts guarantee isolation between processes
- RBED policy
  - Resource allocation: Target rate-of-progress for each process ($\Sigma \leq 100\%$)
  - Dispatching: Period based on process timeliness needs
- RBED mechanism
  - Rate-Enforcing EDF: EDF + programmable timer interrupts

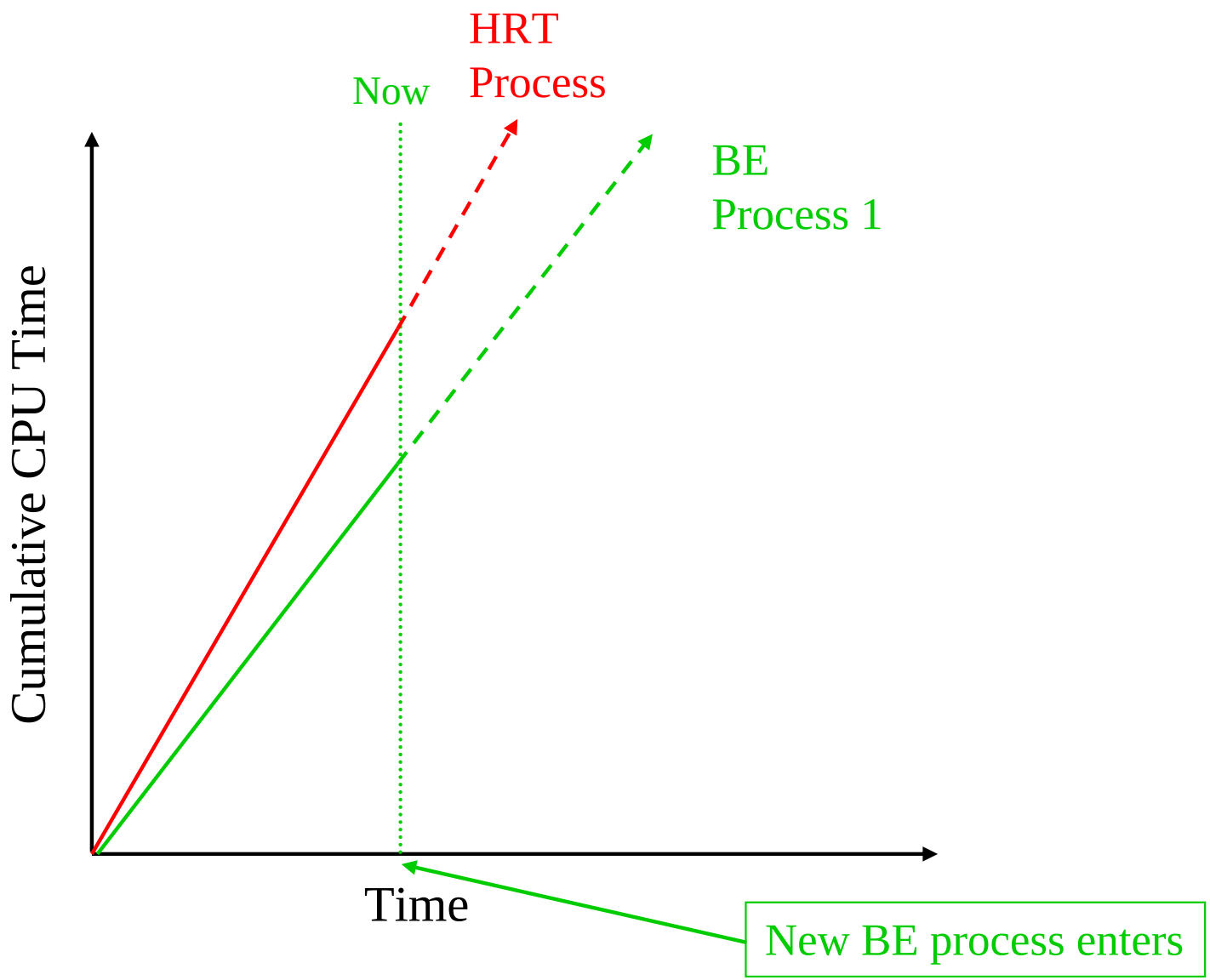
RBED: RAD Scheduler using *rate* and *period* to control resource allocation and dispatching



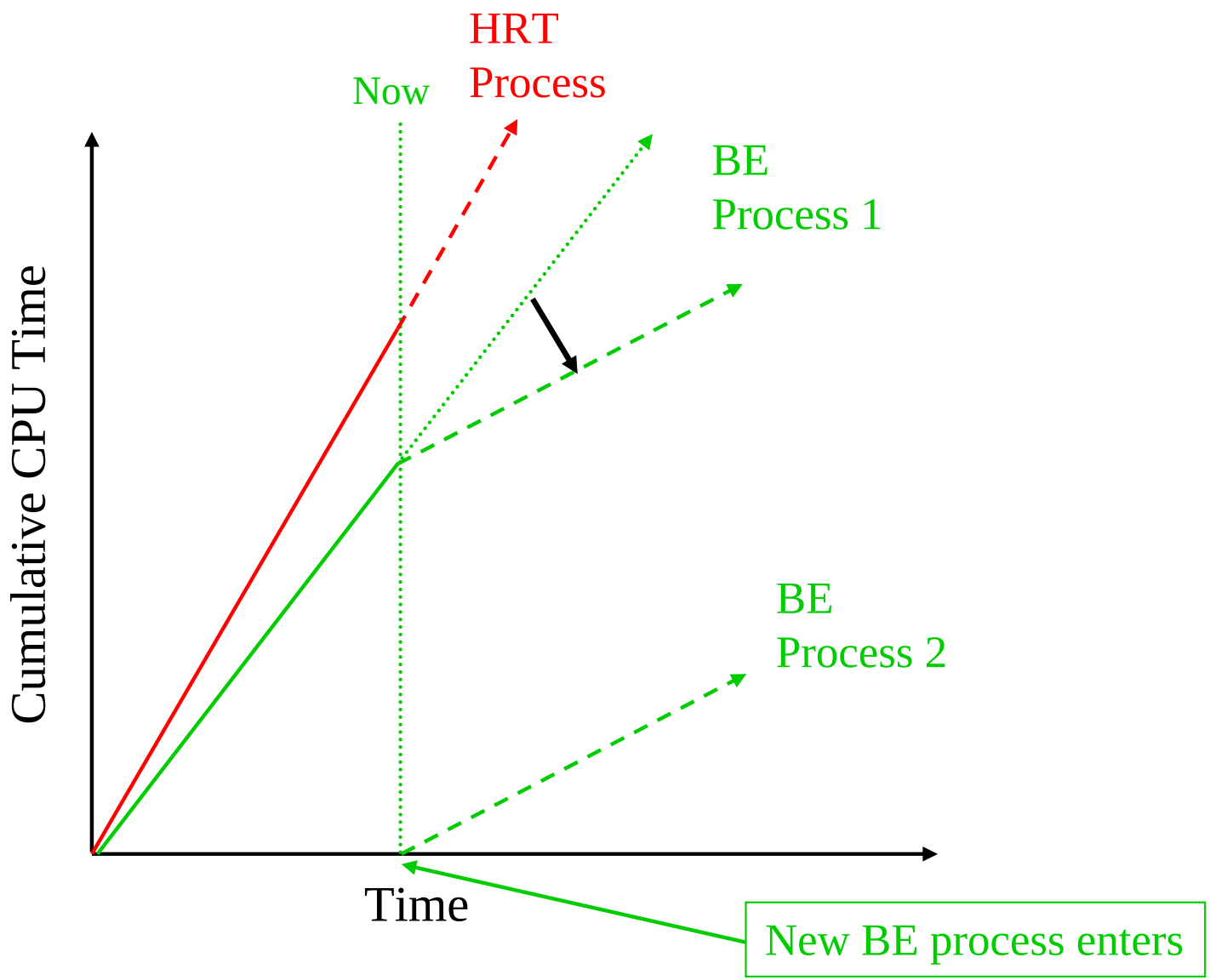
rate = utilization

WCET = rate*period

Adjusting Rates at Runtime



Adjusting Rates at Runtime



RBED Periodic Task Model

1

EDF

- Period and WCET are specified per task
  - T_i has sequential jobs $J_{i,k}$
  - $J_{i,k}$ has release time $r_{i,k}$, period $\mathbf{p_i}$, deadline $d_{i,k}$
  - $r_{i,k} = d_{i,k-1}$, and $d_{i,k} = r_{i,k} + \mathbf{p_i}$
  - $\mathbf{u_i} = \mathbf{e_i}/\mathbf{p_i}$ and $U = \Sigma \mathbf{u_i}$

RBED

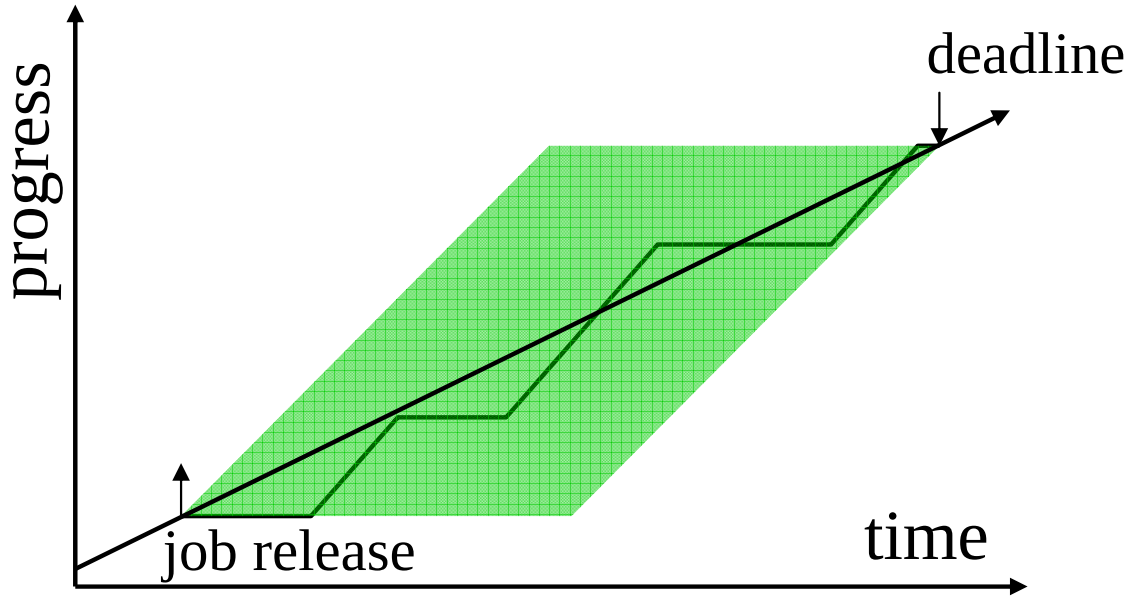
- Period and WCET are specified per job
  - T_i has sequential jobs $J_{i,k}$
  - $J_{i,k}$ has release time $r_{i,k}$, period $\mathbf{p_{i,k}}$, deadline $d_{i,k}$
  - $r_{i,k} = d_{i,k-1}$, and $d_{i,k} = r_{i,k} + \mathbf{p_{i,k}}$
  - $\mathbf{u_{i,k}} = \mathbf{e_{i,k}}/\mathbf{p_{i,k}}$ and $U = \Sigma \mathbf{u_{i,k}}$

- Theorem 1: *EDF is optimal under the new task model*
  - Corollary: *A new task may enter the system at any time, as long as resources are available for it*

Two Observations

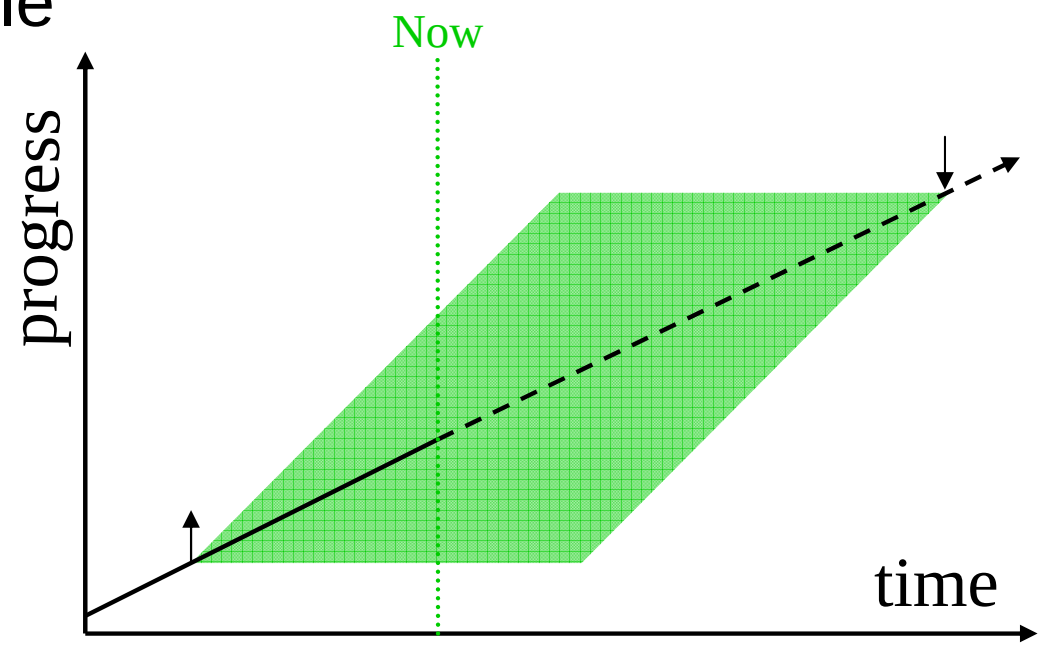
1

- At deadlines, a task's actual resource allocation is equal to its target resource allocation
- Actual resource allocation is bounded to the *feasible region*



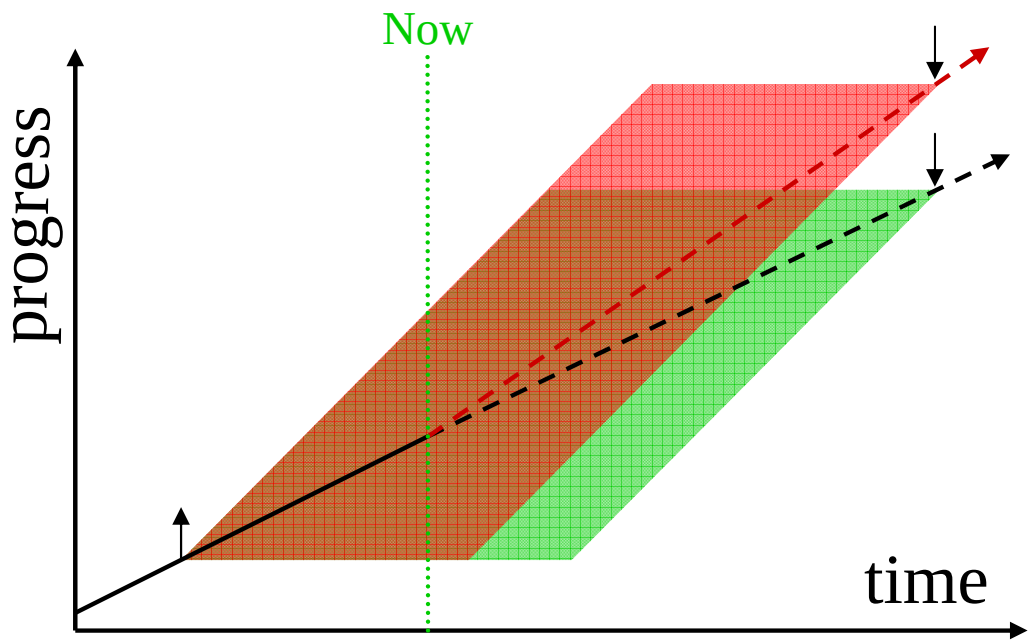
Increasing Rate (= increasing WCET)

- Theorem 2: *The resource usage of any task can be increased at any time, within the available resources*
  - Given a feasible EDF schedule, at any time task T_i may increase utilization by any amount up to $1-U$ without causing any task to miss deadlines in the resulting EDF schedule



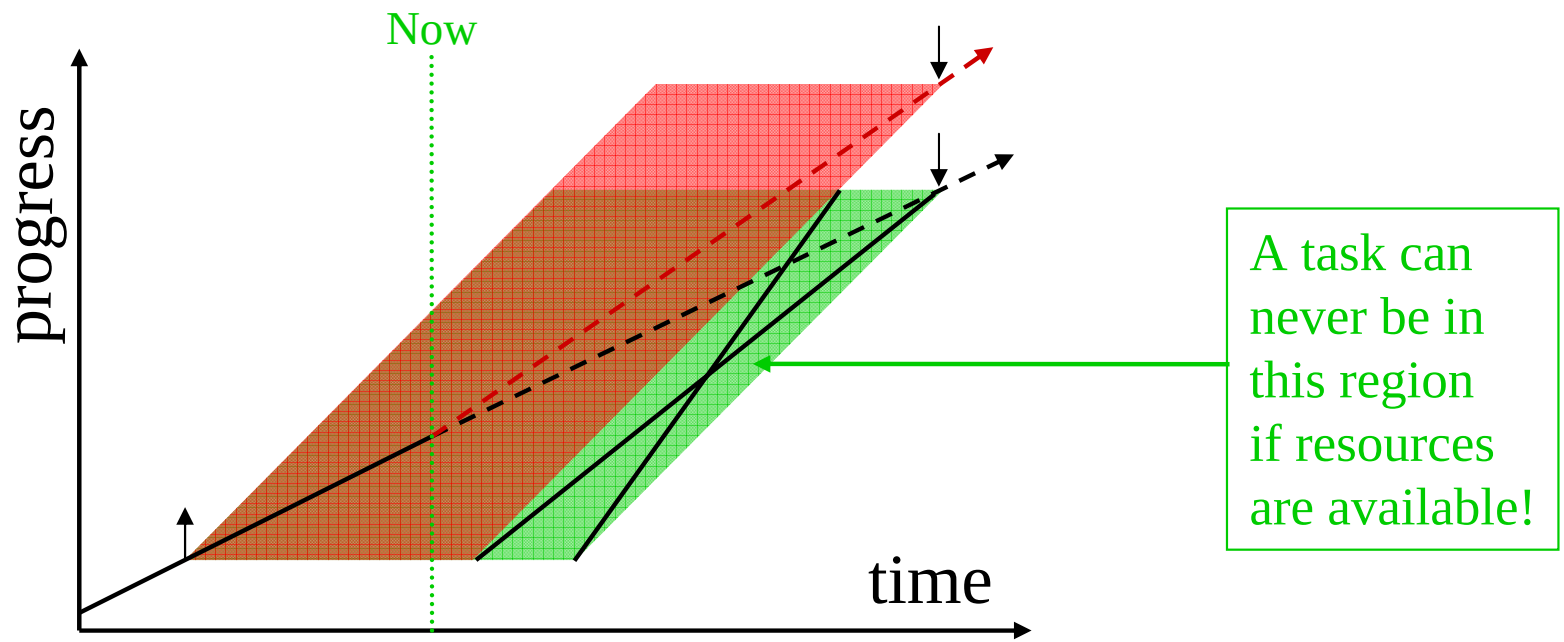
Increasing Rate (= increasing WCET)

- Theorem 2: *The resource usage of any task can be increased at any time, within the available resources*
  - Given a feasible EDF schedule, at any time task T_i may increase utilization by any amount up to $1-U$ without causing any task to miss deadlines in the resulting EDF schedule



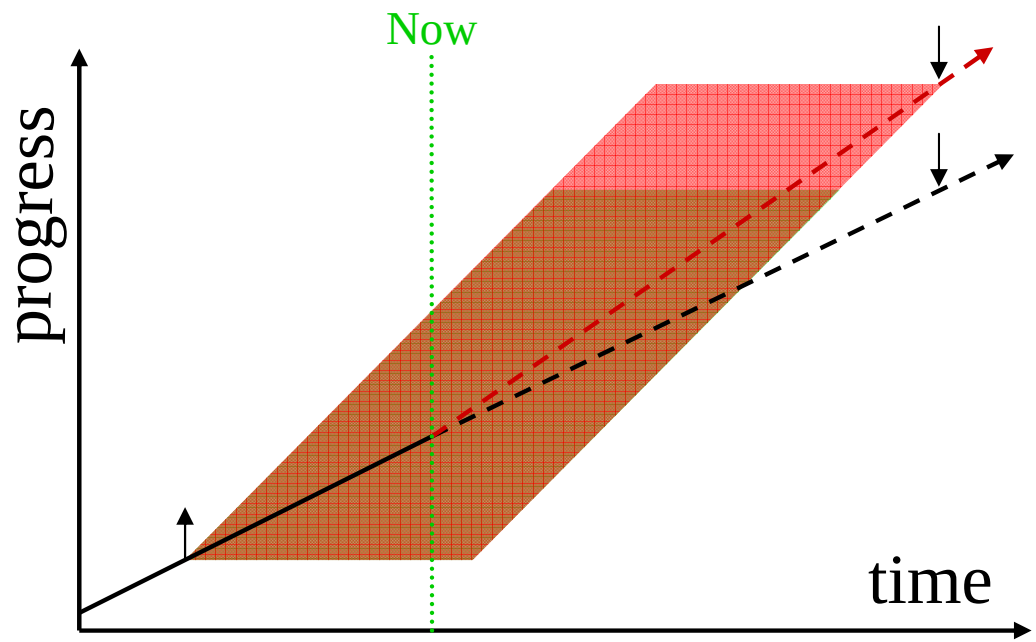
Increasing Rate (= increasing WCET)

- Theorem 2: *The resource usage of any task can be increased at any time, within the available resources*
  - Given a feasible EDF schedule, at any time task T_i may increase utilization by any amount up to $1-U$ without causing any task to miss deadlines in the resulting EDF schedule



Increasing Rate (= increasing WCET)

- Theorem 2: *The resource usage of any task can be increased at any time, within the available resources*
  - Given a feasible EDF schedule, at any time task *Ti* may increase utilization by any amount up to *1–U* without causing any task to miss deadlines in the resulting EDF schedule



RBED EDF Mode Change Theory

- Theorem 1: *EDF is optimal under this task model*
- Corollary: *A new task may enter at any time, within available resources*
- Theorem 2: *The rate of any task can be increased at any time, within available resources*
- Theorem 3: *The period of any task can be increased at any time*
- Theorem 4: *The rate of any task can be lowered at any time, down to what it has already used in the current period*
- Theorem 5: *The period of any task can be reduced at any time, down to the time corresponding to the current period's resource usage*
- Corollary: *The period of any task can be increased at any time (without changing WCET)*
- Corollary: *The period of a job which is ahead of its target allocation can be reduced at any time, down to the time corresponding to its current resource usage (without changing WCET) as long as the resources are available for the rate change*

RBED Theory Summary

- Rate and period can be changed without causing missed deadlines
  - At deadlines, rate and period changes are unconstrained (except by available resources)
  - In between, decreases are constrained by resource usage in the current period
  - The changes may be combined

- Isolation between processes is guaranteed

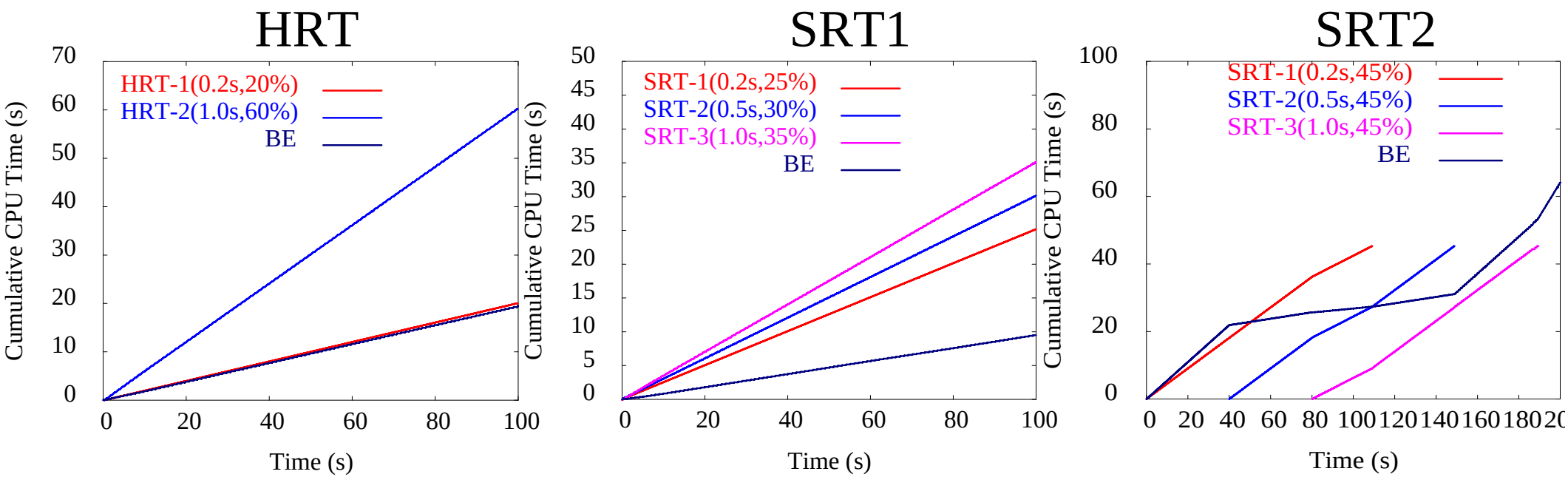
RBED Scheduling Policy

	Rate (Utilization)	Deadlines
HRT	Fixed: WCET	Fixed: Period
RB	Fixed: Rate	Fixed or Variable: Processing characteristics
SRT	Fixed or Variable: ACET, WCET – may vary based on resource availability, application adaptation	Fixed or Variable: Period – may vary based on resource availability, application adaptation
BE	Highly Variable: Based on resource availability, BE processing characteristics	Fixed or Variable: Based on BE processing characteristics

RBED Implementation

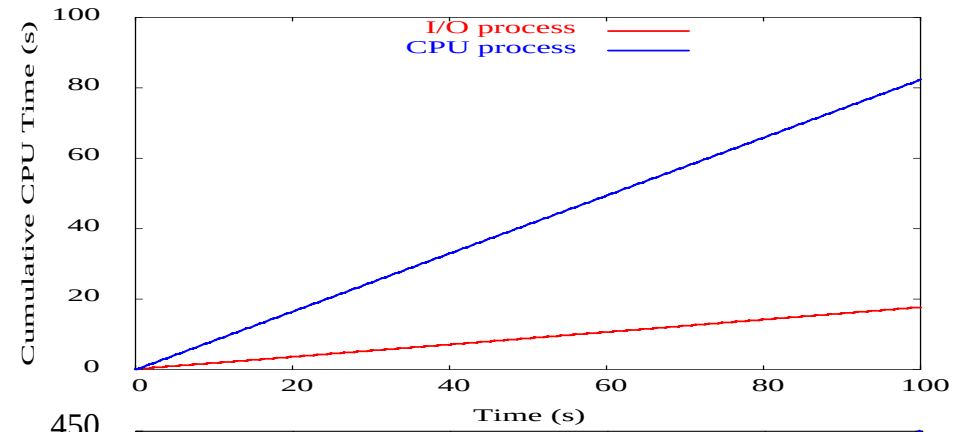
- Proof-of-concept implementation in Linux 2.4.20 kernel
  - Supports HRT, SRT, and BE
  - ~550 lines (kernel≈17000 lines)
    - Modifications to scheduler + new systems calls (for RT scheduling) + APIC setup and modification
- Configuration
  - 1 GHz CPU, 512 MB RAM, 40GB Disk
  - HRT job admission w/5% reserved for BE
- Workload
  - srt-gen: tool to generate flexible MPEG-like periodic workloads
  - Generic BE workload + some benchmarks
- Data Collection
  - Log data (timing information) in kernel every context switch
  - Dump data from the kernel to user space through rbed_tracedump() every second

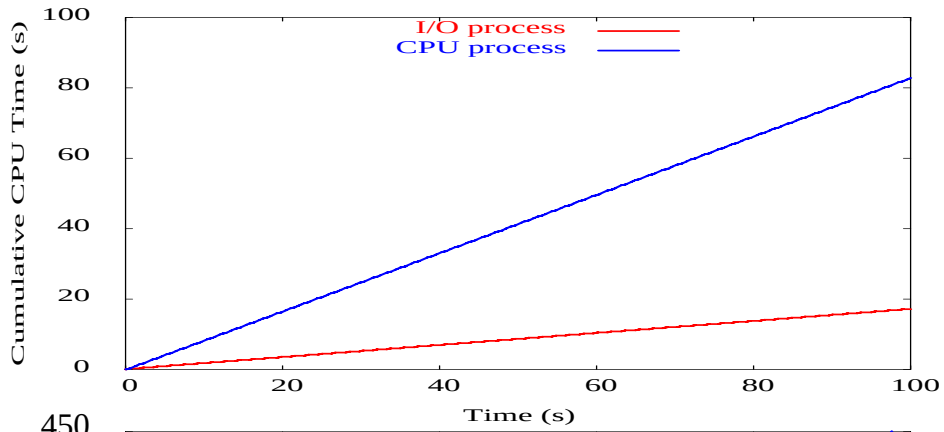
RBED HRT and SRT Performance

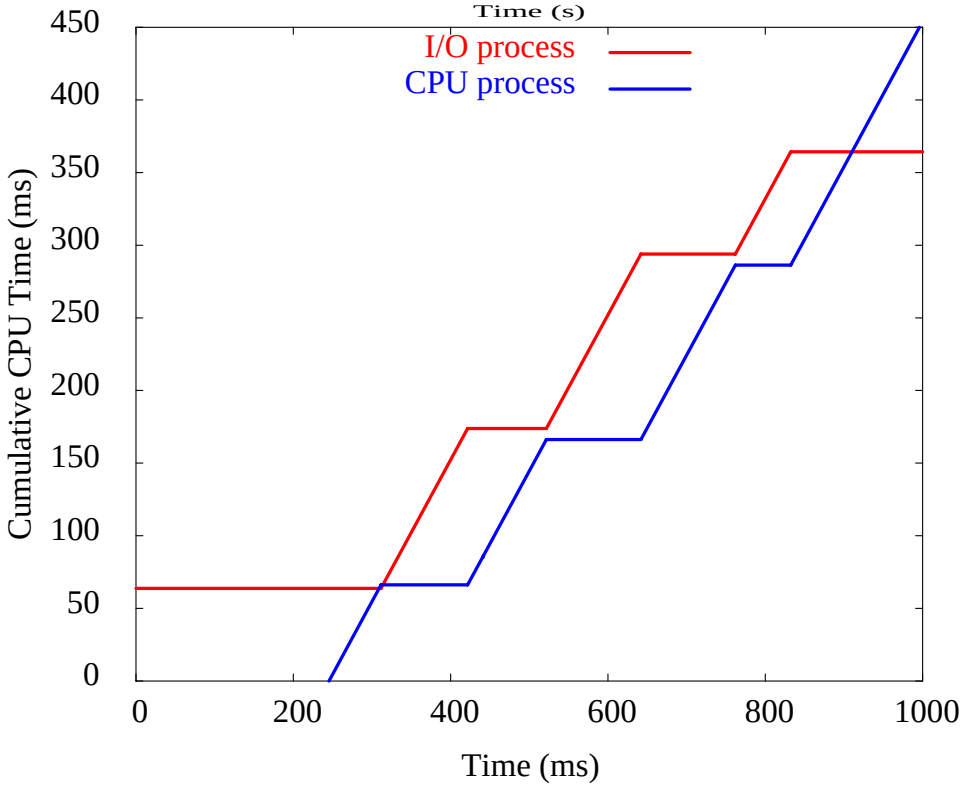


- HRT performance guaranteed by EDF scheduling
  - RBED guarantees isolation, admission control guarantees resource availability
- RB processing natively supported
  - Bounds enforced by release times and deadlines
- SRT performance based on resource availability
  - Rates and periods may vary whenever any HRT or SRT application enters or exits

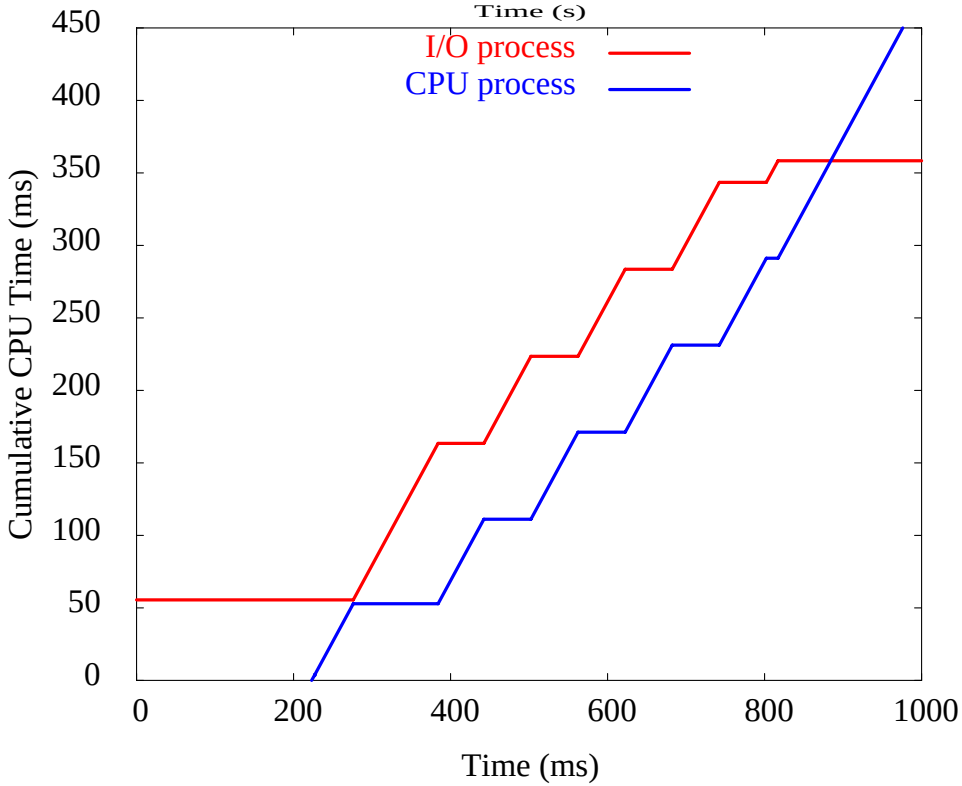
Best-Effort Performance: RBED vs Linux







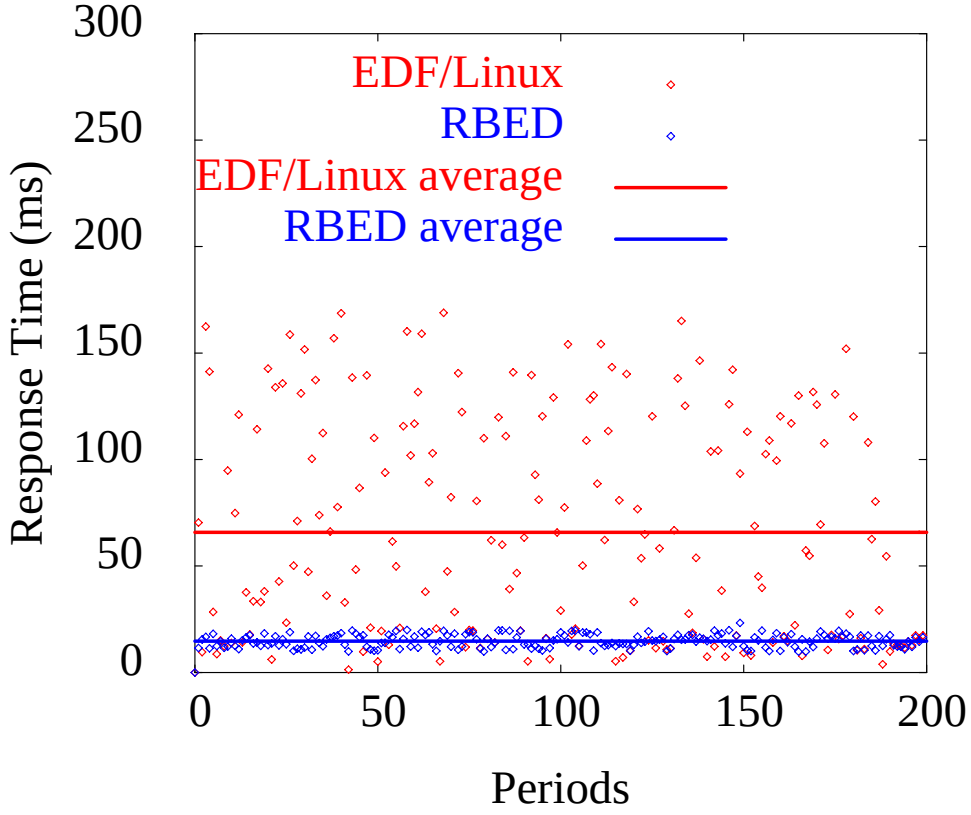
Linux



RBED

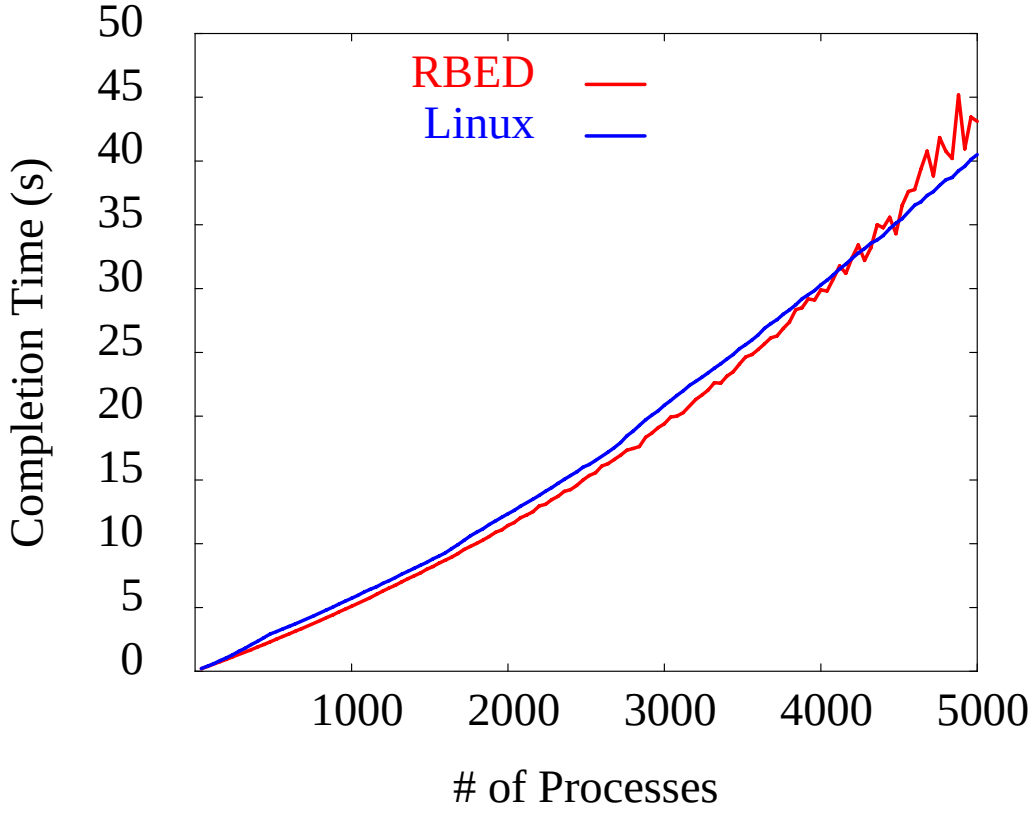
RBED Responsiveness and Overhead

Responsiveness vs. Hierarchical



BE task running with SRT
(Period: 190, WCET: 150)

Overhead vs. Linux

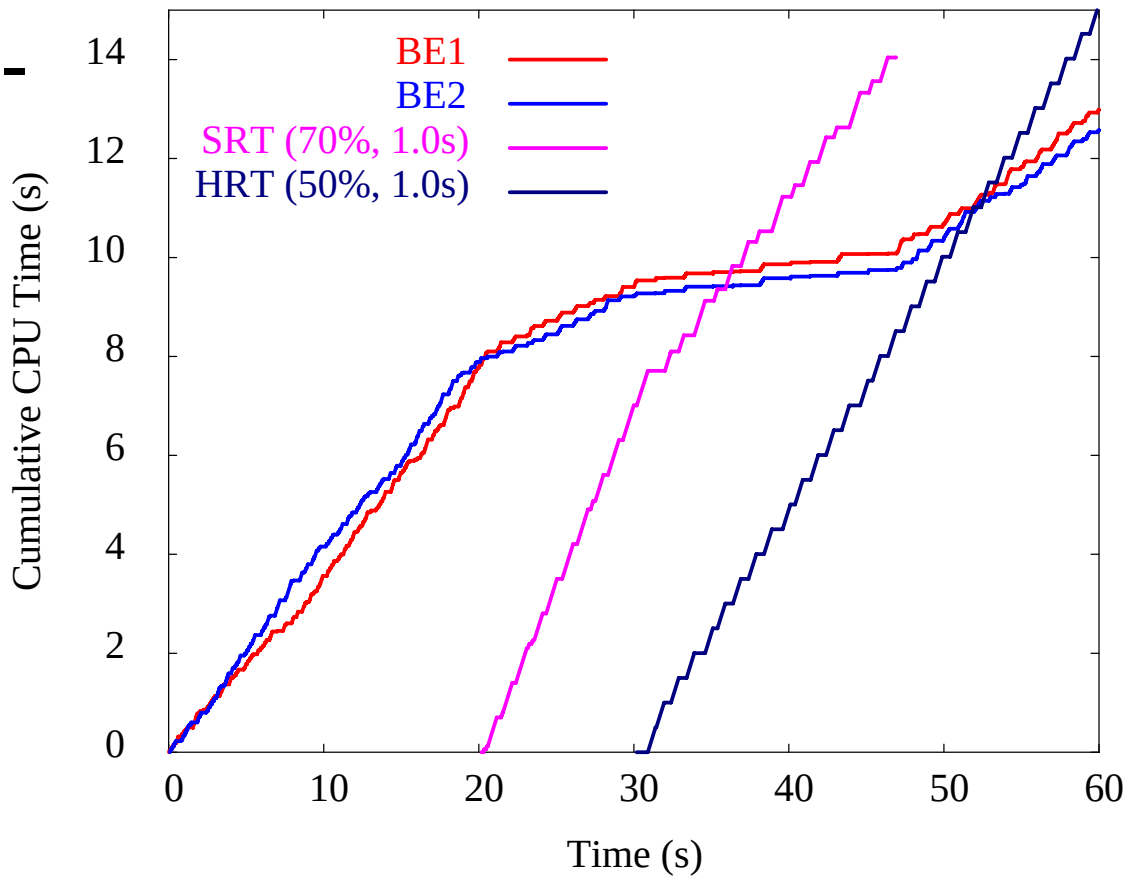


HackBench benchmark (OSDL)

- Divides total processes into two equal groups of readers and writers
- Each writer sends 100 500B messages to a subgroup of 20 Readers

RBED Integrated Real-Time Scheduling

- Guaranteed hard real-time
- Flexible soft real-time
- Robust rate-based
- Dynamic, responsive, starvation resistant best-effort

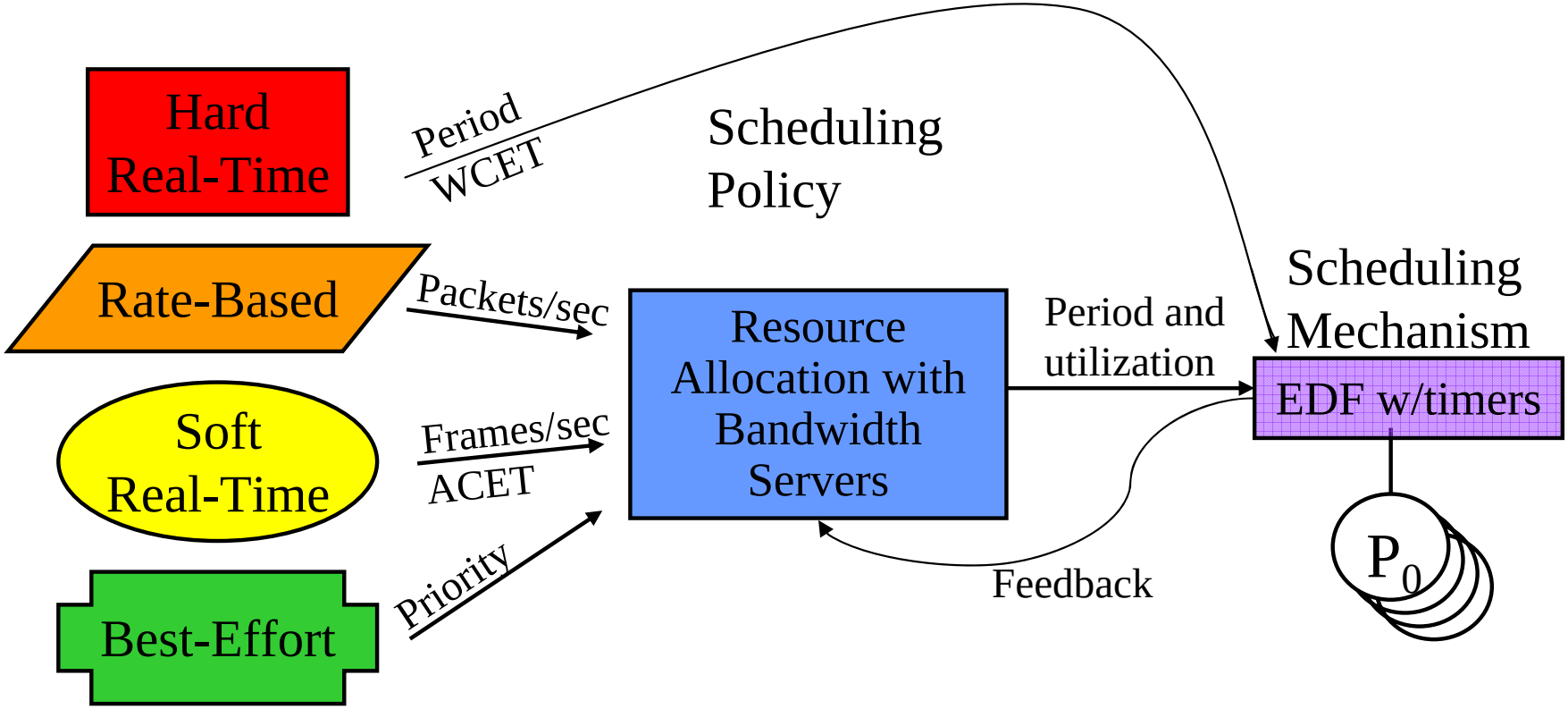


Extension 1: Rich SRT Support

Resource Allocation Bounds \ Dispatching Bounds	D_s in (0, 100%]	D_s in [D_{min}, 100%]	D_s = 100%	
tight	Missed-Deadline SRT	Firm Real-Time	Hard Real-Time	$U_s = U_t$
	CPU Bound	Rate-Based	Resource Adaptive SRT w/m	U_s in [U_{min}, U_t]
loose	Best Effort	I/O Bound	Resource Adaptive SRT wo/m	U_s in (0, U_t]

loose **Dispatching Bounds** tight

Extension 2: Automated SRT Support

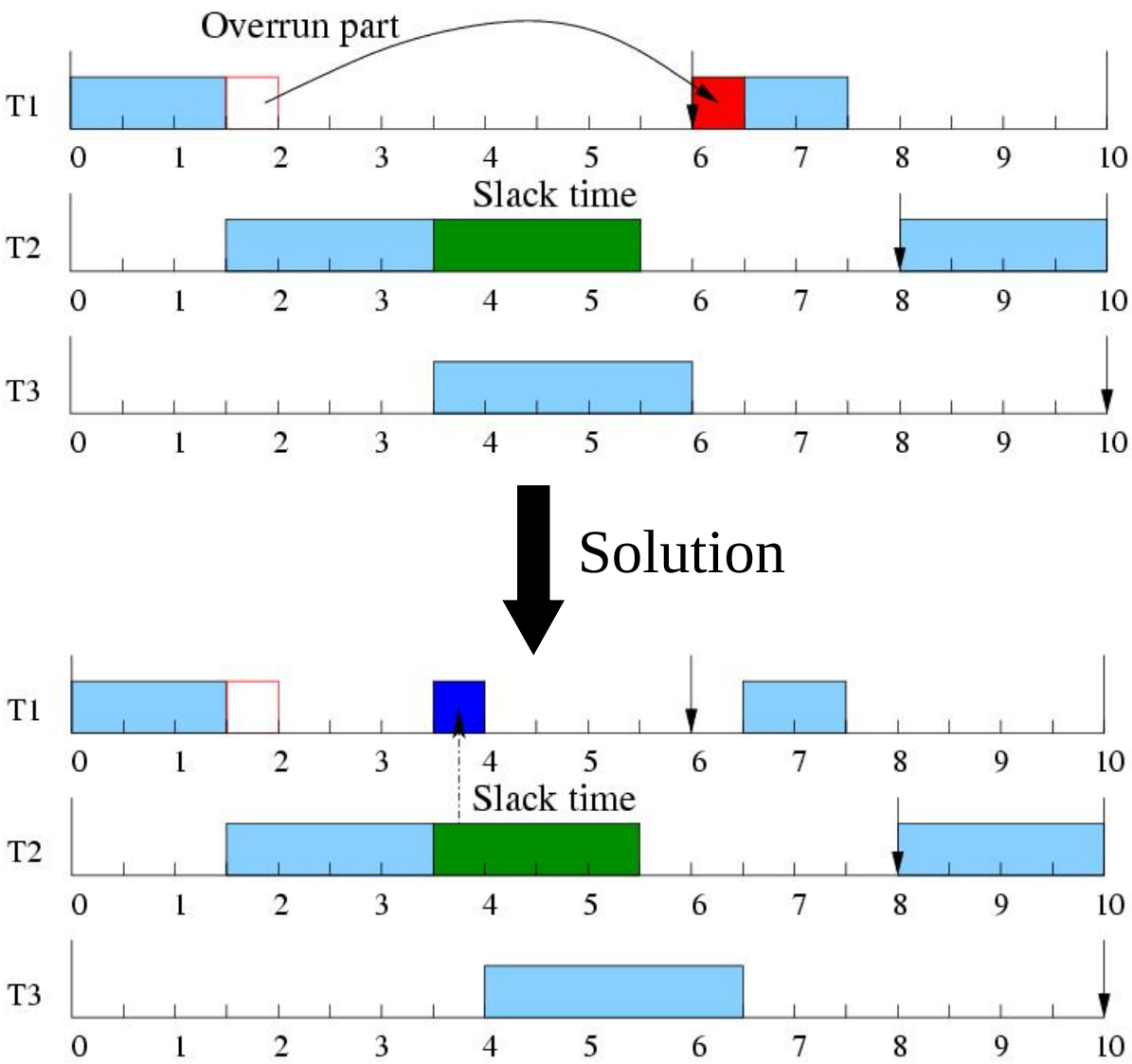


- A third type of BE process: CPU-bound, interactive, and **periodic**
- Aperiodic server for **each** BE/SRT/RB process
- **Dynamically determined rate and period** for each bandwidth server
- Improved slack reclamation

Better Slack Management: BACKSLASH

- Existing algorithms tend to ignore the needs of "background" tasks
  - Slack provided when everything else is idle
  - Aim for "fair" allocation and 100% utilization
- Slack reclamation is critical in an integrated real-time system
  - Utilization is important for best-effort systems
  - Soft real-time and best effort performance depends on the effective use of slack
- BACKSLASH improves performance via slack scheduling
  - Focuses on when slack is allocated, and to which process

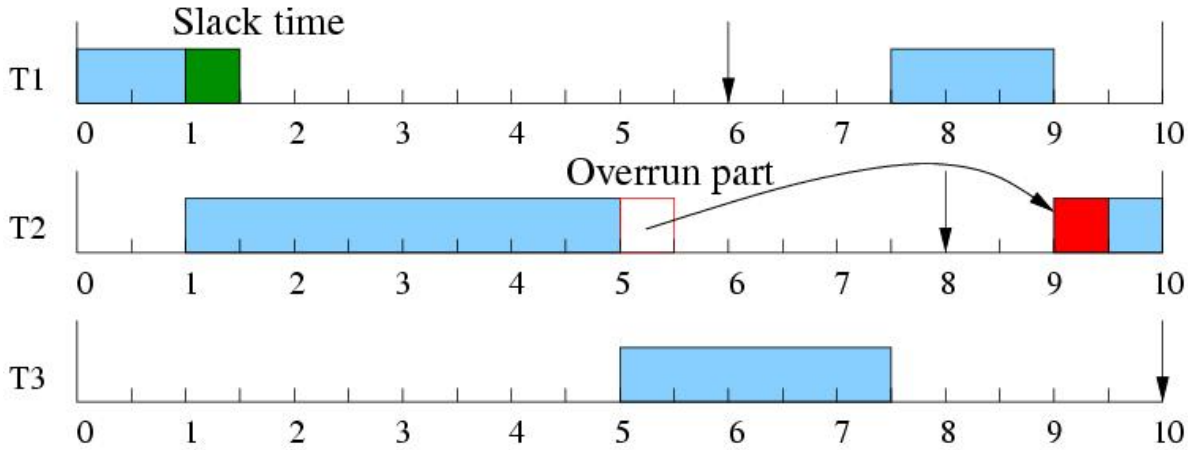
When To Allocate Slack?



Task	Reservation	Period
T1	1.5	6.0
T2	4.0	8.0
T3	2.5	10

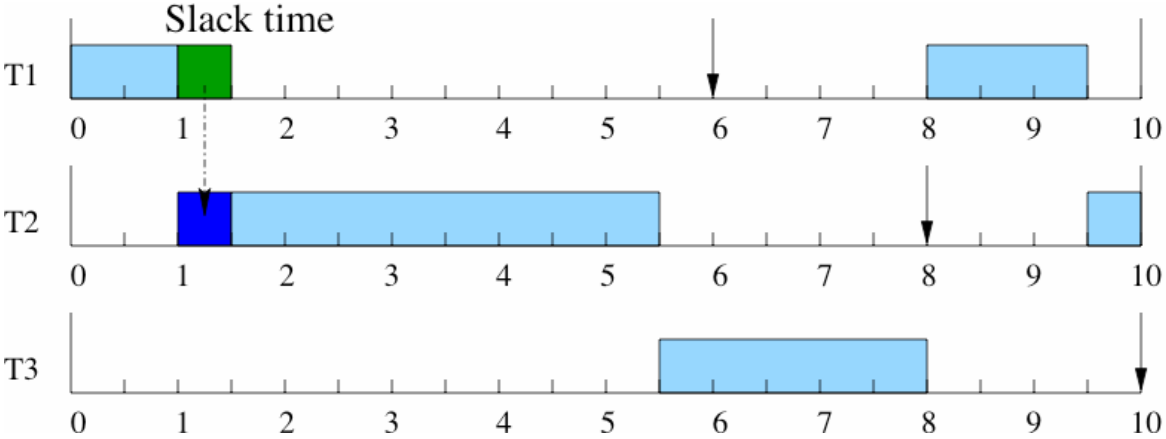
Answer: Allocate slack as early as possible

Who To Allocate Slack To?



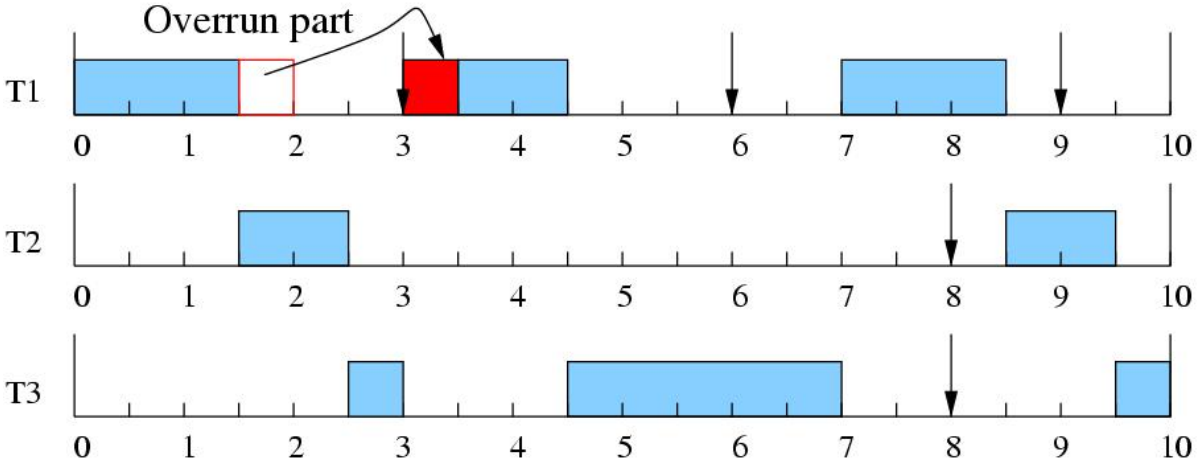
Task	Reservation	Period
T1	1.5	6.0
T2	4.0	8.0
T3	2.5	10

Solution



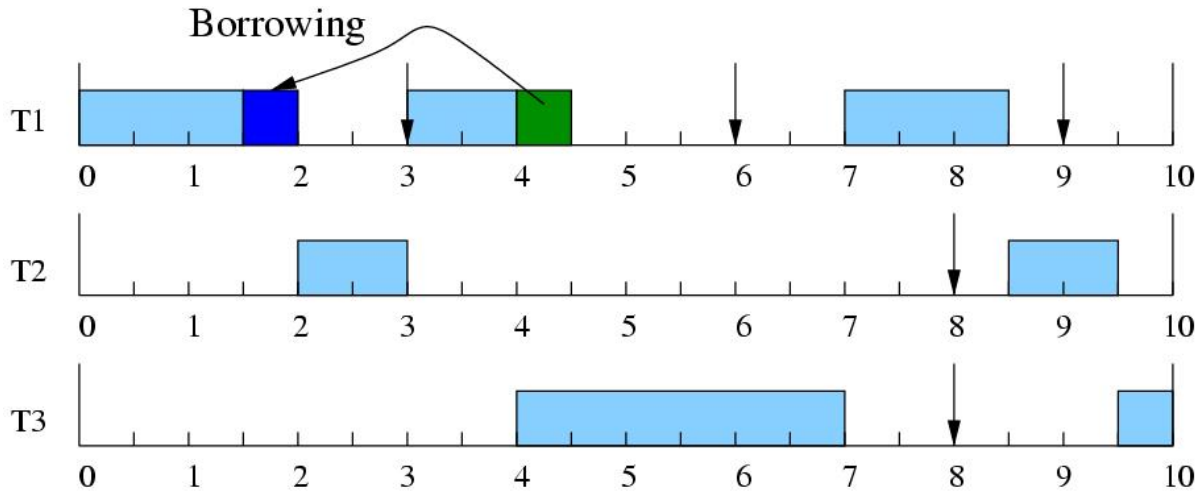
Answer: Allocate slack to the task with the earliest deadline

How To Use Future Slack?



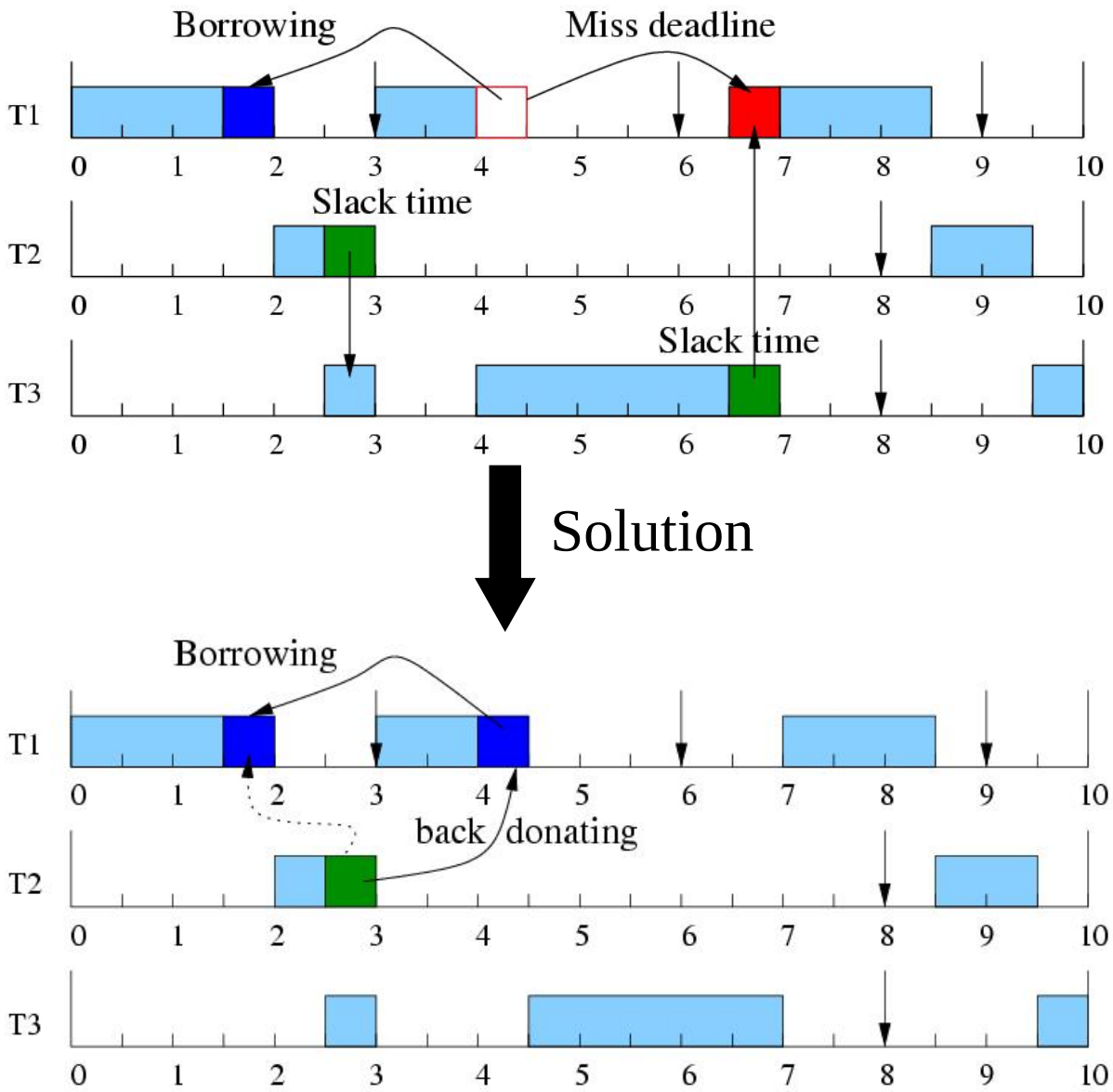
Task	Reservation	Period
T1	1.5	3.0
T2	1.0	8.0
T3	3	8

Solution



Answer: Borrow resources (potential slack) from the next job to meet the current deadline

How to Allocate Slack to Past Overruns?



Task	Reservation	Period
T1	1.5	3.0
T2	1.0	8.0
T3	3	8

Answer: Back-donate slack to tasks that borrowed from the future

Principles

1. Allocate slack as early as possible
   - With the priority of the donating task
2. Allocate slack to the task with highest priority (earliest original deadline)
   - *Task* deadline, not *server* deadline
3. Allow tasks to borrow against their own future resource reservations to complete their current job
   - With the priority of the donating job
4. Retroactively allocate slack to tasks that have borrowed from their current budget to complete a previous job

SRAND

+

SLAD

+

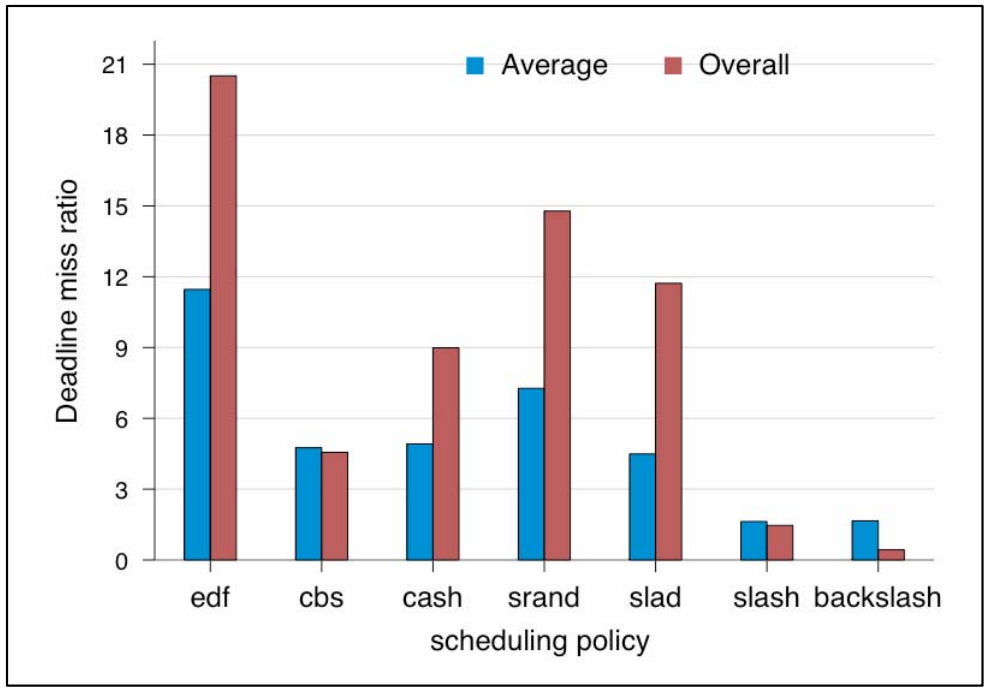
SLASH

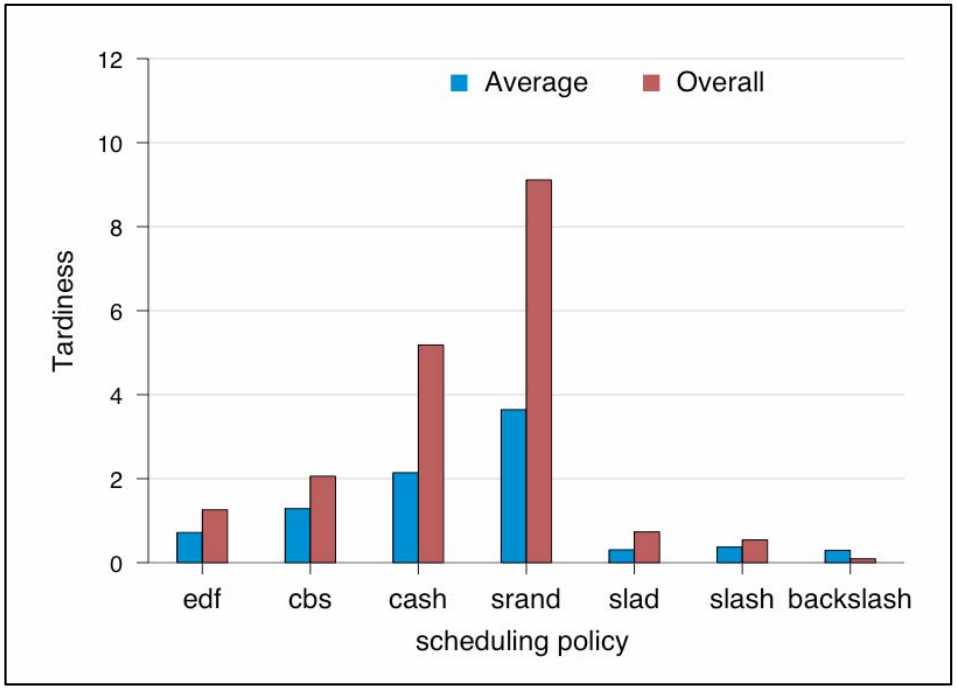
+

BACK SLASH

SRT Deadline Miss Ratio &

- 12 tasks sets with 8 tasks each
- Random # of hard and soft tasks in each set
- Random # of periodic and aperiodic tasks in each set
- Random period and exec. time ranging from 1 ms to 1 s

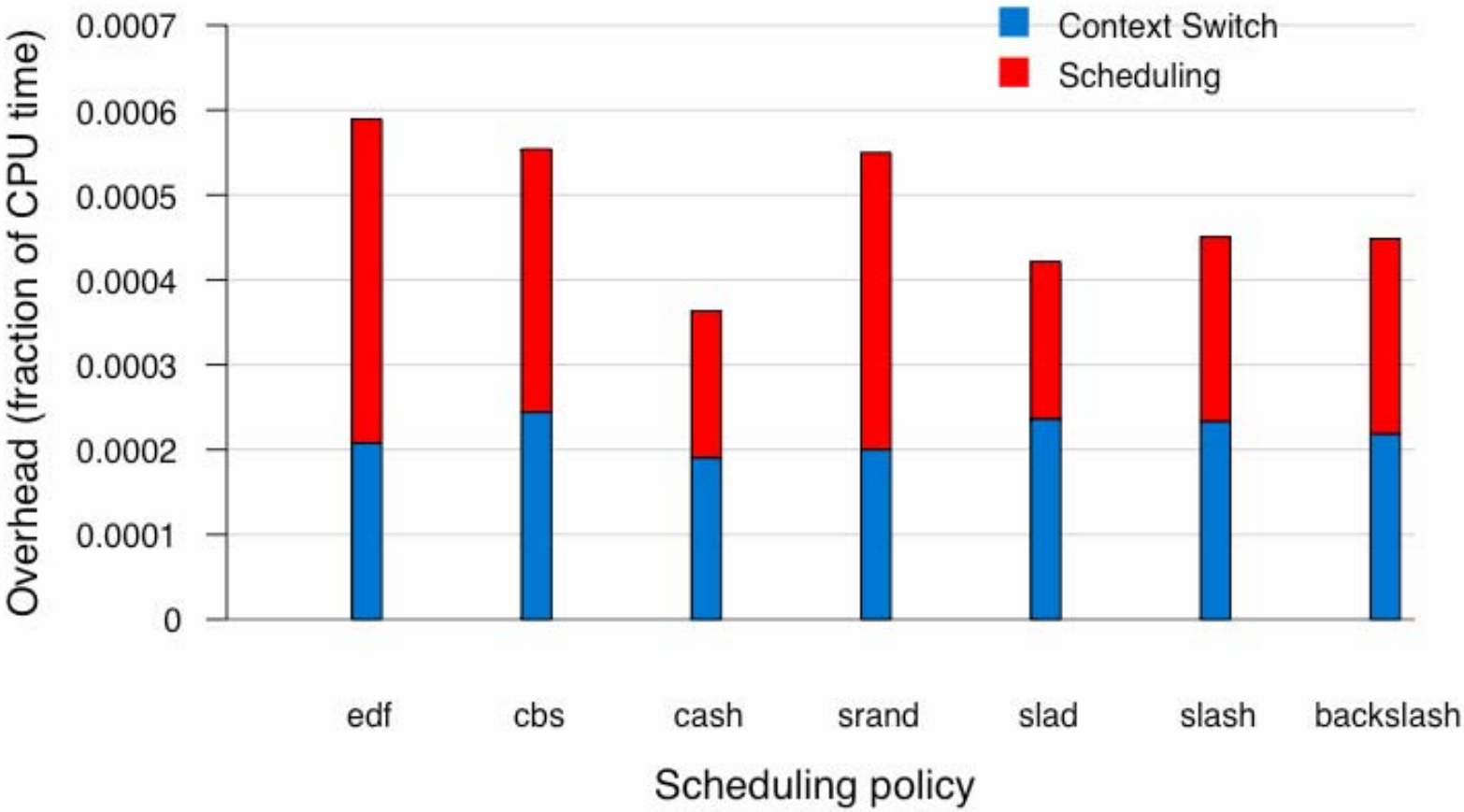




1. BACKSLASH < SLASH < SLAD < SRAND

2. BACKSLASH outperforms all the others

Overhead



Overhead is negligible

BACKSLASH Conclusions

- In an integrated system supporting HRT, SRT and BE, the performance of SRT (and BE) depends on the effective reclamation and distribution of slack
- Four principles for effective slack reclamation and distribution:
  1. Distribute slack as early as possible
  2. Give slack to the ready task with the highest priority
  3. Allow tasks to borrow against future reservations
  4. Retroactively give slack to tasks that needed it
  5. SMASH: Conserve slack across idle times!
- Our results show that these principles are effective: BACKSLASH significantly outperforms the other algorithms and improves SRT (and/or BE) performance