Scaling Global Scheduling with Message Passing

Felipe Cerqueira
Manohar Vanga
Björn Brandenburg
Global Scheduling

Tasks can execute on any processor
Global Scheduling

**In theory**, desirable analytical properties

**In practice**, not scalable due to **high overheads**
Making Global Scheduling Practical
Making Global Scheduling Practical

We can scale global scheduling with low kernel overheads!
Making Global Scheduling Practical

Overhead reduction

Linux

Our Approach

number of processors

overheads (in ms)

8 16 24 32 48 64

Linux

Overhead reduction

Our approach
This Talk

1) Why global scheduling?
2) Current implementations
3) Root causes of overhead
4) How to scale global scheduling?
5) Evaluation
This Talk

1) Why global scheduling?
2) Current implementations
3) Root causes of overhead
4) How to scale global scheduling?
5) Evaluation
Why Global Scheduling?

Reasons

- Optimal schedulers
- Work-conserving
- Soft-real-time
- and more...
Why Global Scheduling?

Optimal real-time schedulers are global

Reasons

- Optimal schedulers
- Work-conserving
- Soft-real-time
- and more...
Why Global Scheduling?

**Reasons**

- Optimal schedulers
- Work-conserving
- Soft-real-time
- and more...

Good for **open** and **dynamic systems**

Resilient to overloads
Why Global Scheduling?

Reasons

Optimal schedulers
Work-conserving
Soft-real-time
and more...

Some global schedulers guarantee \textit{bounded tardiness} without utilization loss
Why Global Scheduling?

**Reasons**

- Optimal schedulers
- Work-conserving
- Soft-real-time
- and more...

Supports **priority inheritance**

Useful in **race-to-idle** energy conservation
Why Global Scheduling?

Reasons

Optimal schedulers
Work-conserving
Soft-real-time
and more...

Properties not fully guaranteed by Partitioned and Clustered Scheduling!
Global Schedulers in Practice

Default scheduler for Linux, QNX and VXWorks.
This Talk

1) Why global scheduling?
2) Current implementations
3) Root causes of overhead
4) How to scale global scheduling?
5) Evaluation
This Talk

1) Why global scheduling?
2) Current implementations
3) Root causes of overhead
4) How to scale global scheduling?
5) Evaluation

G-EDF as a representative of global scheduling
Comparing Two Extremes

GSN-EDF

SCHED_DEADLINE
Comparing Two Extremes

GSN-EDF
Globally shared state, single lock

Distributed state, multiple locks

SCHED_DEADLINE
Comparing Two Extremes

GSN-EDF

Globally shared state, single lock

Distributed state, multiple locks

SCHED_DEADLINE
GSN-EDF

Global-EDF with support for Suspension-based protocols and O(1) Non-preemptable sections

Link-based scheduler (Block et al., 07)
**GSN-EDF**

Global-EDF with support for Suspension-based protocols and $O(1)$ Non-preemptable sections

Link-based scheduler (Block et al., 07)  
→ allows simplified locking
GSN-EDF

Coarse-grained lock

Single task queue

T1 T5 T3 T7
GSN-EDF

Coarse-grained lock

Single task queue

Simple implementation!
How does it scale?

GSN-EDF

Coarse-grained lock

Single task queue

Simple implementation!

How does it scale?
Experimental Setup

- Intel Xeon X7550 @2.0GHz, with 64 cores
- Linux 3.10 with patches
  ➡ LITMUS^RT 2013.1 and SCHED_DEADLINE v8
- Lightweight build — disabled most drivers and debugging options
Overheads Under GSN-EDF

Scheduling Overheads

Higher is worse
Global Lock Does Not Scale!

![Graph showing overheads (in ms) vs number of processors for Maximum and Average cases with GSN-EDF]
Global Lock Does Not Scale!

Average overheads: ~0.5 ms

Overheads (in ms) vs. number of processors for Global Lock in GSN-EDF.
Global Lock Does Not Scale!

For 64 CPUs, maximum overheads of ~3 ms

Average overheads: ~0.5 ms
Comparing Two Extremes

GSN-EDF
Globally share state, single lock

Distributed state, multiple locks

SCHED_DEADLINE
SCHED_DEADLINE

Design inherited from Linux scheduler
Design inherited from Linux scheduler

- Per-CPU locks
- Per-CPU task queues
Intuition:
Fine-grained locking decreases contention

Design inherited from Linux scheduler

Per-CPU locks
Per-CPU task queues

SCHED_DEADLINE
Benefit of Fine-Grained Locking

overheads (in ms)

overheads (in µs)

number of processors

SCHED_DEADLINE

Average
 Benefit of Fine-Grained Locking

![Graph showing average overheads (in µs) versus number of processors. The line indicates average overheads below 50 µs for SCHED_DEADLINE.]

Average overheads below 50 µs!
Fine-Grained Locking Fails in the Worst Case

overheads (in ms)

number of processors

Average
Maximum

overheads (in µs)

number of processors (m)

SCHED_DEADLINE
Fine-Grained Locking Fails in the Worst Case

overheads (in ms)

number of processors (m)

Average
Maximum

overheads (in µs)

number of processors (m)

SCHED_DEADLINE
Fine-Grained Locking
Fails in the Worst Case!

Very high overheads in the worst case!
Fine-Grained vs. Coarse-Grained Locks

overheads (in ms) vs. number of processors
Fine-Grained vs. Coarse-Grained Locks

SCHED_DEADLINE (Max)

overheads (in ms)

number of processors
Fine-Grained vs. Coarse-Grained Locks

Both approaches do not scale in the worst case!
This Talk

1) Why global scheduling?
2) Current implementations
3) **Root causes of overhead**
4) How to scale global scheduling?
5) Evaluation
Fine-grained Locking:
Average Case
Fine-grained Locking:
Average Case
Fine-grained Locking: Average Case
Fine-grained Locking:
Average Case

Task

Lock
Fine-grained Locking: Average Case

Low contention!
Fine-grained Locking:
Worst Case

Locking *every* processor:
$O(m)$ iterations
Fine-grained Locking:
Worst Case

Locking every processor: $O(m)$ iterations

$O(m)$ processors already waiting for this lock

Wait queue
Fine-grained Locking: Worst Case

Locking **every** processor: $O(m)$ iterations

$O(m)$ processors **already** waiting for this lock

Wait queue
Fine-grained Locking: Worst Case

Locking *every* processor: $O(m)$ iterations

$O(m)$ processors *already* waiting for this lock

![Diagram](image-url)
Fine-grained Locking: Worst Case

Locking **every** processor: \(O(m)\) iterations

\(O(m)\) processors **already** waiting for this lock

[Diagram showing a task and a wait queue with processors locked]
Fine-grained Locking: Worst Case

Locking *every* processor: 
O(m) iterations

O(m) processors *already* waiting for this lock

O(m) iterations x O(m) blocking 
= quadratic blocking times
Peak Contention

Observation #1:
Peak Contention is more important than synchronization granularity with respect to worst-case blocking.
Cache-Line Bouncing

Cache-line ownership jumps from core to core

Scheduler state shared among all cores

GSN-EDF

SCHED_DEADLINE
Observation #2:
State sharing results in overheads due to cache-line bouncing, even if it’s distributed across cores.
Root Causes of Overhead

- Peak Contention
- Cache-Line Bouncing
This Talk

1) Why global scheduling?
2) Current implementations
3) Root causes of overhead
4) How to scale global scheduling?
5) Evaluation
Candidate Solutions

Lock-free algorithms
Candidate Solutions

Lock-free algorithms
multiple CAS in the same location, unpredictable fail-retry operations
Candidate Solutions

Lock-free algorithms

multiple CAS in the same location, unpredictable fail-retry operations

Wait-free queue of events
Candidate Solutions

Lock-free algorithms
multiple CAS in the same location, unpredictable fail-retry operations

Wait-free queue of events
complex garbage collection and serialization, didn’t reduce cache-line bouncing
Candidate Solutions

Lock-free algorithms
multiple CAS in the same location, unpredictable fail-retry operations

Wait-free queue of events
complex garbage collection and serialization, didn’t reduce cache-line bouncing

All-to-all broadcast of events
Candidate Solutions

Lock-free algorithms
multiple CAS in the same location, unpredictable fail-retry operations

Wait-free queue of events
complex garbage collection and serialization, didn’t reduce cache-line bouncing

All-to-all broadcast of events
message ordering, consensus
Reducing Cache-Line Bouncing

Scheduler State

T₆  T₇
Reducing **Cache-Line Bouncing**

**Dedicated Scheduler Processor**
- Stores the full scheduler state
- Dedicated interrupt handling

**Scheduler State**

| T6 | T7 | [Blank] |

---
Reducing Cache-Line Bouncing

**Scheduler State**
- Stores the full scheduler state
- Dedicated interrupt handling

**Client Processors**
- Only know which task they should schedule (local state)

**Dedicated Scheduler Processor**
Reducing **Cache-Line Bouncing**

**Scheduler State**
- Stores the full scheduler state
- Dedicated interrupt handling

**Local states**
- Only know which task they should schedule (local state)

**Dedicated Scheduler Processor**
- Stores the full scheduler state
- Dedicated interrupt handling

**Centralized state**
- reduces sharing
Communication with low Peak Contention

Centralized coordination
• No interaction among clients
• Low-cost communication via message passing
Communication with low Peak Contention

Centralized coordination
• No interaction among clients
• Low-cost communication via message passing

Contention limited to at most two processors

Local states

Scheduler State

T₆  T₇

T₃  T₂  T₅
Message Passing

Scheduling decision

Scheduler State

T_5 T_6 T_7

Task state change
Message Passing

Scheduling decision

Scheduler State

T_4 completed!

Task state change

P_1  P_2  P_3

T_3  T_2  T_4
Message Passing

Computing scheduling decision

Scheduler State

Task state change

Scheduling decision

P1  P2  P3
Message Passing

P3, execute T₅!

Scheduler State
T₅ T₆ T₇

Scheduling decision

Task state change

P₁ P₂ P₃
Message Passing

P3, execute $T_5$!

Scheduling decision

Scheduler State

Task state change

Message Passing

$P_1$: $T_3$

$P_2$: $T_2$

$P_3$: $T_5$

Scheduler

$T_6$ $T_7$
Implementing Messages Efficiently

- Message passing via per-cpu-socket mailboxes
- Shared-memory buffer with wait-free writes

Source code at www.litmus-rt.org
G-EDF-MP

Centralized Scheduling with Message passing
This Talk

1) Why global scheduling?
2) Current implementations
3) Root causes of overhead
4) How to scale global scheduling?
5) Evaluation
Low Scheduling Overheads

Maximum

overheads (in ms)

number of processors

GSN-EDF
G-EDF-MP
SD
Low Scheduling Overheads

Maximum

G-EDF-MP incurs low maximum scheduling overheads!
Low Scheduling Overheads
Low Scheduling Overheads

Average

<table>
<thead>
<tr>
<th>overheads (in ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
</tr>
<tr>
<td>0.2</td>
</tr>
<tr>
<td>0.3</td>
</tr>
<tr>
<td>0.4</td>
</tr>
<tr>
<td>0.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>number of processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>32</td>
</tr>
<tr>
<td>48</td>
</tr>
<tr>
<td>64</td>
</tr>
</tbody>
</table>

GSN-EDF
G-EDF-MP
SD
Low Scheduling Overheads

G-EDF-MP incurs low average overheads!
Low Scheduling Overheads

But G-EDF-MP incurs additional message-passing overheads!
Two Sources of Overhead

Message Latency

Scheduler State

T4 completed!

waiting...
Two Sources of Overhead

Computing scheduling decision

Scheduler State

T5  T6  T7

Message Callback Overhead

Message Latency

waiting...
Message-Passing Overheads

- Client Latency (Max)
- Callback Overhead (Max)
- Client Latency (Avg)
- Callback Overhead (Avg)

Message passing overheads are significant!
Message-Passing Overheads

What's the overall impact on schedulability?

Message passing overheads are significant!

What’s the overall impact on schedulability?
Overhead-Aware Analysis

Hard-real-time → Max. overheads
Schedulability test

Soft-real-time → Avg. overheads
Bounded Tardiness
Schedulability Results for 64 CPUs

Higher is better
Hard-Real-Time Schedulability

![Graph showing schedulability ratio vs. task set utilization for different algorithms: No overheads, G-EDF-MP, GSN-EDF, SD. The graph indicates how the schedulability ratio changes as the task set utilization increases.]
Hard-Real-Time Schedulability

Higher schedulability, even with additional message-passing delays
SCHED_DEADLINE does not implement dedicated interrupt handling, yielding a pessimistic analysis.

Higher schedulability, even with additional message-passing delays.

The graph shows the schedulability ratio as a function of task set utilization. The x-axis represents the task set utilization, while the y-axis represents the schedulability ratio. The graph includes lines for different schedulers:

- No overheads
- G-EDF-MP
- GSN-EDF
- SD

The graph indicates that GSN-EDF and SD have higher schedulability ratios compared to G-EDF-MP and No overheads, especially at higher task set utilizations.
**SCHED_DEADLINE** does not implement dedicated interrupt handling, yielding a pessimistic analysis.

Higher schedulability, even with additional message-passing delays.

G-EDF-MP scales well under worst-case scenarios! What about the average case?

SCHED_DEADLINE does not implement dedicated interrupt handling, yielding a pessimistic analysis.

No overheads

G-EDF-MP

GSN-EDF

G-EDF-MP scales well under worst-case scenarios! What about the average case?
Soft-Real-Time Schedulability

![Graph showing schedulability ratio vs. task set utilization]

- No overheads
- G-EDF-MP
- GSN-EDF
- SD
SCHED_DEADLINE works well in the average case, but cannot be shown to do so analytically.
SCHED_DEADLINE works well in the average case, but cannot be shown to do so analytically.

G-EDF-MP also performs well in the average case.
Global-EDF with Low Overheads

Pair-wise coordination + Message passing

Scalable G-EDF implementation up to 64 CPUs
Limitations

Dedicated scheduling processor is still a **scalability bottleneck** at extreme core counts.

→ G-EDF-MP scales **much further** than prior approaches.

G-EDF-MP is **inappropriate** for workloads that do not tolerate excessive migration overheads.

→ Migrations are inherent to global scheduling policies, *irrespective of implementation.*
This approach can be applied to global scheduling in general, not just G-EDF.
Conclusion
Conclusion

Fine-grained locking is not enough. Scalability of worst-case overheads requires avoiding peak contention and cache-line bouncing.
Conclusion

Fine-grained locking is not enough. Scalability of worst-case overheads requires avoiding peak contention and cache-line bouncing.

To reduce overheads, we used a centralized scheduler and message passing.
Conclusion

Fine-grained locking is not enough. Scalability of worst-case overheads requires avoiding peak contention and cache-line bouncing.

To reduce overheads, we used a centralized scheduler and message passing.

G-EDF-MP’s design can be applied to other global schedulers and extends the range of processor counts that can be practically supported.
Thanks!

www.litmus-rt.org

New release 2014.1 is now available!