Global Scheduling Not Required:
Simple, Near-Optimal Multiprocessor Real-Time Scheduling
with Semi-Partitioned Reservations

November 30, 2016
RTSS 2016

Björn B. Brandenburg and Mahircan Gül

http://www.litmus-rt.org
Global Scheduling Not Required:
Simple, Near-Optimal Multiprocessor Real-Time Scheduling
with Semi-Partitioned Reservations

November 30, 2016
RTSS 2016

Björn B. Brandenburg and Mahircan Gül

for static sets of independent implicit-deadline sporadic tasks

http://www.litmus-rt.org
Global Scheduling Not Required:
Simple, Near-Optimal Multiprocessor Real-Time Scheduling with Semi-Partitioned Reservations

November 30, 2016
RTSS 2016

Björn B. Brandenburg and Mahircan Gül

for static sets of independent implicit-deadline sporadic tasks

empirically
Main Observation and Conclusions
Main Observation and Conclusions

Empirically, near-optimal hard real-time schedulability — usually ≥99% schedulable utilization — can be achieved with simple, well-known and well-understood, low-overhead techniques (+ a few tweaks).
Main Observation and Conclusions

Empirically, near-optimal hard real-time schedulability — usually \( \geq 99\% \) schedulable utilization — can be achieved with simple, well-known and well-understood, low-overhead techniques (+ a few tweaks).

\[ \rightarrow \text{Global, optimal scheduling not required} \]
(for the considered type of workloads!)

Pragmatically speaking, little reason to favor complex algorithms that are (more) difficult to understand, to implement, and to extend if a simple approach will do.
Main Observation and Conclusions

Empirically, near-optimal hard real-time schedulability — usually $\geq 99\%$ schedulable utilization — can be achieved with simple, well-known and well-understood, low-overhead techniques (+ a few tweaks).

➔ Global, optimal scheduling not required (for the considered type of workloads!)

Pragmatically speaking, little reason to favor complex algorithms that are (more) difficult to understand, to implement, and to extend if a simple approach will do.

➔ Future work should focus on more demanding workloads (on preemptive multiprocessor real-time scheduling)

Static, independent, implicit-deadline tasks are by now very well supported.
Motivation
Multiprocessor Real-Time Scheduling
Multiprocessor Real-Time Scheduling

Partitioning

1. **Assign** tasks to cores (offline).

2. Schedule each core **independently** (like a uniprocessor).
Multiprocessor Real-Time Scheduling

**Partitioning**

1. **Assign** tasks to cores (offline).
2. Schedule each core **independently** (like a uniprocessor).

**Partitioned Scheduling**
- simple to implement
- simple to understand
- simple to extend
- KISS-compatible
**Multiprocessor Real-Time Scheduling**

**Partitioning**
1. **Assign** tasks to cores (offline).
2. Schedule each core *independently* (like a uniprocessor).

**Partitioned Scheduling**
- simple to implement
- simple to understand
- simple to extend
- KISS-compatible

**But: non-optimal**
- need to place tasks (bin packing)
- mapping may be difficult to find
- mapping may not exist
- worst-case utilization bound ~50%
**Multiprocessor Real-Time Scheduling**

### Partitioning
1. **Assign** tasks to cores (offline).
2. Schedule each core ***independently*** (like a uniprocessor).

### Partitioned Scheduling
- simple to implement
- simple to understand
- simple to extend
- KISS-compatible

**But: non-optimal**
- need to place tasks (bin packing)
- mapping may be difficult to find
- mapping may not exist
- worst-case utilization bound ~50%

### Global/Optimal Scheduling
1. Keep all cores busy with sequential tasks.
2. Globally **coordinate** to reschedule and migrate tasks as needed.
Global Scheduling Not Required: Simple, Near-Optimal Multiprocessor Real-Time Scheduling with Semi-Partitioned Reservations

Multiprocessor Real-Time Scheduling

**Partitioning**

1. **Assign** tasks to cores (offline).
2. Schedule each core **independently** (like a uniprocessor).

**Partitioned Scheduling**

- simple to implement
- simple to understand
- simple to extend
- KISS-compatible

**But: non-optimal**

- need to place tasks (bin packing)
- mapping may be difficult to find
- mapping may not exist
- worst-case utilization bound ~50%

**Global/Optimal Scheduling**

1. Keep all cores busy with sequential tasks.
2. Globally **coordinate** to reschedule and migrate tasks as needed.

**Global Scheduling**

- optimality possible: 100% utilization
  - under restrictive assumptions
- many elegant algorithms: PD², BF, LLRef, EKG, U-EDF, RUN, QPS, ...
Global Scheduling Not Required: Simple, Near-Optimal Multiprocessor Real-Time Scheduling with Semi-Partitioned Reservations

Multiprocessor Real-Time Scheduling

Partitioning
1. Assign tasks to cores (offline).
2. Schedule each core independently (like a uniprocessor).

Partitioned Scheduling
- simple to implement
- simple to understand
- simple to extend
- KISS-compatible

But: non-optimal
- need to place tasks (bin packing)
- mapping may be difficult to find
- mapping may not exist
- worst-case utilization bound ~50%

Global/Optimal Scheduling
1. Keep all cores busy with sequential tasks.
2. Globally coordinate to reschedule and migrate tasks as needed.

Global Scheduling
- optimality possible: 100% utilization
  - under restrictive assumptions
- many elegant algorithms: PD², BF, LLRef, EKG, U-EDF, RUN, QPS, ...
The Question

(How) can we make global scheduling work in practice?

Much work in the last 10 years, both theory and systems…
The Question

(How) can we make global scheduling work in practice?

Do we actually need global, optimal scheduling!?
The Real Question

The “claim to fame” of global, optimal multiprocessor scheduling is 100% schedulable utilization…
The Real Question

The “claim to fame” of global, optimal multiprocessor scheduling is 100% schedulable utilization…

How to get close to 100% without giving up on simplicity?
The Real Question

The “claim to fame” of global, optimal multiprocessor scheduling is 100% schedulable utilization…

How to get close to 100% without giving up on simplicity?

…and how close can we get?
The Real Question

The “claim to fame” of global, optimal multiprocessor scheduling is 100% schedulable utilization…

How to get close to 100% without giving up on simplicity?

…and how close can we get?

Assumptions in Optimality Proofs:

- static set of tasks w/ static parameters
- independent tasks
- periodic or sporadic tasks
- implicit deadlines
- no jitter, no overheads, etc.
Essential Background
Hybrid: Semi-Partitioned Scheduling
(Anderson et al., 2005)
Hybrid: Semi-Partitioned Scheduling
(Anderson et al., 2005)

- Statically assign most tasks
Hybrid: Semi-Partitioned Scheduling
(Anderson et al., 2005)

- Statically assign **most** tasks
- Tasks that don’t fit are **split** into subtasks with precedence constraints
Hybrid: Semi-Partitioned Scheduling
(Anderson et al., 2005)

› Statically assign **most** tasks
› Tasks **that don’t fit** are **split** into **subtasks** with precedence constraints
› Assign subtasks to cores → some original tasks **migrate**
Hybrid: Semi-Partitioned Scheduling  
(Anderson et al., 2005)

- Statically assign **most** tasks
- Tasks **that don’t fit** are **split** into **subtasks** with **precedence constraints**
- Assign subtasks to cores  
  ➞ some original tasks **migrate**
- this is a **process migration**  
  ➞ no code changes in the task
Task $T_5$ split into two logical subtasks (= two budgets)

At runtime, $T_5$ migrates between cores 1 and 2.

- Statically assign **most** tasks
- Tasks that don’t fit are **split into subtasks** with precedence constraints
- Assign subtasks to cores → some original tasks **migrate**
- this is a **process migration** → no code changes in the task
Task $T_5$ split into two logical subtasks (= two budgets)

At runtime, $T_5$ migrates between cores 1 and 2.

- Statically assign **most** tasks
- Tasks that don’t fit are **split** into subtasks with precedence constraints
- Assign subtasks to cores → some original tasks **migrate**
- this is a **process migration** → no code changes in the task

Many **heuristics** for how to split, when to migrate, and where to assign subtasks...
Hybrid: Semi-Partitioned Scheduling

Simple Example
- Three identical tasks
  - period $P = 15$
  - WCET $C = 10$

One approach: split $T_3$
- into two subtasks $T'_3$, $T''_3$
  - $C' = C'' = 5$
  - $P' = P'' = 15$
  - $D' = 8$, $D'' = 7$
**Semi-Partitioning**

Still **core-local decisions**, one **cross-core activation**.

---

**Simple Example**

- Three identical tasks
  - period $P = 15$
  - WCET $C = 10$

**One approach:** split $T_3$

- into two subtasks $T'_3$, $T''_3$
  - $C' = C'' = 5$
  - $P' = P'' = 15$
  - $D' = 8$, $D'' = 7$
The C=D Splitting Strategy
(Burns et al., 2012)
The C=D Splitting Strategy
(Burns et al., 2012)

Assumption

- Earliest-Deadline First (**EDF**) policy is in use on each core
The C=D Splitting Strategy
(Burns et al., 2012)

Assumption
- Earliest-Deadline First (EDF) policy is in use on each core

Suppose $T_5$ does not fit (in its entirety) onto Core 1
- How to allocate some part of $T_5$ on Core 1?
The C=D Splitting Strategy
(Burns et al., 2012)

Assumption
- Earliest-Deadline First (EDF) policy is in use on each core

Suppose $T_5$ does not fit (in its entirety) onto Core 1
- How to allocate some part of $T_5$ on Core 1?

C=D Approach
- Given parameters $(C, D, P)$…
  …identify largest $C'$ and matching $C''$ such that
  - $C' + C'' = C$ // split execution cost
  - $D' = C'$ // zero-laxity subtask
  - $D'' = D - D'$ // remaining laxity subtask
  - $P' = P'' = P$ // period remains unchanged
  - and first subtask is schedulable on Core 1
Suppose $T_5$ does not fit (in its entirety) onto Core 1

⇒ How to allocate some part of $T_5$ on Core 1?

**C=D Approach**

⇒ Given parameters ($C$, $D$, $P$)...

...identify largest $C'$ and matching $C''$ such that

- $C' + C'' = C$  // split execution cost
- $D' = C'$  // zero-laxity subtask
- $D'' = D - D'$  // remaining laxity subtask
- $P' = P'' = P$  // period remains unchanged
- and first subtask is schedulable on Core 1

**Assumption**

⇒ Earliest-Deadline First (EDF) policy is in use on each core

Suppose $T_5$ does not fit (in its entirety) onto Core 1

⇒ How to allocate some part of $T_5$ on Core 1?

**C=D Approach**

⇒ Given parameters ($C$, $D$, $P$)...

...identify largest $C'$ and matching $C''$ such that

- $C' + C'' = C$  // split execution cost
- $D' = C'$  // zero-laxity subtask
- $D'' = D - D'$  // remaining laxity subtask
- $P' = P'' = P$  // period remains unchanged
- and first subtask is schedulable on Core 1

**Assumption**

⇒ Earliest-Deadline First (EDF) policy is in use on each core
Reservation-Based Scheduling
(Mercer et al., 1993)
Reservation-Based Scheduling
(Mercer et al., 1993)

Two-Level Scheduling
- threads / tasks encapsulated in reservations
- to schedule:
  ‣ first pick reservation
  ‣ then pick thread

reservation
reservation
reservation

T
T
T
Reservation-Based Scheduling
(Mercer et al., 1993)

Two-Level Scheduling
- threads / tasks encapsulated in reservations
- to schedule:
  ‣ first pick reservation
  ‣ then pick thread

Reservations (or Servers)
- many algorithms available in the literature
- Most simple one:
  ‣ sporadic polling server
    = sporadic task
    + budget enforcement
Reservation-Based Scheduling
(Mercer et al., 1993)

Two-Level Scheduling
- threads / tasks encapsulated in reservations
- to schedule:
  ‣ first pick reservation
  ‣ then pick thread

Reservations (or Servers)
- many algorithms available in the literature
- Most simple one:
  ‣ sporadic polling server
    = sporadic task
    + budget enforcement

Hard vs. Soft Reservations
(Rajkumar et al., 1998)
When running out of budget:

hard = cut off from service
soft = may consume idle time with background priority
A Simple Semi-Partitioned Reservations Approach
Approach in a Nutshell

$T_1 \quad T_2 \quad T_3 \quad … \quad T_n$
Approach in a Nutshell

1) Partitioned Reservation Scheduler
- EDF-based, completely local
- simple to implement efficiently
Approach in a Nutshell

1) Partitioned Reservation Scheduler
   - EDF-based, completely local
   - simple to implement efficiently

2) One Task ↔ One Reservation
   - initially, reservation parameters = task parameters
   - soft sporadic polling reservations
     (or CBS or…)

\[ T_1 \rightarrow S_1, T_2 \rightarrow S_2, T_3 \rightarrow S_3, \ldots, T_n \rightarrow S_n \]
Approach in a Nutshell

1) Partitioned Reservation Scheduler
   - EDF-based, completely local
   - simple to implement efficiently

2) One Task ↔ One Reservation
   - initially, reservation parameters = task parameters
   - soft sporadic polling reservations
     (or CBS or…)

3) Use C=D + Some Tweaks…
   - place all reservations, splitting some if necessary
   - potentially tweak reservation parameters
   - try to avoid migrations whenever possible
Approach in a Nutshell

1) Partitioned Reservation Scheduler
- EDF-based, completely local
- simple to implement efficiently

2) One Task ↔ One Reservation
- initially, reservation parameters = task parameters
- soft sporadic polling reservations
  (or CBS or…)

3) Use C=D + Some Tweaks…
- place all reservations, splitting some if necessary
- potentially tweak reservation parameters
- try to avoid migrations whenever possible

…and that’s it!
Tweak 1: Try Many Heuristics
Tweak 1: Try Many Heuristics

Most heuristics are cheap...
→ ...so don’t choose, run them “all.”
Tweak 1: Try Many Heuristics

Most heuristics are cheap…
→ …so don’t choose, run them “all.”

Observation: It pays to play with details
→ When to split, how much to split off, where to place subtasks…
→ Minor differences add up.
Tweak 1: Try Many Heuristics

Most heuristics are cheap...
- ...so don’t choose, run them “all.”

Observation: It pays to play with details
- When to split, how much to split off, where to place subtasks...
- Minor differences add up.

Observation: C=D works also well with worst-fit decreasing (WFD)
- Trivial...
- ...but prior evaluations of C=D have focused primarily on first-fit decreasing (FFD) and thus not exploited its full potential.
Tweak 2: Pre-Assign Failures (PAF) Meta-Heuristic
Tweak 2: Pre-Assign Failures (PAF) Meta-Heuristic

Idea: Use **Heuristic Failure** as a **Signal** in a **Feedback Loop**

- The tasks that couldn’t be placed must be difficult somehow…

  …so try to place them first!
Tweak 2: **Pre-Assign Failures (PAF) Meta-Heuristic**

Idea: Use **Heuristic Failure as a Signal in a Feedback Loop**

- The tasks that couldn't be placed must be difficult somehow…
  
  ...so try to place them first!

**Procedure PAF(h1, h2, taskset)**

Initialize:

- rest = taskset
- failures = ∅

While no solution is found:

1. **Assign** all tasks in **failures** with **h1**
   - **give up** if this fails

2. Assign all tasks in **rest** with **h2** while respecting pre-assignment by **h1**
   - **success** if complete solution is found
   - otherwise move any **unplaced tasks** to **failures**
Tweak 2: **Pre-Assign Failures (PAF)** Meta-Heuristic

**Idea:** Use Heuristic Failure as a Signal in a Feedback Loop

- *The tasks that couldn’t be placed must be difficult somehow…*
  ...so try to place them first!

**Procedure PAF**(h₁, h₂, taskset)

**Initialize:**
- rest = taskset
- failures = ∅

**While no solution is found:**

1. **Assign** all tasks in **failures** with **h₁**
   → *give up* if this fails

2. Assign all tasks in **rest** with **h₂** while respecting pre-assignment by **h₁**
   → *success* if complete solution is found
   → otherwise move any **unplaced tasks** to **failures**

**regular task-placement heuristics** (e.g., WFD, FFD + C=D)
Tweak 3: **Reduce Periods (RP) Meta-Heuristic**
Tweak 3: **Reduce Periods (RP) Meta-Heuristic**

Observation: the $C=D$ splitting heuristic is not “scale invariant”

- splitting off a subtask with $C'=D'=1$ from a $(C=2, P=10)$ task vs.
  splitting off a subtask with $C'=D'=100$ from a $(C=200, P=1000)$ task

- Both subtasks have 10% utilization and 100% density…
  …but $C'=D'=1$ is **much easier to accommodate**.
Tweak 3: **Reduce Periods (RP) Meta-Heuristic**

Observation: the $C=D$ splitting heuristic is not “scale invariant”

- splitting off a subtask with $C'=D'=1$ from a $(C=2, P=10)$ task vs.
  - splitting off a subtask with $C'=D'=100$ from a $(C=200, P=1000)$ task
- Both subtasks have 10% utilization and 100% density…
  - but $C'=D'=1$ is *much easier to accommodate*.

Idea: transform period prior to semi-partitioning

- apply **period transformation** to spread out the load of “difficult” tasks
- very effective at reducing the “chunk size” that $C=D$ must deal with
Example

If a task requires 2 ms every 10 ms, we can instead also schedule it for 1 ms every 5 ms:

Idea: transform period prior to semi-partitioning

→ apply period transformation to spread out the load of “difficult” tasks

→ very effective at reducing the “chunk size” that C=D must deal with
Tweak 3: **Reduce Periods (RP) Meta-Heuristic**

**Observation**: the C=D splitting heuristic is not “scale invariant”
- splitting off a subtask with \( C'=D'=1 \) from a \( (C=2, P=10) \) task
  vs.
  splitting off a subtask with \( C'=D'=100 \) from a \( (C=200, P=1000) \) task
- Both subtasks have 10% utilization and 100% density…
  …but \( C'=D'=1 \) is much easier to accommodate.

**Idea**: transform period prior to semi-partitioning
- apply **period transformation** to spread out the load of “difficult” tasks
- very effective at reducing the “chunk size” that C=D must deal with

**Practical Considerations**
- trivial to support: no code changes, just tweak reservation parameters
- tradeoff: increased preemption / migration frequency
Tweak 4: Flip the C=D Subtask Order
Tweak 4: Flip the C=D Subtask Order

zero laxity

C=D subtask

job arrival

job deadline

time

normal order
Tweak 4: Flip the C=D Subtask Order

- **C=D subtask**
  - zero laxity
- **C ≤ D subtask**
  - non-zero laxity, subject to interference

Diagram:
- Job arrival to time
- Job deadline

Normal order
Tweak 4: Flip the C=D Subtask Order

Idea: execute zero-laxity subtask(s) last

- Irrelevant from theory point of view: order is arbitrary.
- Quite useful from systems point of view…
Tweak 4: Flip the C=D Subtask Order

Idea: execute zero-laxity subtask(s) last
- Irrelevant from theory point of view: order is arbitrary.
- Quite useful from systems point of view…
Tweak 4: Flip the C=D Subtask Order

Idea: execute **zero-laxity** subtask(s) last
- Irrelevant from theory point of view: order is arbitrary.
- Quite useful from systems point of view…
Tweak 5: Use Slack to Avoid Migrations

job arrival  

job deadline  

time

\[\text{time} \quad \text{job arrival} \quad \text{job deadline}\]
Tweak 5: Use Slack to Avoid Migrations

- **C ≤ D subtask**
- **C=D subtask**
- **zero laxity**
- **non-zero laxity, subject to interference**

job arrival \( \rightarrow \) \( \rightarrow \) \( \rightarrow \) job deadline

\( C \leq D \) subtask

worst case

time
Tweak 5: Use Slack to Avoid Migrations

Idea: use a *simple* slack reclamation scheme

- finish job before it must migrate (➔ thanks to *flipped* subtask order)
- our implementation uses CASH (Caccamo et al., 2000)
Tweak 5: Use Slack to Avoid Migrations

Idea: use a simple slack reclamation scheme
- finish job before it must migrate (thanks to flipped subtask order)
- our implementation uses CASH (Caccamo et al., 2000)
Tweak 5: Use Slack to Avoid Migrations

Idea: use a simple slack reclamation scheme

- finish job before it must migrate (→ thanks to flipped subtask order)
- our implementation uses CASH (Caccamo et al., 2000)
Tweak 5: Use Slack to Avoid Migrations

- **C ≤ D subtask**
  - zero laxity
  - migration avoided
- **C = D subtask**
  - non-zero laxity, subject to interference

**Idea:** use a *simple* slack reclamation scheme

- finish job before it must migrate (thanks to *flipped* subtask order)
- our implementation uses CASH (Caccamo et al., 2000)

---

**¡Feliz Año Nuevo!**
Does it work in theory?

— schedulability experiments —
Schedulability Experiments — Setup
Schedulability Experiments — Setup

**Metric**

- schedulability = \[
\frac{\text{number of schedulable task sets}}{\text{total number of tested task sets}}
\]

- optimal ↔ schedulability = 1
Schedulability Experiments — Setup

**Metric**
- schedulability = \( \frac{\text{number of schedulable task sets}}{\text{total number of tested task sets}} \)

- optimal \( \leftrightarrow \) schedulability = 1

**Number of Processors** \( m \)
- considered range: 2, 4, 8, 16, 24, 32, 64 processors
Schedulability Experiments — Setup

**Metric**

- schedulability $= \frac{\text{number of schedulable task sets}}{\text{total number of tested task sets}}$

- optimal $\leftrightarrow$ schedulability $= 1$

**Number of Processors** $m$

- considered range: 2, 4, 8, 16, 24, 32, 64 processors

**Number of Tasks** $n$

- considered range: $m + 1$, ..., $3m$
Schedulability Experiments — Setup

Metric

- schedulability = \[
\frac{\text{number of schedulable task sets}}{\text{total number of tested task sets}}
\]

- optimal \leftrightarrow \text{schedulability} = 1

Number of Processors \(m\)

- considered range: 2, 4, 8, 16, 24, 32, 64 processors

Number of Tasks \(n\)

- considered range: \(m + 1, \ldots, 3m\)

Task Periods

- chosen from \{1, 2, 4, 5, 8, 10, 20, 25, 40, 50, 100, 125, 200, 250, 500, 1000\}
  uniformly at random (in milliseonds)
- range commonly found in automotive systems
Schedulability Experiments — Setup

Metric

- schedulability = \( \frac{\text{number of schedulable task sets}}{\text{total number of tested task sets}} \)
- optimal ↔ schedulability = 1

Number of Processors \( m \)
- considered range: 2, 4, 8, 16, 24, 32, 64 processors

Number of Tasks \( n \)
- considered range: \( m + 1, \ldots, 3m \)

Task Periods
- chosen from \{1, 2, 4, 5, 8, 10, 20, 25, 40, 50, 100, 125, 200, 250, 500, 1000\} uniformly at random (in milliseconds)
- range commonly found in automotive systems

Task Utilization
- Emberson et al. (2010) task-set generator (designed to be unbiased)
- “UNC style” task-set generator (used in prior LITMUS\textsuperscript{RT} studies)
Smaller $n =$ more difficult bin-packing instance

- fewer, larger items = harder problem

Higher utilization = more difficult bin-backing instance

- less spare capacity = harder problem
Expected Outcome

![Graph showing system utilization vs. ratio of schedulable task sets]

- **gap**: measure of sub-optimality
- **optimal**: 1 up to 100% utilization
- **heuristic (illustration)**
- **partitioning utilization bound**: 1 up to 50% utilization, no guarantee thereafter

**Smaller n = more difficult bin-packing instance**
- fewer, larger items = harder problem

**Higher utilization = more difficult bin-backing instance**
- less spare capacity = harder problem
Performance of **Partitioned Scheduling** (8 Cores)

![Graph showing the performance of partitioned scheduling](image)

- **n=9**, **n=12**, **n=16**, **n=24**, **n=32**, **n=10**, **n=14**, **n=20**, **n=28**

- The graph indicates the fraction of schedulable task sets against system utilization (in percent).
Performance of **Partitioned Scheduling** (8 Cores)

No problems up to 75% utilization.
Performance of **Partitioned** Scheduling (8 Cores)

For small values of $n$ (9 = $m+1$), the performance is better for smaller workloads. For large values of $n$ (32 = 3$m$), the performance is better for larger workloads.
Close to optimal (>95% schedulable utilization) for $n = 3m = 24$

→ scheduling with implicit deadlines is difficult

only for small $n$, high-utilization task sets

![Graph showing the fraction of schedulable task sets against system utilization for different values of $n$. The graph indicates that the scheduling is optimal up to 100% utilization for certain values of $n$.](image)
Global Scheduling Not Required: Simple, Near-Optimal Multiprocessor Real-Time Scheduling with Semi-Partitioned Reservations

Performance of Partitioned Scheduling (8 Cores)

Not a big gap for optimal algorithms to exploit: much complexity for little gain!

Let’s try semi-partitioning…

optimal
1 up to 100% utilization
Before: Partitioning Only

fraction of schedulable task sets

system utilization (in percent)

n=9  n=12  n=16  n=24  n=32
n=10  n=14  n=20  n=28
With Basic Semi-Partitioning

Let's zoom in…
With Basic Semi-Partitioning [Zoomed In]

![Graph showing the fraction of schedulable task sets against system utilization (in percent). The x-axis ranges from 90 to 100, and the y-axis ranges from 0 to 1. Different lines represent different task set sizes (n=9, n=12, n=16, n=24, n=32, n=10, n=14, n=20, n=28).]
With Basic Semi-Partitioning [Zoomed In]

No problems up to 90% utilization!
(X scale changed)
With Basic Semi-Partitioning [Zoomed In]

Even smaller gap
at 95% utilization, lowest curve
at ~75% schedulability

Still, can’t we get there somehow…?

optimal
1 up to 100% utilization
With Pre-Assign Failures Heuristic (PAF)

Let’s zoom in again…
With Pre-Assign Failures Heuristic (PAF) [2X Zoomed In]
With Pre-Assign Failures Heuristic (PAF) [2X Zoomed In]

Scale starts at 95% utilization!

No problems up to 98% utilization!
With Pre-Assign Failures Heuristic (PAF) [2X Zoomed In]

Even smaller gap at 99% utilization, lowest curve at 90+% schedulability. Still, can’t we get there somehow…?

optimal 1 up to 100% utilization
Semi-Partitioning with PAF + Period Transformation

- n=9
- n=12
- n=16
- n=24
- n=32
- n=10
- n=14
- n=20
- n=28

The graph shows the fraction of schedulable task sets against system utilization (in percent). The x-axis represents the system utilization ranging from 95% to 100%, and the y-axis represents the fraction of schedulable task sets ranging from 0 to 1.
Semi-Partitioning with PAF + Period Transformation

Near-optimal 99+% schedulability for any task count.
Summary for 8 Cores, 16 Tasks ($n=2m$)

For overview, let’s consider just one task count ($n=16$).
Summary for 8 Cores, 16 Tasks (n=2m)

For overview, let’s consider just one task count (n=16).

![Graph showing fraction of schedulable task sets vs. system utilization]

- **basic semi-partitioning**
- **partitioning only**

With both meta-heuristics:
- PAF meta-heuristic
- basic semi-partitioning
- partitioning only
Summary for 8 Cores, 16 Tasks ($n=2m$)

PAF surprisingly effective & period transformation closes the last gap — empirically, virtually optimal schedulability
What about other core counts? $(n=2m)$

$\rightarrow$ trends largely independent of $m$
What about other core counts? \((n=2m)\)

\(\rightarrow\) trends largely independent of \(m\)
What about other core counts? \((n=2m)\)

\[\rightarrow \text{trends largely independent of } m\]
What about other core counts? (n=2m) → trends largely independent of m
What about other core counts? \( (n=2m) \)

\[ \rightarrow \text{trends largely independent of } m \]
What about the task-set generator? \( (n=2m) \)

→ very similar for completely different task-set generator
What about the task-set generator? (n=2m) 
→ very similar for completely different task-set generator

m=4, UNC gen., exponential-heavy
What about the task-set generator? (n=2m) → very similar for completely different task-set generator

m=4, UNC gen., exponential-heavy

m=8, UNC gen., exponential-heavy
What about the task-set generator? \((n=2m)\)

→ very similar for completely different task-set generator

m=4, UNC gen., exponential-heavy

m=8, UNC gen., exponential-heavy

m=16, UNC gen., exponential-heavy
What about the task-set generator? (n=2m)

→ very similar for completely different task-set generator

m=4, UNC gen., exponential-heavy

m=8, UNC gen., exponential-heavy

m=16, UNC gen., exponential-heavy

m=24, UNC gen., exponential-heavy
Schedulability increases for larger $m$ since task count is not controlled with this task-set generator ($\rightarrow$ more cores = more tasks/core = easier problem).
What about context-switch rates? \((m=8, n=2m)\)
What about context-switch rates? (m=8, n=2m)

→ while there is an uptick in context switches, it is usually lower than under the most competitive optimal schedulers
What about context-switch rates? (m=8, n=2m)

→ while there is an uptick in context switches, it is usually lower than under the most competitive optimal schedulers.
What about context-switch rates?

→ while there is an uptick in context switches,
under the most competitive optimal

with both meta-heuristics
basic semi-partitioning
partitioning only
RUN
QPS

RUN & QPS
optimal schedulers with
fewest preemptions
(Regnier et al., 2013; Massa et al., 2016)
What about context-switch rates? \((m=8, n=2m)\)

→ while there is an uptick in context switches, it is usually lower than under the most competitive optimal schedulers.

With both meta-heuristics
- basic semi-partitioning
- partitioning only

Reasonable context-switch rate even with **period transformation** enabled.
Schedulability Experiments — Summary

![Graph showing schedulability experiments summary]

- With both meta-heuristics
- With PAF meta-heuristic
- Basic semi-partitioning
- Partitioning only

fraction of schedulable task sets
system utilization (in percent)
Empirically, near-optimal hard real-time schedulability — usually $\geq 99\%$ schedulable utilization — can be achieved with simple, well-known and well-understood, low-overhead techniques (+ a few tweaks).
Does it work in practice?

— Implementation in LITMUS<sup>RT</sup> —

www.litmus-rt.org
Global Scheduling Not Required: Simple, Near-Optimal Multiprocessor Real-Time Scheduling with Semi-Partitioned Reservations

LitmusRT
Linux Testbed for Multiprocessor Scheduling in Real-Time Systems

Linux-based Multiprocessor Research RTOS.

Actively maintained since 2006
17 public releases,
spanning 40 Linux kernel versions
Latest release: 2016.1

[2006–2011]

www.litmus-rt.org
Experiments with the Real System
Experiments with the Real System

Experiment 1: Comparison with stock LITMUS\textsuperscript{RT} schedulers

- **Partitioned** Fixed-Priority (P-FP)
- **Partitioned** Earliest-Deadline First (P-EDF)
- **Global** Earliest-Deadline First (G-EDF)
Experiments with the Real System

Experiment 1: Comparison with stock LITMUS\textsuperscript{RT} schedulers
- **Partitioned** Fixed-Priority (P-FP)
- **Partitioned** Earliest-Deadline First (P-EDF)
- **Global** Earliest-Deadline First (G-EDF)

Experiment 2: Comparison with RUN and QPS
- Latest and greatest *optimal* multiprocessor schedulers
- Implementations kindly provided by Compagnin et al. (2014, 2015)
Experiments with the Real System

Experiment 1: Comparison with stock LITMUS\textsuperscript{RT} schedulers
- **Partitioned** Fixed-Priority (P-FP)
- **Partitioned** Earliest-Deadline First (P-EDF)
- **Global** Earliest-Deadline First (G-EDF)

Experiment 2: Comparison with RUN and QPS
- latest and greatest *optimal* multiprocessor schedulers
- implementations kindly provided by Compagnin et al. (2014, 2015)

Experiment 3: Effect of Slack on Frequency of Migrations
- How effective is “flipped C=D + slack reclamation”?
Experiments with the Real System

Experiment 1: Comparison with stock LITMUS\textsuperscript{RT} schedulers
- **Partitioned** Fixed-Priority (P-FP)
- **Partitioned** Earliest-Deadline First (P-EDF)
- **Global** Earliest-Deadline First (G-EDF)

Experiment 2: Comparison with RUN and QPS
- latest and greatest **optimal** multiprocessor schedulers
- implementations kindly provided by Compagnin et al. (2014, 2015)

Experiment 3: Effect of Slack on Frequency of Migrations
- How effective is “flipped C=D + slack reclamation”?

Platform: Stress Scalability
- 44 cores: 2 × 22-core Xeon E5-2699 v4 @ 2.2 GHz
- 256 KiB private L2, 55 MiB shared L3
Experiments with the Real System

Experiment 1: Comparison with stock LITMUS$^\text{RT}$ schedulers
- **Partitioned** Fixed-Priority (P-FP)
- **Partitioned** Earliest-Deadline First (P-EDF)
- **Global** Earliest-Deadline First (G-EDF)

Experiment 2: Comparison with RUN and QPS
- latest and greatest **optimal** multiprocessor schedulers
- implementations kindly provided by Compagnin et al. (2014, 2015)

Experiment 3: Effect of Slack on Frequency of Migrations
- How effective is “flipped C=D + slack reclamation”?

Platform: Stress Scalability
- 44 cores: 2 × 22-core Xeon E5-2699 v4 @ 2.2 GHz
- 256 KiB private L2, 55 MiB shared L3

Data
- traced overhead with *Feather-Trace*, schedule with *sched-trace*
- over six billion samples collected over 12+ hours of execution
- here: **scheduling overhead** — *picking the next process to run*
Experiment 1: Comparison with Stock Schedulers

scheduling overhead measured on a 44-core Intel Xeon platform
Experiment 1: Comparison with Stock Schedulers

**scheduling overhead** measured on a 44-core Intel Xeon platform

- semi-partitioned (SP-RES) largely similar to partitioned schedulers (P-FP, P-EDF), not similar to (non-optimal) global EDF (G-EDF)
Experiment 1: Comparison with Stock Schedulers

*scheduling* overhead measured on a 44-core Intel Xeon platform

\[ \text{Observed CDF} \]

higher & to the left is better

\[ \text{processor cycles} \]

(\(~2200\) per µs)

semi-partitioned (SP-RES) largely similar to partitioned schedulers (P-FP, P-EDF), not similar to (non-optimal) global EDF (G-EDF)
Experiment 1: Comparison with Stock Schedulers

**scheduling overhead measured on a 44-core Intel Xeon platform**

Stock LITMUS\textsuperscript{RT} Schedulers

- **G-EDF**: single, global lock
- **P-FP**: per-core locks
- **P-EDF**: per-core locks
- **SP-RES**: per-core locks

\(\rightarrow\) semi-partitioned (SP-RES) largely similar to partitioned schedulers (P-FP, P-EDF), not similar to (non-optimal) global EDF (G-EDF)
Experiment 1: Comparison with Stock Schedulers

`scheduling` overhead measured on a 44-core Intel Xeon platform

- **Long Tail**
  - **hardware unpredictability:** x86, Broadwell Xeon, multicore...

→ semi-partitioned (SP-RES) largely similar to partitioned schedulers (P-FP, P-EDF), not similar to (non-optimal) global EDF (G-EDF)
Comparison of Schedulers

- 99th percentile overhead
  - SP-RES: 2,092 cycles (~1µs)
  - P-EDF: 2,150 cycles (~1µs)
  - P-FP: 2,059 cycles (~1µs)
  - G-EDF: 181,934 cycles (~82µs)

- Semi-partitioned (SP-RES) largely similar to partitioned schedulers (P-FP, P-EDF), not similar to (non-optimal) global EDF (G-EDF)

Experiment 1: Comparison with Stock Schedulers

- Overhead measured on a 44-core Intel Xeon platform

→ semi-partitioned (SP-RES) largely similar to partitioned schedulers (P-FP, P-EDF), not similar to (non-optimal) global EDF (G-EDF)
Experiment 2: Comparison with RUN and QPS scheduling overhead measured on a 44-core Intel Xeon platform
Experiment 2: Comparison with RUN and QPS scheduling overhead measured on a 44-core Intel Xeon platform

semi-partitioned (SP-RES) shows much lower overhead than implementations of RUN and QPS (latest optimal schedulers)

(RUN and QPS implementations kindly provided by Compagnin et al.)
Experiment 2: Comparison with RUN and QPS scheduling overhead measured on a 44-core Intel Xeon platform

- **RUN & Two QPS Variants**
  - QPS-C: per-processor locks
  - QPS-G: single, global lock
  - RUN: single global lock

  ➔ semi-partitioned (SP-RES) shows much lower overhead than implementations of RUN and QPS (latest optimal schedulers)

[Compagnin et al, 2014, 2015]

(RUN and QPS implementations kindly provided by Compagnin et al.)
Comparison of overhead measured on a 44-core Intel Xeon platform:

- **SP-RES** shows much lower overhead than implementations of RUN and QPS (latest optimal schedulers).

  - **SP-RES**: 2,255 cycles (~1µs)
  - **RUN**: 101,294 cycles (~46µs)
  - **QPS-G**: 135,994 cycles (~61µs)
  - **QPS-C**: 4,993 cycles (~2µs)

⇒ semi-partitioned (SP-RES) shows much lower overhead than implementations of RUN and QPS (latest optimal schedulers).

*(RUN and QPS implementations kindly provided by Compagnin et al.)*
Experiment 3: Impact of Slack Reclamation
Experiment 3: Impact of Slack Reclamation

slack reclamation is effective: more slack = fewer migrations
Experiment 3: Impact of Slack Reclamation

Number of migrations per core per second

Average amount of available slack (as percentage of WCET)

→ Slack reclamation is effective: more slack = fewer migrations
Experiment 3: Impact of Slack Reclamation

5x reduction in migration rate
if the worst case is twice as high as the average case

→ slack reclamation is effective: more slack = fewer migrations
Discussion & Conclusion
Study Limitations
Study Limitations

Software Malleability

- This schedulability study is **biased against** partitioned / semi-partitioned scheduling (as are many before it).
- *If no mapping is found, engineers may be able to refactor "large" tasks and redistribute or pipeline some functionality.*
- Example: remapping *runnables* in AUTOSAR.
Study Limitations

Software Malleability

- This schedulability study is **biased against** partitioned / semi-partitioned scheduling (as are many before it).
- *If no mapping is found, engineers may be able to refactor “large” tasks and redistribute or pipeline some functionality.*
- Example: remapping *runnables* in AUTOSAR.

Task Set Generation

- Randomly generated task sets, based on standard methods.
- Is there a *practically relevant* class of independent, implicit-deadline workloads for which all semi-partitioning heuristics consistently fail?

(I don’t think so.)
Practical Extensions
Practical Extensions

What about \textit{constrained/arbitrary} deadlines?
Practical Extensions

What about \textit{constrained/arbitrary} deadlines?

\ldots everything but the RP meta-heuristic \textit{still works}.

\ldots no optimal online schedulers exist (Fisher et al., 2010).
Practical Extensions

What about constrained/arbitrary deadlines?

...everything but the RP meta-heuristic still works.

...no optimal online schedulers exist (Fisher et al., 2010).

What about precedence constraints?
Practical Extensions

What about constrained/arbitrary deadlines?

...everything but the RP meta-heuristic still works.
...no optimal online schedulers exist (Fisher et al., 2010).

What about precedence constraints?

...can reuse uniprocessor techniques (jitter).
...can introduce additional heuristics.
Practical Extensions

What about **constrained/arbitrary** deadlines?

...everything but the RP meta-heuristic **still works**.
...no optimal online schedulers exist (Fisher et al., 2010).

What about **precedence** constraints?

...can **reuse uniprocessor techniques** (jitter).
...can introduce additional heuristics.

What about **self-suspensions**?
Practical Extensions

What about constrained/arbitrary deadlines?
...everything but the RP meta-heuristic **still works**.
...no optimal online schedulers exist (Fisher et al., 2010).

What about precedence constraints?
...can **reuse uniprocessor techniques** (jitter).
...can introduce additional heuristics.

What about self-suspensions?
...already supported (**slack**).
...implementation already supports deferrable servers.
...**semi-partitioned deferrable servers**?
Practical Extensions

What about constrained/arbitrary deadlines?
  …everything but the RP meta-heuristic still works.
  …no optimal online schedulers exist (Fisher et al., 2010).

What about precedence constraints?
  …can reuse uniprocessor techniques (jitter).
  …can introduce additional heuristics.

What about self-suspensions?
  …already supported (slack).
  …implementation already supports deferrable servers.
  …semi-partitioned deferrable servers?

What about locking?
Practical Extensions

What about **constrained/arbitrary** deadlines?

...everything but the RP meta-heuristic **still works**.
...no optimal online schedulers exist (Fisher et al., 2010).

What about **precedence** constraints?

...can **reuse uniprocessor techniques** (jitter).
...can introduce additional heuristics.

What about **self-suspensions**?

...already supported (**slack**).
...implementation already supports deferrable servers.
...**semi-partitioned deferrable servers**?

What about **locking**?

...**multiprocessor bandwidth inheritance** (MBWI).
...**spin locks** (Biondi et al., 2015).
...future work (MC-IPC, MrsP, ???).
Further Overheads and Challenges
Further Overheads and Challenges

What about migration overheads?
Further Overheads and Challenges

What about migration overheads?

...lower than under global scheduling.

...can control precisely which tasks migrate (⇒ PAF).
Further Overheads and Challenges

What about **migration** overheads?

...lower than under global scheduling.

...can **control** precisely which **tasks** migrate (➔ PAF).

What about **cache/bus/memory interference**?
Further Overheads and Challenges

What about migration overheads?

...lower than under global scheduling.
...can control precisely which tasks migrate (PAF).

What about cache/bus/memory interference?

...orthogonal concern (known techniques apply).
...no worse than under global scheduling.
Further Overheads and Challenges

What about **migration** overheads?
   ...lower than under global scheduling.
   ...can **control** precisely **which tasks** migrate (➔ PAF).

What about **cache/bus/memory interference**?
   ...orthogonal concern (known techniques apply).
   ...**no worse** than under global scheduling.

What about **energy/power/thermal** constraints?
Further Overheads and Challenges

What about **migration** overheads?

...lower than under global scheduling.
...can **control** precisely **which tasks** migrate (→ PAF).

What about **cache/bus/memory interference**?

...**orthogonal concern** (known techniques apply).
...**no worse** than under global scheduling.

What about **energy/power/thermal** constraints?

...much prior work available (uni + partitioned).
...but **race-to-idle** might favor global scheduling.
Further Overheads and Challenges

What about migration overheads?
...lower than under global scheduling.
...can control precisely which tasks migrate (→ PAF).

What about cache/bus/memory interference?
...orthogonal concern (known techniques apply).
...no worse than under global scheduling.

What about energy/power/thermal constraints?
...much prior work available (uni + partitioned).
...but race-to-idle might favor global scheduling.

What about adaptive, dynamic, or open systems?
Further Overheads and Challenges

What about migration overheads?
  ...lower than under global scheduling.
  ...can control precisely which tasks migrate (→ PAF).

What about cache/bus/memory interference?
  ...orthogonal concern (known techniques apply).
  ...no worse than under global scheduling.

What about energy/power/thermal constraints?
  ...much prior work available (uni + partitioned).
  ...but race-to-idle might favor global scheduling.

What about adaptive, dynamic, or open systems?
  ...this is were global scheduling really shines.
  ...future work on on-the-fly repartitioning and load-balancing.
Summary
Summary

**Simple Approach**
- semi-partitioned scheduling + reservations + try many heuristics
- effective: pre-assign failures (PAF), period transformation (RP)
Summary

**Simple Approach**
- semi-partitioned scheduling + reservations + try many heuristics
- effective: pre-assign failures (PAF), period transformation (RP)

**Theoretical performance: Schedulability**
- near optimal: empirically, ~99% schedulable utilization
- under same conditions as assumed in proofs of optimality
Summary

Simple Approach
- semi-partitioned scheduling + reservations + try many heuristics
- effective: pre-assign failures (PAF), period transformation (RP)

Theoretical performance: Schedulability
- near optimal: empirically, ~99% schedulable utilization
- under same conditions as assumed in proofs of optimality

Practical performance: Overheads
- similar to a plain partitioned scheduler (→ quite low)
- migration frequency can be reduced with slack reclamation
Summary

**Simple Approach**
- semi-partitioned scheduling + reservations + try many heuristics
- effective: pre-assign failures (PAF), period transformation (RP)

**Theoretical performance: Schedulability**
- near optimal: empirically, ~99% schedulable utilization
- under same conditions as assumed in proofs of optimality

**Practical performance: Overheads**
- similar to a plain partitioned scheduler (→ quite low)
- migration frequency can be reduced with slack reclamation

**Subjective Complexity**
- *Much* simpler to understand and explain than optimal schedulers
- *Much* simpler to build and maintain than optimal schedulers
- **Future work**: hopefully much simpler to extend, too.
Companion Web Page

https://mpi-sws.org/~bbb/papers/details/rtss16

Code
- illustrative pseudo code (not in paper)
- LITMUSRT scheduler plugin + libraries
- schedulability experiments (SchedCAT)

Artifact Evaluation Instructions
- how to run our experiments (quite detailed)
- also a good LITMUSRT tutorial / recipe

All Data & Graphs
- including comparisons of all individual heuristics (not in paper)
- including all “UNC style experiments “ (not in paper)
- including all overhead CDFs and plots
Thanks! Questions?

**Companion page**
https://mpi-sws.org/~bbb/papers/details/rtss16

**LitmusRT**
Linux Testbed for Multiprocessor Scheduling in Real-Time Systems

http://www.litmus-rt.org

Björn B. Brandenburg
bbb@mpi-sws.org
http://www.mpi-sws.org/~bbb
Call for Papers
International Conference on Embedded Software
October 15-20, 2017 Seoul, South Korea

The ACM SIGBED International Conference on Embedded Software (EMSOFT) brings together researchers and developers from academia, industry, and government to advance the science, engineering, and technology of embedded software development. Since 2001, EMSOFT has been the premier venue for cutting-edge research in the design and analysis of software that interacts with physical processes, with a long-standing tradition for results on cyber-physical systems, which compose computation, networking, and physical dynamics.

real-time systems — embedded software — CPS — IoT
control — testing and validation — verification
operating and runtime systems — compilers & analysis tools
security — reliability — dependability — energy — …

Abstract Submission:
March 31, 2017
Full Paper Submission:
April 7, 2017 (firm deadline)
Conference:
October 15-20, 2017
Venue:
Lotte Hotel, Seoul, South Korea
Appendix
Individual Heuristics — Partitioning Only

fraction of schedulable task sets

system utilization (in percent)

m=8, n=24

any part. heuristic

FFD

WFD
Individual Heuristics — Basic Semi-Partitioning

Fraction of schedulable task sets vs. system utilization (in percent)

- Any semi-part. heuristic
- 2WFD-C=D
- FWFD
- FFFD
- WWFD
- WFFD
- FFD-C=D

$m=8, n=16$
Individual Heuristics — Semi-Partitioning + PAF

any PAF-based heuristic
PAF: $h_1 = FWFD$
PAF: $h_1 = FFFD$
PAF: $h_1 = WWFD$
PAF: $h_1 = 2WFD-C=D$
PAF: $h_1 = WFFD$
PAF: $h_1 = FFD-C=D$

$m=8, n=16$
Individual Heuristics — Semi-Partitioning + RP

$m=8, n=16$

- any RP-based heuristic
- RP: $h = WWFD$
- RP: $h = FWFD$
- RP: $h = WFD-C=D$

Fraction of schedulable task sets vs. system utilization (in percent)
UNC Style Experiments, Varying Task Count

with both meta-heuristics
with PAF meta-heuristic
basic semi-partitioning
partitioning only
QPS (optimal)

m=8, exp. heavy
Minimum-Split Size Experiments

\[ m=8, n=16 \]

fraction of schedulable task sets

system utilization (in percent)

min-slice=100
min-slice=200
min-slice=300
min-slice=400
min-slice=500
min-slice=750
min-slice=1000
min-slice=2000
Release Overhead (1/2)

percent of samples ≤ X

processor cycles [logscale]

SP-RES
G-EDF
P-FP
P-EDF
Release Overhead (2/2)

![Graph showing release overhead over processor cycles]

Legend:
- SP-RES
- QPS-C
- QPS-G
- RUN

The graph illustrates the percent of samples ≤ X as a function of processor cycles on a log scale.
Global Scheduling Not Required: Simple, Near-Optimal Multiprocessor Real-Time Scheduling with Semi-Partitioned Reservations

**Extra Overheads (1/2)**

![Graph showing extra overheads over processor cycles](graph.png)

- **percent of samples \( \leq X \)**
- **processor cycles**
- **schedule locally**
- **migration timer**
- **subtask activation**

B. Brandenburg and M. Gül
Extra Overheads (2/2)
Percentile Plots — Schedule Overhead (1/2)
Percentile Plots — Schedule Overhead (2/2)
Percentile Plots — Release Overhead (1/2)

- SP-RES
- G-EDF (red dotted line)
- P-FP (blue dashed line)
- P-EDF (green dashed line)

Graph showing the observed overhead [cycles] on the y-axis and percentile on the x-axis.

The graph compares different scheduling algorithms for release overhead, with SP-RES having the least overhead, followed by G-EDF, P-FP, and P-EDF.
Percentile Plots — Release Overhead (2/2)

![Graph showing observed overhead for different processes](image-url)
Percentile Plots — Release Overhead (1/2)

- SP-RES migration timer
- SP-RES subtask activation

observed overhead [cycles]

percentile
Percentile Plots – Release Overhead (2/2)

SP-RES migration timer
SP-RES subtask activation
RUN reduction tree update

observed overhead [cycles]

percentile

0 10 20 30 40 50 60 70 80 90 100