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

ILLINOIS
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

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

Doppler Nav System

Night Vision

HH-60 Pave Hawk

Look-Ahead Infrared Radar

Future Capabilities

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

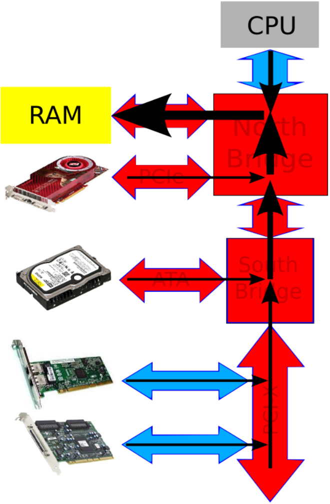
Example (Bus)

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

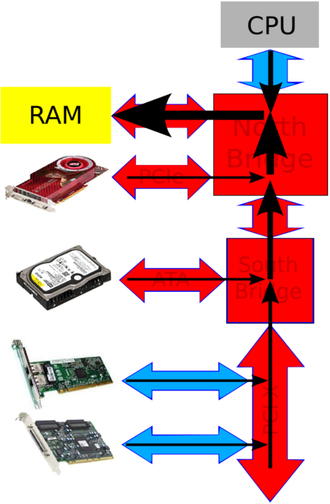
Problem #1: Memory Contention



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



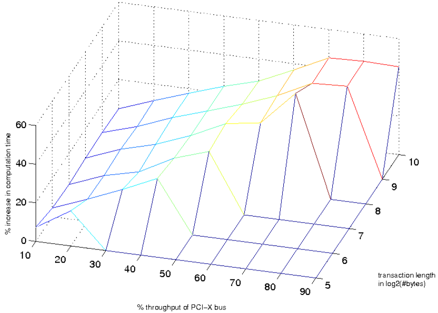
Cache-peripheral conflict:

- 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

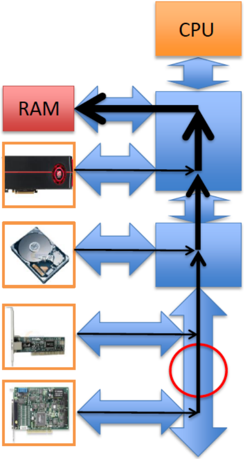
- Experiment on Intel platform
- PCI-X 133MHz, 64 bit fully loaded by **traffic generator** peripheral
- Task suffers continuous cache misses
- Up to 44% wcet increase



problem #2: I/O bus contention

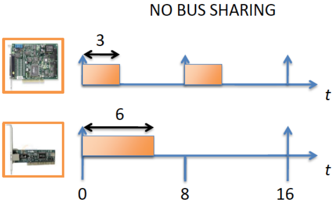
- Two DMA peripherals transmitting at full speed on PCI-X bus
- Round-robin arbitration does not allow timing guarantees

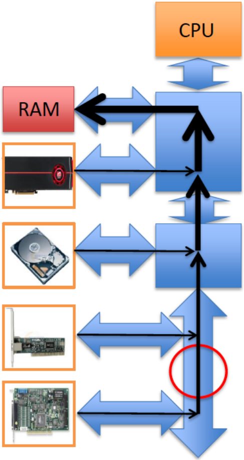
 **Transaction Length**	 **Bandwidth (256B)**
No interference	596MB/s (100%)
128 bytes	441MB/s (74%)
256 bytes	346MB/s (58%)
512 bytes	241MB/s (40%)



problem #2: I/O bus contention

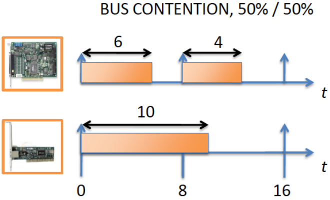
- Two DMA peripherals transmitting at full speed on PCI-X bus
- Round-robin arbitration does not allow timing guarantees

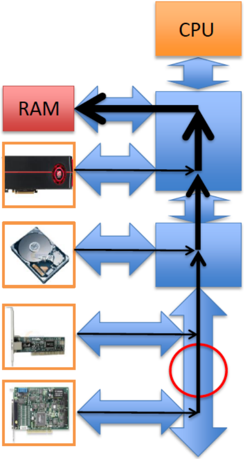




problem #2: I/O bus contention

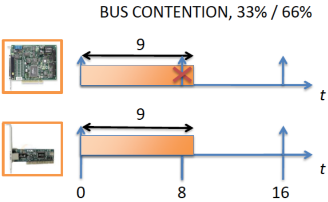
- Two DMA peripherals transmitting at full speed on PCI-X bus
- Round-robin arbitration does not allow timing guarantees

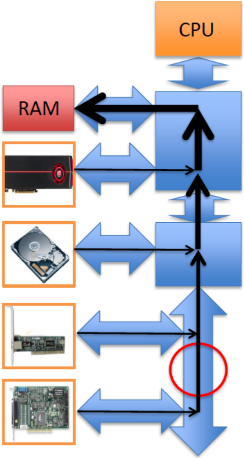




problem #2: I/O bus contention

- 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 **PRedictable Execution Model (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:

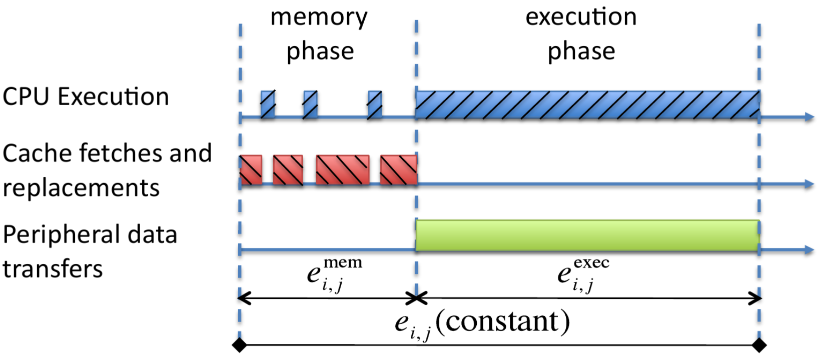
- execution time of predictable intervals is kept constant (monitoring a CPU time counter at run-time)

  ⇒ *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

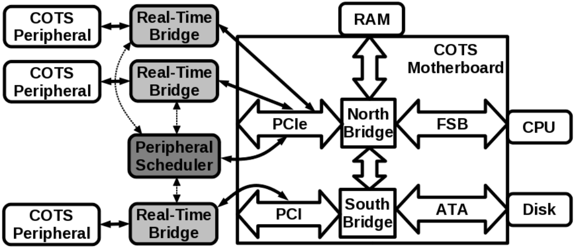
  ⇒ *rt bridge allows for synchronous delivery of interrupts*

PREM predictable interval



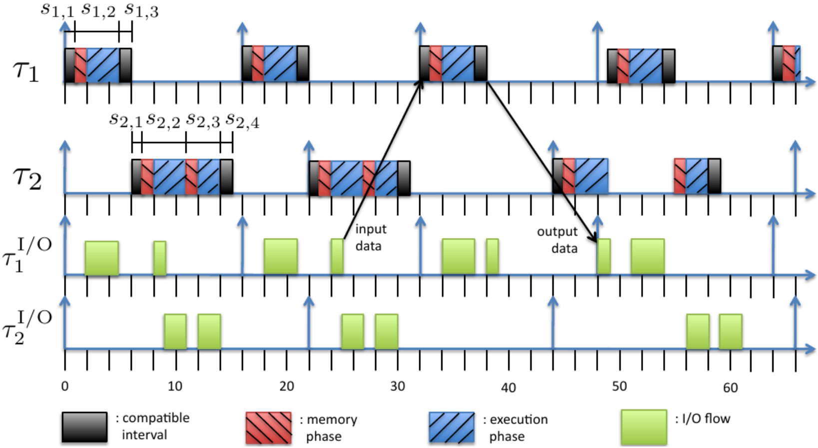
Constant WCET is enforced to provide rt guarantee for I/O flows

PREM system architecture



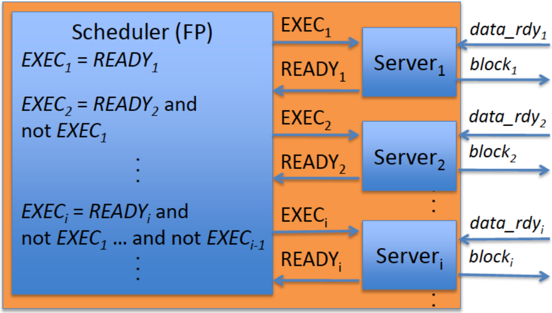
- **peripheral scheduler** receives scheduling messages from CPU and enforces I/O and CPU coscheduling
- **real-time bridge** can independently acknowledge interrupts raised by peripheral, store incoming data in its local buffer and deliver them predictably according to PREM rules

PREM coscheduling



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



Scheduler (FP)

$EXEC_1 = READY_1$

$EXEC_2 = READY_2$ and not $EXEC_1$

⋮

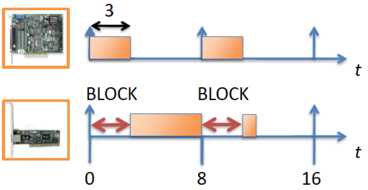
$EXEC_i = READY_i$ and not $EXEC_1$ … and not $EXEC_{i-1}$

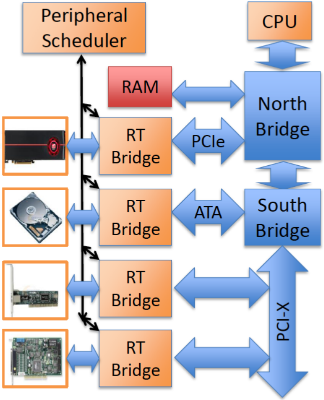
⋮

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!

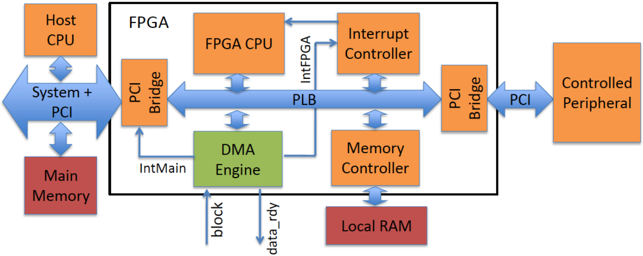
IMPLICIT SCHEDULE ENFORCEMENT





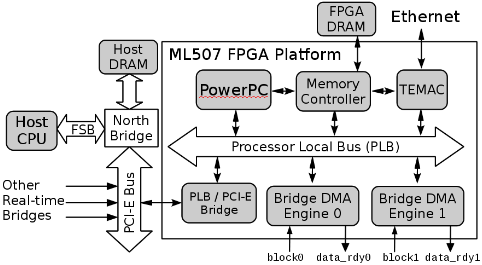
Real-time Bridge

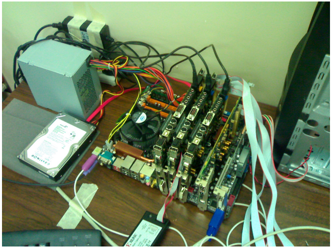
- FPGA System-on-Chip design with CPU, external memory, and custom DMA Engine
- The controlled peripheral reads/writes to/from Local RAM instead of Main Memory (completely transparent to the peripheral)
- DMA Engine transfers data from/to Main Memory to/from Local RAM.



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

PREM implementation

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

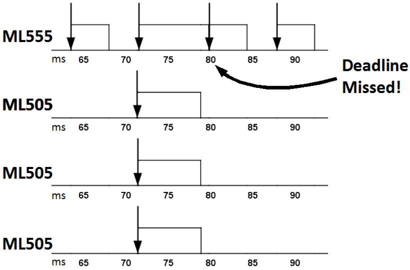
Peripheral Scheduling: example of unscheduled I/O flows

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

Peripheral	Transfer Time	Budget	Period
RT Bridge	7.5ms	9ms	72ms
Generator	4.4ms	5ms	8ms

Utilization 1, harmonic periods.

Unscheduled I/O Trace



Deadline Missed!

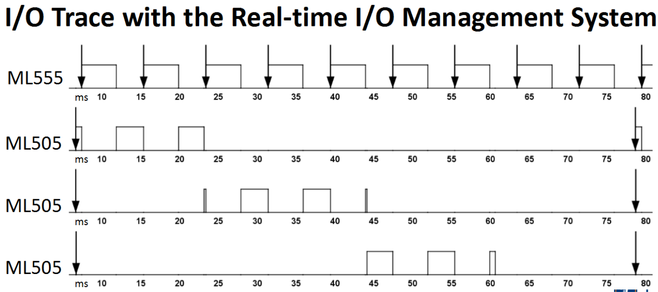
Unscheduled I/O flows suffer deadline miss!

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

Peripheral	Transfer Time	Budget	Period
RT Bridge	7.5ms	9ms	72ms
Generator	4.4ms	5ms	8ms

No deadline misses with peripheral scheduler



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*

Compiler evaluation

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

Input bytes	*4K*	*8K*	*32K*	*128K*	*512K*	*1M*
Non-PREM miss	151	277	1046	4144	16371	32698
PREM prefetch	255	353	1119	4185	16451	32834
PREM exec-miss	1	1	1	1	1	104

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

	PREM			Non-PREM	
	prefetch	exec-miss	time(μs)	miss	time(μs)
JPEG(1 Mpix)	810	13	778	588	797
JPEG(8 Mpix)	1736	19	3039	1612	3110

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

	PREM			Non-PREM	
	prefetch	exec-miss	time(μs)	miss	time(μs)
qsort	3136	3	2712	3135	2768
susan_smooth	313	2	7159	298	7170
susan_edge	680	4	3089	666	3086
susan_corner	3286	3	341	598	232

- 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

WCET experiments

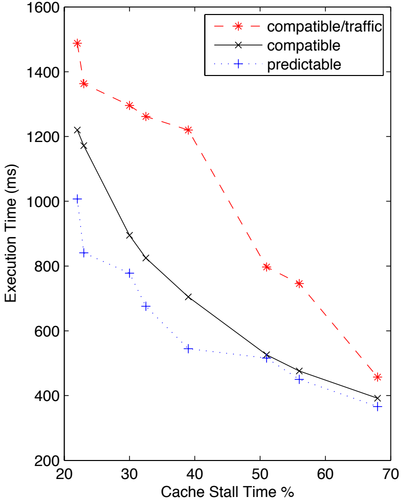
To evaluate how PREM affects task execution time, we developed two synthetic applications, `random_access` and `linear_access`:

- 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

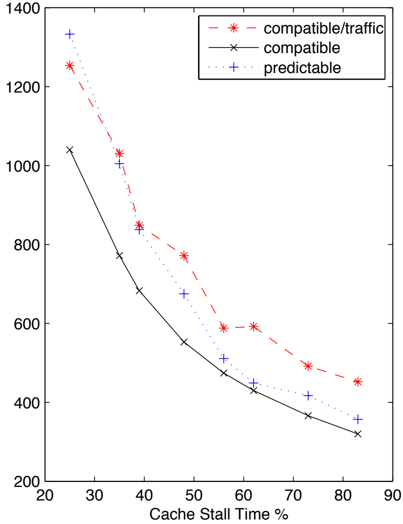
Random Memory Access Test



"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

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

As future work, we plan to:

- reduce the programmer's effort by extending compiler capabilities
- extend PREM to multicore platforms.