Efficient Partitioning of Sporadic Real-Time Tasks with Shared Resources and Spin Locks

Alexander Wieder
Björn B. Brandenburg

Max Planck Institute for Software Systems (MPI-SWS)
Motivation

- space, weight and power constraints
- tasks with real-time requirements
- multi-core architectures
- shared resources protected by spin locks
Motivation

Our Focus: Processor Efficiency
Make best use of multi-core architectures despite shared resources protected by spin locks
Example: Autosar

 multicore architectures: partitioned fixed-priority scheduling
Example: Autosar

multicore architectures:

shared resources:
- sensors
- communication bus
- kernel objects

partitioned fixed-priority scheduling

global resources:
- non-preemptable spin locks

local resources:
- Priority Ceiling Protocol (PCP)
The Task Assignment Problem

sporadic real-time tasks

Assignment Algorithm
The Task Assignment Problem

Shared resources protected by spin locks
The Task Assignment Problem

Assignment Algorithm

tasks mapped to processor cores s.t. all deadlines met
The Task Assignment Problem

Challenge: Task sets using spin locks
The Task Assignment Problem

Challenge: Task sets using spin locks

Suppose $T_4$ assigned to core 3 instead…
The Task Assignment Problem

Challenge:
Task sets using spin locks
The Task Assignment Problem

Assignment Algorithm

Challenge:
Task sets using spin locks

Remote task can miss deadline!
The Task Assignment Problem

Assignment Algorithm

How efficient are prior heuristics...?
This Paper

Observation

Part I

Optimality matters! with shared resources, potential wasted by prior heuristics

Contribution
Optimality matters! with shared resources, potential wasted by prior heuristics

Optimal ILP-based partitioning scheme for task sets with shared resources
This Paper

Part I

Optimality matters! with shared resources, potential wasted by prior heuristics

Part II

Optimal ILP-based partitioning scheme for task sets with shared resources

Prior sharing-aware heuristics are complicated and brittle
This Paper

Part I

**Optimality matters!**
with shared resources,
potential wasted by prior heuristics

Part II

**Optimal** ILP-based partitioning scheme for task sets with shared resources

Part III

Prior sharing-aware heuristics are complicated and brittle

**Greedy Slacker:**
simple and robust heuristic
Task Model

- sporadic tasks: \( T_i : (e_i, d_i, p_i) \)

- constrained deadlines: \( d_i \leq p_i \)

- shared resources accessed in mutual exclusion

- coordinating resource access:
  - \textbf{global}: non-preemptable FIFO spinlocks
  - \textbf{local}: SRP (blocking equivalent to PCP)
Task Model

- sporadic tasks: \( T_i : (e_i, d_i, p_i) \)
- constrained deadlines: \( d_i \leq p_i \)
- shared resources accessed in mutual exclusion

- coordinating resource access:
  - \textbf{global}: non-preemptable FIFO spinlocks
  - \textbf{local}: SRP (blocking equivalent to PCP)

Multiprocessor Stack Resource Policy [1]

Part I
How efficient are prior heuristics?
Task Assignment with Heuristics

Assignment Algorithm

Task Assignment Heuristic
Schedulability Experiments

- Fraction of schedulable task sets
- Task set size
- Less idle time
- More contention
8 processors, 16 resources, critical section lengths in [1us, 100us], periods in [3ms, 33ms], 10% average task utilization

Exploring Wasted Potential

Potential left wasted!

- 8 processors, 16 resources, critical section lengths in \([1\text{us}, 100\text{us}]\),
- periods in \([3\text{ms}, 33\text{ms}]\), 10% average task utilization
Exploring Wasted Potential

Potential left wasted!

How can we exploit this potential?
What’s the smallest platform we can use?

8 processors, 16 resources, critical section lengths in [1us,100us], periods in [3ms,33ms], 10% average task utilization
Exploring Wasted Potential

Potential left wasted!

How can we exploit this potential?

What’s the smallest platform we can use?

With shared resources, optimal partitioning matters!
Part II
An Optimal ILP-based Partitioning Scheme
Optimal Task Assignment

Assignment Algorithm

optimal ILP-based assignment
Optimality

What is an optimal partitioning scheme?

If a valid partitioning under a given analysis exists, a valid partitioning can be found.
Optimality

What is an optimal partitioning scheme?

If a valid partitioning under a given analysis exists, a valid partitioning can be found.

All tasks claimed schedulable by analysis (= no task misses a deadline)
Optimality

What is an optimal partitioning scheme?

If a valid partitioning under a given analysis exists, a valid partitioning can be found.

We use the MSRP blocking analysis from Gai, Lipari, Di Natale (2001).
Basic ILP Model

Integer Linear Programming model encodes for a **fixed number of processors**:

- task assignment
- priority assignment
- constraints to enforce valid assignments
Basic ILP Model

Integer Linear Programming model encodes for a fixed number of processors:
- task assignment
- priority assignment
- constraints to enforce valid assignments

We need to encode the MSRP blocking analysis into the ILP!
Encoding Blocking in ILP

\[ R_i = \text{own execution} + \text{interference} \]

Classic fixed-priority response-time analysis  [1]

Encoding Blocking in ILP

Blocking analysis from Gai et al.:

\[ R_i = \text{own execution} + \text{spinning} + \text{own local blocking} + \text{interference} \]

Blocking under classic MSRP analysis from Gai et al.
Encoding Blocking in ILP

Blocking analysis from Gai et al.:

\[ R_i = \text{own execution} + \text{spinning} + \text{own local blocking} \]

\[ \times \left( \text{#higher-prio jobs} \times \left( \text{execution cost} + \text{spinning} \right) \right) \]
Encoding Blocking in ILP

Blocking analysis from Gai et al.:

\[ R_i = \text{own execution} + \text{spinning} + \text{own local blocking} \]

interference \( \times \left( \text{execution cost} + \text{spinning} \right) \)

depends on \( R_i \), priority/locality!

depends on locality!
Encoding Blocking in ILP

Blocking analysis from Gai et al.: 

\[ R_i = \text{own execution} + \text{spinning} + \text{own local blocking} \]

\[ \#\text{higher-prio jobs} \times \left( \text{execution cost} + \text{spinning} \right) \]

depends on \( R_i \), priority/locality! 

depends on \text{locality}!
Encoding Blocking in ILP

How can we express blocking in purely linear terms?

#higher-prio jobs \( R_i \), priority/locality!

interference

\[ \text{variables} \]

depends on locality!

\[ \text{variables} \]

depends on

\[ \text{variables} \]

### how can we express blocking in purely linear terms?

\[ R_i = \text{own execution} + \text{spinning} + \text{interference} \]

 Related Jobs \( x \) (execution cost + spinning)
Encoding Blocking in ILP

How can we express blocking in purely linear terms?

Transform to multiplication of variables with constants!

depends on \( r_i \), priority/locality!

interference
How long can $T_1$‘s job spin?

![Diagram showing the relationship between tasks $T_1$, $T_2$, $T_3$, $T_4$, and lines $l_1$ and $l_2$ across different cores.](image-url)
Encoding Spinning in Linear Terms

spinning =

\[ T_1^2 T_2 T_3 T_4 \]

Diagram:
- Core 1: \( T_1 \)
- Core 2: \( T_3 \)
- Core 3: \( T_4 \)
- Core 4: \( T_2 \)

Connections:
- \( l_1 \) connects to \( T_1 \) and \( T_2 \)
- \( l_2 \) connects to \( T_3 \) and \( T_4 \)
Encoding Spinning in Linear Terms

\[ \text{spinning} = \text{waiting for } T_1 + \text{waiting for } T_2 \]

Diagram:
- Core 1: \( T_1 \) and \( l_1 \)
- Core 2: \( T_3 \)
- Core 3: \( T_4 \) and \( l_2 \)
- Core 4: \( T_2 \)
Encoding Spinning in Linear Terms

\[
\text{spinning} = \text{waiting for } l_1 + \text{waiting for } l_2
\]

= \text{waiting for core 4} + \text{waiting for core 2} + \text{waiting for core 3}

\[
T_1 \rightarrow T_2 \rightarrow l_1 \quad \text{core 4}
\]
\[
T_3 \rightarrow T_4 \rightarrow l_2 \quad \text{core 3}
\]
Encoding Spinning in Linear Terms

\[
\text{spinning} = \text{waiting for } l_1 + \text{waiting for } l_2
\]

\[
= \text{waiting for core 4} + \text{waiting for core 2} + \text{waiting for core 3}
\]

\[
\geq T_2\text{'s request length} + T_3\text{'s request length} + T_4\text{'s request length}
\]
Encoding Spinning in Linear Terms

\[ \text{spinning} = \text{waiting for } l_1 + \text{waiting for } l_2 \]

\[ \geq \]

- \text{waiting for core 4}
- \text{waiting for core 2}
- \text{waiting for core 3}

\[ T_2 \text{‘s request length} + T_3 \text{‘s request length} + T_4 \text{‘s request length} \]

\[ l_1 \rightarrow T_1 \rightarrow T_2 \rightarrow l_2 \rightarrow T_3 \rightarrow T_4 \rightarrow \text{constants} \]
Encoding Spinning in Linear Terms

spinning = waiting for \( l_1 \) + waiting for \( l_2 \)

= waiting for core 4 + waiting for core 2 + waiting for core 3

\[ T_2 \text{'s request length} = T_3 \text{'s request length} = T_4 \text{'s request length} \]

Details provided in the paper.
Making ILP-based Partitioning Practical

In the real world, we also want to...

- minimize number of processors
- objective function unused in basic ILP
Making ILP-based Partitioning Practical

In the real world, we also want to...

- **minimize number of processors**
  - objective function unused in basic ILP

- **handle precedence constraints**
  - one additional constraint per task
Making ILP-based Partitioning Practical

In the real world, we also want to...

- minimize number of processors
  - objective function unused in basic ILP

- handle precedence constraints
  - one additional constraint per task

- incorporate partial specifications
  - constrain existing helper variables
ILP Solving Overhead

4 processors

- utilization 2.0
- utilization 2.5
- utilization 3.0
solving time grows with:
- task set size
- utilization
- resource contention

ILP Solving Overhead

4 processors

solving time grows with:
- task set size
- utilization
- resource contention
ILP Solving Overhead

Solving time grows with:
- task set size
- utilization
- resource contention

Only a one-time cost for exploiting wasted potential!
ILP Solving

What can we do if we cannot afford ILP solving?

Solving time grows with:
- task set size
- utilization
- resource contention

Only a one-time cost for exploiting wasted potential!
Part III
A Simple Sharing-Aware Partitioning Heuristic
Sharing-Aware Partitioning Heuristics

Prior Sharing-Aware Heuristics:

- LNR-heuristic [1]

- Blocking-Aware Partitioning Algorithm (BPA) [2]

High-Level View of Sharing-Aware Partitioning Heuristics

- identify connected components
- assign components
- if not possible, split
- cost functions
High-Level View of Sharing-Aware Partitioning Heuristics

- identify connected components
- assign components
- if not possible, split
- cost functions
High-Level View of Sharing-Aware Partitioning Heuristics

- identify connected components
- assign components
- if not possible, split
- cost functions
High-Level View of Sharing-Aware Partitioning Heuristics

- identify connected components
- assign components
- if not possible, split
- cost functions
High-Level View of Sharing-Aware Partitioning Heuristics

- identify connected components
- assign components
- if not possible, split
  - cost functions
High-Level View of Sharing-Aware Partitioning Heuristics

- identify connected components
- assign components
- if not possible, split
- cost functions
High-Level View of Sharing-Aware Partitioning Heuristics

- identify connected components
- assign components
- if not possible, split
- cost functions

Assignment Algorithm

T

T

T

T

l

l

core 1

core 2

core 3

core 4

l

l
High-Level View of Sharing-Aware Partitioning Heuristics

- identify connected components
- assign components
- if not possible, split
- cost functions
High-Level View of Sharing-Aware Partitioning Heuristics

- identify connected components
- assign components
- if not possible, split
- cost functions
High-Level View of Sharing-Aware Partitioning Heuristics

- identify connected components
- assign components
- if not possible, split
- cost functions
High-Level View of Sharing-Aware Partitioning Heuristics

- identify connected components
- assign components
- if not possible, split
- cost functions
High-Level View of Sharing-Aware Partitioning Heuristics

- identify connected components
- assign components
- if not possible, split
- cost functions
High-Level View of Sharing-Aware Partitioning Heuristics

- identify connected components

Assignment Algorithm

Splitting requires finding a *good* cut
High-Level View of Sharing-Aware Partitioning Heuristics

- identify connected components
- Splitting requires finding a good cut
- This is not so easy...
High-Level View of Sharing-Aware Partitioning Heuristics

Can we do something simpler?

Splitting requires finding a *good* cut

This is not so easy...
Greedy Slacker

Embarrassingly simple:

- disregard graph structure
- greedily try to maximize minimum slack
Greedy Slacker

embarrassingly simple:

- disregard graph structure
- greedily try to maximize minimum slack

time until deadline is missed
Greedy Slacker

embarrassingly simple:

- disregard graph structure
- greedily try to maximize minimum slack

for each task $T_i$ in order of increasing period:
  for each processor $C_k$:
    - compute slack when $T_i$ assigned to $C_k$
      if there is no $C_r$ such that minimum slack $\geq 0$:
        fail
      else:
        assign $T_i$ to $C_r$ s.t. minimum slack is maximized
Greedy Slacker

- disregard graph structure
- greedily try to maximize minimum slack

for each task $T_i$ in order of increasing period:
  for each processor $C_k$:
    compute slack when $T_i$ assigned to $C_k$
    if there is no $C_r$ such that minimum slack $\geq 0$:
      fail
    else:
      assign $T_i$ to $C_r$ s.t. minimum slack is maximized

Works with any blocking analysis.

No cost functions!

Ignores graph structure!
Greedy Slacker

for each task in order of increasing period:
for each processor:
compute slack when assigned to
if there is no \( T_i \) such that minimum slack \( c_k \) 0:
fail
else:
assign \( T_i \) to \( C_R \) s.t. minimum slack is maximized

Works with \textbf{any} blocking analysis.
No cost functions!
Ignores graph structure!

Can this possibly work?

Works with \textbf{any} blocking analysis.
No cost functions!
Ignores graph structure!

Can this possibly work?

else:
assign \( T_i \) to \( C_R \) s.t. minimum slack is maximized
Experimental Setup

Heuristics:
- sharing-oblivious (bin-packing)
- LNR-heuristic
- BPA
- Greedy Slacker
Experimental Setup

Heuristics:
- sharing-oblivious (bin-packing)
- LNR-heuristic
- BPA
- Greedy Slacker

Configuration:
- 8 processors
- 4 shared resources
- 10% average task utilization
- each resource accessed by 25% of tasks
- 100 samples
Resource Access Patterns

Unstructured
Resource Access Patterns

Unstructured

Structured
Resource Access Patterns

Unstructured

Structured

Structured with global resources
Unstructured Resource Accesses

![Graph showing the comparison of scheduling strategies]

- Sharing-oblivious
- LNR-heuristic
- BPA
- GS

The graph plots the schedulable tasks against the task count, illustrating the performance of different scheduling algorithms.
Unstructured Resource Accesses

Greedy slacker can partition more task sets than other heuristics
Why do prior sharing-aware heuristics perform poorly in this scenario?
Unstructured Resource Accesses

Why do prior sharing-aware heuristics perform poorly in this scenario?
Unstructured Resource Accesses

Why do prior sharing-aware heuristics perform poorly in this scenario?

No convenient structure that can be exploited by heuristics!
Structured Resource Accesses
Structured Resource Accesses

Grouping of tasks and resources into functional components
Structured Resource Accesses

Greedy slacker works reasonably well
Structured Resource Accesses

LNR-heuristic can exploit structure better than any other heuristic.
Structural Resource Accesses

LNR-heuristic and BPA built to exploit structure
In practice, some resources (e.g., kernel objects) are shared among all tasks.

Structured Resource Accesses

LNR-heuristic and BPA built to exploit structure
Structured Resource Accesses with global resources

Grouping of tasks and resources into functional components, some resources access by all tasks
Structured Resource Accesses with global resources
LNR-heuristic and BPA suffer from a single global resource
Structured Resource Accesses

Greedy slacker can partition (slightly) more task sets than other heuristics.
Structured Resource Accesses with global resources

Greedy Slacker works well independent of resource access patterns!
Summary

Optimal partitioning matters in the face of shared resources protected by spin locks.

Blocking due to spin locks in the MSRP can be expressed with purely linear expressions which allows using ILP techniques.
Summary

Fast and robust sharing-aware partitioning heuristic can be embarrassingly simple.
Thanks!