A Predictable Execution Model for COTS-based Embedded Systems

Rodolfo Pellizzoni†  Emiliano Betti‡  Stanley Bak‡
Gang Yao‡  John Criswell‡  Marco Caccamo‡
Russel Kegley*

†University of Waterloo, Canada
‡University of Illinois at Urbana-Champaign, USA
♯Scuola Superiore Sant’Anna, Italy
*Lookheed Martin Corp., USA
Outline

1. Motivation
2. PRedictable Execution Model (PREM)
3. Evaluation
4. Conclusions
Modern safety-critical embedded systems

Traditionally safety-critical embedded systems run on federated architectures.

Nowadays, such systems use integrated architectures and demand for more CPU cycles and I/O bandwidth.
Commercial Off-The-Shelf (COTS) components

In term of cost and avg. throughput, COTS-based systems usually provide better performance than specialized HW.

Example (Bus)
- Boeing777 SAFEbus: 60 Mbit/s
- PCI Express: 16 Gbyte/s

COTS components are mainly optimized for the average case scenario and not for the worst-case:

- CPU RT scheduling is no longer sufficient to provide end-to-end temporal guarantee.
- Any shared physical resource can become a source of temporal unpredictability.
Problem #1: Memory Contention

Cache-peripheral conflict:

- Arbitration policy of Front Side Bus (FSB) is unknown and non-RT
- CPU activity can be stalled due to interference on FSB
- Contention for access to main memory can greatly increase a task worst-case computation time!
Problem #1: Memory Contention

Cache-peripheral conflict:
- Arbitration policy of Front Side Bus (FSB) is unknown and non-RT
- CPU activity can be stalled due to interference on FSB
- Contention for access to main memory can greatly increase a task worst-case computation time!

Integrating COTS hardware within a hard real-time system is a serious challenge!
Experiment: Task and Peripherals

- Experiment on Intel platform
- PCI-X 133MHz, 64 bit fully loaded by traffic generator peripheral
- Task suffers continuous cache misses
- Up to 44% wcet increase
problem #2: I/O bus contention

- Two DMA peripherals transmitting at full speed on PCI-X bus
- Round-robin arbitration does not allow timing guarantees

<table>
<thead>
<tr>
<th>Transaction Length</th>
<th>Bandwidth (256B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No interference</td>
<td>596MB/s (100%)</td>
</tr>
<tr>
<td>128 bytes</td>
<td>441MB/s (74%)</td>
</tr>
<tr>
<td>256 bytes</td>
<td>346MB/s (58%)</td>
</tr>
<tr>
<td>512 bytes</td>
<td>241MB/s (40%)</td>
</tr>
</tbody>
</table>
problem #2: I/O bus contention

- Two DMA peripherals transmitting at full speed on PCI-X bus
- Round-robin arbitration does not allow timing guarantees
problem #2: I/O bus contention

- Two DMA peripherals transmitting at full speed on PCI-X bus
- Round-robin arbitration does not allow timing guarantees
problem #2: I/O bus contention

- Two DMA peripherals transmitting at full speed on PCI-X bus
- Round-robin arbitration does not allow timing guarantees
Integrating COTS within a real-time system

We propose a **P**Redictable **E**xecution **M**odel (PREM)

**Key aspects of PREM:**
- real-time embedded applications should be compiled according to a new set of rules to achieve predictability
- high-level coscheduling should be enforced among all active components of a COTS system

Contention for accessing shared resources is implicitly resolved by the high-level co-scheduler without relaying on low-level, non-real-time arbiters
PREM challenges

Several challenges had to be overcome to realize PREM:

- I/O peripherals contend for bus and memory in an unpredictable manner
  ⇒ *Real-time bridge* [Bak et al., RTSS 2009]

- Memory access patterns of tasks exhibit high variance:
  - predict a precise pattern of cache fetches is very difficult
  - conservative assumptions lead to pessimistic schedulability analysis
  ⇒ *new PRedictable Execution Model*

- COTS arbiters usually achieve fairness instead of real-time performance
  ⇒ *high-level coscheduling among active components*
PREM overview (1/2)

PREM is a *novel execution model* with following main features:

- jobs are divided into a sequence of non-preemptive scheduling intervals
- scheduling intervals are divided in two classes:
  - **compatible interval**: compiled and executed normally (backward compatible). Cache misses can happen at any time and code can use OS system calls
  - **predictable interval**: specially compiled and executed predictably by prefetching all required data at the beginning of the interval itself
PREM overview (2/2)

PREM is a novel execution model with following main features:

- execution time of predictable intervals is kept constant (monitoring a CPU time counter at run-time)
  \[\Rightarrow \text{to provide rt guarantee for I/O flows}\]

- peripherals access bus and main memory only during predictable intervals
  - coscheduling: CPU sends scheduling messages to a peripheral scheduler to enable system-level coscheduling
  - real-time bridge: rt bridges (one per peripheral) buffer incoming traffic from each peripheral and deliver it predictably to main memory according to a global coschedule.

- rt bridges raise interrupts only during compatible intervals
  \[\Rightarrow \text{rt bridge allows for synchronous delivery of interrupts}\]
PREM predictable interval

CPU Execution
Cache fetches and replacements
Peripheral data transfers

memory phase
execution phase

\[ e_{i,j}^{\text{mem}} \]
\[ e_{i,j}^{\text{exec}} \]

\[ e_{i,j} \text{(constant)} \]

Constant WCET is enforced to provide rt guarantee for I/O flows
**PREM system architecture**

- **Peripheral scheduler** receives scheduling messages from CPU and enforces I/O and CPU coscheduling.

- **Real-time bridge** can independently acknowledge interrupts raised by peripheral, store incoming data in its local buffer and deliver them predictably according to PREM rules.
PREM coscheduling

\( \tau_1 \)

\( \tau_2 \)

\( \tau_{I/O}^1 \)

\( \tau_{I/O}^2 \)

\( S_{1,1}, S_{1,2}, S_{1,3} \)

\( S_{2,1}, S_{2,2}, S_{2,3}, S_{2,4} \)

: compatible interval

: memory phase

: execution phase

: I/O flow

input data

output data

0 10 20 30 40 50 60
Peripheral Scheduler

- Peripheral scheduler receives $data_{rdy_i}$ information from Real-time Bridges and output $block_i$ signals.
- Servers provide isolation by enforcing a timing reservation.
- Fixed priority, cyclic executive, etc. can be implemented in HW with very little area.
Peripheral Scheduler

- Implicit schedule of I/O flows (arbitration resolved at high-level)
- I/O flows are scheduled according to rt priorities by the peripheral scheduler
- No need to know low-level parameters!
Real-time Bridge

- FPGA System-on-Chip design with CPU, external memory, and custom DMA Engine
- The controlled peripheral reads/writes to/from Local RAM instead of Main Memory (completely transparent to the peripheral)
- DMA Engine transfers data from/to Main Memory to/from Local RAM.
Implemented prototype

- Xilinx TEMAC 1Gb/s ethernet card (integrated on FPGA)
- Optimized virtual driver implementation with no software packet copy (PowerPC running Linux)
- Full VHDL HW code and SW implementation available
PREM programming model

PREM can be used with a high level programming language, like C by:

- setting some **programming guidelines**
- using a **modified compiler**
PREM programming model

PREM can be used with a high level programming language, like C by:

- setting some **programming guidelines**
- using a **modified compiler**

PREM with C language:

1. The programmer provides annotations
2. The modified compiler generates code that:
   - performs cache prefetching at beginning of each predictable interval
   - enforces constant execution time for each predictable interval
   - sends coscheduling messages to peripheral scheduler
### Programmer vs Compiler

#### What the programmer does
- profile the code to identify where the program spends most of its execution time (to make it predictable)
- estimate WCET of each interval (static analysis)
- identify global memory regions (future work for compiler)
- put annotations at the begin and end of each interval

#### What the compiler does
- prefetch code and stack
- prefetch global memory
- insert code to instruct the peripheral scheduler
- enforce constant execution time for predictable intervals
PREM constraints

Programming constraints for predictable intervals:

- no link-based data structures (like a binary tree)
- programmer indicates the accessed memory regions
- no system calls
- no stack allocation within loops
- no recursive function calls
- no indirect function calls that are not decidable at compile time
- all prefetched memory regions (global, code, and stack) must fit in cache

These constraints are not significantly more restrictive than those imposed by state-of-the-art static analysis
Porting Legacy Applications

Adding annotations to correctly split the code into predictable blocks requires some low-level knowledge of cache parameters. However:

- data-intensive real-time applications are already optimized based on hardware architecture
- compiler could help the programmer to:
  - create predictable blocks
  - verify if some restrictions are violated (using static analysis)
  - identify used global memory regions
  - verify that all prefetched memory regions fit in cache
- if some code/function cannot be made predictable, it can always run as compatible interval
We realized a prototype system for PREM. In particular, we developed:

- a **peripheral scheduler** that is connected to the PCIe bus
- a **peripheral scheduler driver** that allows the CPU to send *scheduling messages* for I/O and CPU co-scheduling
- a real-time bridge that can **buffer I/O data and interrupts**
- a **compiler pass** (using LLVM Compiler Infrastructure) that implements the described compiler techniques
Peripheral Scheduling: example of unscheduled I/O flows

- 3 x Real-Time Bridges, 1 x Traffic Generator with synthetic traffic

<table>
<thead>
<tr>
<th>Peripheral</th>
<th>Transfer Time</th>
<th>Budget</th>
<th>Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>RT Bridge</td>
<td>7.5ms</td>
<td>9ms</td>
<td>72ms</td>
</tr>
<tr>
<td>Generator</td>
<td>4.4ms</td>
<td>5ms</td>
<td>8ms</td>
</tr>
</tbody>
</table>

Utilization 1, harmonic periods.

Unscheduled I/O flows suffer deadline miss!
Peripheral Scheduling: example of rt scheduled I/O flows

- 3 x Real-Time Bridges, 1 x Traffic Generator with synthetic traffic
- Rate Monotonic with Sporadic Servers

<table>
<thead>
<tr>
<th>Peripheral</th>
<th>Transfer Time</th>
<th>Budget</th>
<th>Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>RT Bridge</td>
<td>7.5ms</td>
<td>9ms</td>
<td>72ms</td>
</tr>
<tr>
<td>Generator</td>
<td>4.4ms</td>
<td>5ms</td>
<td>8ms</td>
</tr>
</tbody>
</table>

No deadline misses with peripheral scheduler

I/O Trace with the Real-time I/O Management System
PREM: evaluation testbed

Testbed: Intel Q6700 CPU (4 cores, two 4MB L2 caches) with Linux 2.6.31. To emulate a uni-processor system we used following setting:

- 2 cores (sharing the same L2 cache) handle all the system activities, non critical drivers, and non-real-time tasks
- 1 core (dedicated cache) runs the rt tasks (and related drivers)
- 1 core is turned off
- CPU frequency is set to 1GHz
- memory bandwidth is 1.8Gbytes/sec
- speculative CPU hardware prefetcher disabled
- real-time tasks use 4MB page size (multiple of cache way)

⇒ same cache line index for virtual and physical addresses no matter what is the page allocation policy
To test correctness and applicability of proposed PREM compiler, we tested it on several benchmarks:

- a DES cypher benchmark
- a JPEG Image Encoding benchmark
- the automotive program group of MiBench (6 benchmarks)
DES Benchmark

<table>
<thead>
<tr>
<th>Input bytes</th>
<th>4K</th>
<th>8K</th>
<th>32K</th>
<th>128K</th>
<th>512K</th>
<th>1M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-PREM miss</td>
<td>151</td>
<td>277</td>
<td>1046</td>
<td>4144</td>
<td>16371</td>
<td>32698</td>
</tr>
<tr>
<td>PREM prefetch</td>
<td>255</td>
<td>353</td>
<td>1119</td>
<td>4185</td>
<td>16451</td>
<td>32834</td>
</tr>
<tr>
<td>PREM exec-miss</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>104</td>
</tr>
</tbody>
</table>

**Table**: DES benchmark cache misses

Key result is **predictability**: during the execution phase of a real-time task, I/O flows do not affect its timing behavior.
JPEG Image Encoding benchmark

<table>
<thead>
<tr>
<th>JPEG(1 Mpix)</th>
<th>JPEG(8 Mpix)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREM</td>
<td>Non-PREM</td>
</tr>
<tr>
<td>810</td>
<td>588</td>
</tr>
<tr>
<td>13</td>
<td>19</td>
</tr>
<tr>
<td>778</td>
<td>3039</td>
</tr>
<tr>
<td>1736</td>
<td>1612</td>
</tr>
<tr>
<td>19</td>
<td>3110</td>
</tr>
</tbody>
</table>

Table: JPEG results without peripheral traffic

- 80% of the execution time was spent in function `compress_data()`
- `compress_data()` was recompiled as predictable interval
- few residual cache misses are due to random cache replacement policy
### Automotive program group of MiBench

<table>
<thead>
<tr>
<th></th>
<th>PREM</th>
<th></th>
<th>Non-PREM</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>prefetch</td>
<td>exec-miss</td>
<td>time(µs)</td>
<td>miss</td>
</tr>
<tr>
<td>qsort</td>
<td>3136</td>
<td>3</td>
<td>2712</td>
<td>3135</td>
</tr>
<tr>
<td>susan_smooth</td>
<td>313</td>
<td>2</td>
<td>7159</td>
<td>298</td>
</tr>
<tr>
<td>susan_edge</td>
<td>680</td>
<td>4</td>
<td>3089</td>
<td>666</td>
</tr>
<tr>
<td>susan_corner</td>
<td>3286</td>
<td>3</td>
<td>341</td>
<td>598</td>
</tr>
</tbody>
</table>

- All six benchmarks were recompiled as predictable interval
- Two were not data intensive, so PREM was not necessary
- Three benchmarks were well-suited for PREM
- `susan_corner` had variable size input, hence prefetching was too pessimistic
To evaluate how PREM affects task execution time, we developed two synthetic applications, `random_access` and `linear_access`:

- each scheduling interval operates on 256Kb
- computation varies between memory references to control total cache stall time
- `random_access` accesses data randomly
- `linear_access` accesses data sequentially

Following scenarios were compared:

- Predictable
- Compatible with and without I/O traffic
Random Memory Access Test

"Predictable" is up to 28% faster than "Compatible" without I/O and up to 60% faster than Compatible with I/O

- constant # of cache misses
- exec. time decreases as cache stall time increases
- DRAM behavior: adjacent addresses are served faster than random ones
- "Predictable" is insensitive to I/O traffic!
Sequential Memory Access Test

Out of order execution
assuming sequential accesses, “Compatible” exploits better
CPU out-of-order execution

DRAM mem. and burst mode
sequential accesses are served in burst mode reducing cache/IO interference suffered by a rt task

In practice we expect the impact of PREM on task execution time to be between these two cases
Conclusions and Future Work

We designed and tested PREM, a predictable task execution model. Main lessons are:

- real-time embedded applications should be compiled according to a new set of rules to achieve predictability
- high-level coscheduling should be enforced among all active components of a COTS system
We designed and tested **PREM**, a predictable task execution model. Main lessons are:

- real-time embedded applications should be compiled according to a new set of rules to achieve predictability

- high-level coscheduling should be enforced among all active components of a COTS system

As future work, we plan to:

- reduce the programmer’s effort by extending compiler capabilities

- extend PREM to multicore platforms.