The Migration
of Safety-Critical RT Software
to Multicore

Marco Caccamo
University of Illinois
at Urbana-Champaign
Outline

• Motivation

• Memory-centric scheduling theory
  – Background: PRedictable Execution Model (PREM)
  – Multicore memory-centric scheduling

• Single Core Equivalence (SCE)
  – Memory bandwidth Isolation (MemGuard)
  – Cache space management (Colored Lockdown)
Real-Time Applications

- Resource intensive real-time applications
  - Real-time sensor fusion and object tracking, multimedia processing(*), real-time data analytic(**)

(*) ARM, QoS for High-Performance and Power-Efficient HD Multimedia, 2010
The Great Migration

- Multicore benefits include reduced space, weight, power, cooling and increased CPU bandwidth.

- Industry has a large body of certified single core hard real time (HRT) software proven in practice.

- HRT software will be migrated to multicore en masse. And new software will be developed to take advantage of the increased computational power.

- However, there are also great risks along the way.
The Risks of Multicore Integration

- Original hardware allows each software application to **own** all the processor resources (memory, cache, I/O bandwidth)
- Re-integration on a single multicore processor means that applications must **compete** for these same resources

Original Distributed System

New Multicore System

Software from each node is re-integrated on a single core

Applications moving from platforms where they “own” the entire node to one where they must compete for cache, memory bus, I/O resources

Source: Russell Kegley and Dennis Perlman
## Problem:
### Shared Memory Hierarchy in Multicore

<table>
<thead>
<tr>
<th>App 1</th>
<th>App 2</th>
<th>App 3</th>
<th>App 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core1</td>
<td>Core2</td>
<td>Core3</td>
<td>Core4</td>
</tr>
<tr>
<td>Memory Controller (MC)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shared Last Level Cache (LLC)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DRAM</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Space sharing**
- Access contention
- Shared banks contention

---

- **Problem:**
  - Shared hardware resources
  - OS has little control

---

On-chip network is a shared resource, but it is currently overprovisioned in embedded systems (e.g., Freescale P4080). Hence, I will not discuss about on chip network since it is still far from being a bottleneck in all of our existing studies.
Proposed Solutions

**Solutions:**

- a new clean-slate real-time scheduling approach called **memory-centric scheduling theory**.
  - the high-level idea is to effectively co-schedule cores activity, together with their memory and I/O usage
  - memory accesses are scheduled at coarse granularity
  - it leverages the Predictable Execution Model (PREM, RTAS’11)

- an engineering solution, named **Single Core Equivalence (SCE):**
  - control cores’ memory bandwidth usage (MemGuard)
  - manage cache space in a predictable manner (Colored Lockdown)
  - use a DRAM bank-aware memory allocator (Palloc, Yun et al. RTAS’14)

These are all software based solutions that don’t require any modification to the HW.
Outline

• Motivation

• Memory-centric scheduling theory
  – Background: PRedictable Execution Model (PREM)
  – Multicore memory-centric scheduling

• Single Core Equivalence (SCE)
  – Memory bandwidth Isolation (MemGuard)
  – Cache space management (Colored Lockdown)
Predictable Execution Model (PREM single core)

BACKGROUND:

• *(The idea)* The execution of a task can be distinguished between a memory phase (with cache prefetching) and a local execution phase (with cache hits)

• *(The benefit)* High-level real-time coscheduling can be enforced among all active components of a COTS system
  ➔ contention for accessing shared resources is implicitly resolved by the high-level coscheduler without relaying on low level arbiters

**PREM predictable interval**

![Diagram](image)

- **CPU Execution**
- **Cache fetches and replacements**
- **Peripheral data transfers**

Memory phase:

- $e_{i,j}^{\text{mem}}$

Execution phase:

- $e_{i,j}^{\text{exec}}$

$e_{i,j}$ (constant)

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

---

The Migration of Safety-Critical RT Software to Multicore
Multicore memory-centric scheduling

- It uses the PREM task model: each task is composed by a sequence of intervals, each including a memory phase followed by a computation phase.

- It enforces a coarse-grain memory schedule for granting memory access to each core.

- Scheduling policies:
  - fixed priority, partitioned CPU schedule, TDMA memory schedule
  - fixed priority, global CPU schedule, fixed priority memory schedule
  - dynamic priority, partitioned CPU schedule, Least Laxity First memory schedule

Example of memory-centric scheduling: TDMA memory schedule

• Assumption: fixed priority, partitioned CPU schedule

• Rule 1: enforce a coarse-grain TDMA schedule among the cores for granting access to main memory;

• Rule 2: raise scheduling priority of memory phases over execution phases when TDMA memory slot is granted;

• Rule 3: memory phases are non-preemptive.

Memory promotion
TDMA memory slot of core 1 (two cores example)

With a coarse-grained TDMA, tasks on one core can perform the memory access only when the TDMA slot is granted

Core Isolation
In an 8-core system, the memory-centric scheduling bound is superior to the contention-based scheduling bound.
Schedulability of synthetic tasks

The Migration of Safety-Critical RT Software to Multicore
Memory-centric scheduling: what is next?

Future directions for memory-centric scheduling:
- Investigate use of cache stashing (e.g., Freescale P4080)
- Investigate use of Local RAMs (e.g., Freescale MPC5777M)
- Compare different memory-centric scheduling policies (global/partitioned, static/dynamic/fully dynamic priorities)
- Investigate hybrid solutions (profile application memory usage; enforce memory-aware CPU scheduling only for memory-intensive sections of code)

Some lessons learned so far:
- Memory promotion is crucial for memory-centric scheduling (without cache stashing)
- Memory promotion + Least Laxity First seem best combination to schedule memory phases
- Slight improvement with non-preemptive memory phases
Outline

• Motivation

• Memory-centric scheduling theory
  – Background: PRedictable Execution Model (PREM)
  – Multicore memory-centric scheduling

• Single Core Equivalence (SCE)
  – Memory bandwidth Isolation (MemGuard)
  – Cache space management (Colored Lockdown)
Single Core Equivalence (SCE)

• **Goal of Single Core Equivalence:**
  – allow industry to reuse each core of a multicore chip as if it was a core in the conventional single core one.
  – allow the reuse of not only software but also the development and certification process as is.

• **Technology that implements Single Core Equivalence (SCE):**
  – control cores’ memory bandwidth usage (MemGuard)
  – manage cache space in a predictable manner (Colored Lockdown)
  – use a DRAM bank-aware memory allocator

See: [http://rtsl-edge.cs.illinois.edu/SCE/](http://rtsl-edge.cs.illinois.edu/SCE/)
The Challenge

- Schedulability analysis has been \textit{tractable} because of the fundamental assumption on \textit{constant (measured) worst case execution time} (WCET).
- That is, in a single core chip, \textit{the measured worst case execution time (WCET) when a task $T$ executes alone can be reused as is when task $T$ is running concurrently with other tasks.}
- In a multicore chip, the constant WCET assumption is \textit{very far from truth}. The conflicts in the concurrent sharing of globally available DRAM banks, memory bandwidth, last level cache, and I/O channels makes a task’s measured WCET becoming \textit{a random variable with large variance}.

- SCE’s resource partition technology restores constant WCET assumption at each core.
- SCE’s resource optimization technology greatly reduces the WCET size at each core.
The Difference

- Successful re-integration on the multicore platform requires that the applications execute with similar performance as before.
- For example, LM Space Systems tests indicated that competing with only one other core can mean a 2X increase in execution time (blue bars).

- SCE technology can control increases in measured worst case execution time (red).
- This could make the difference between a successful integration and failure.

Source: Lockheed Space Systems HWIL Testbed
The SCE architecture

The Migration of Safety-Critical RT Software to Multicore
Memory Access Pattern

- Memory access patterns vary over time
- Static resource reservation is inefficient
Memory Bandwidth Isolation

- MemGuard provides an OS mechanism to enforce memory bandwidth reservation for each core

MemGuard

• Characteristics
  – Memory bandwidth reservation system
  – Memory bandwidth: guaranteed + best-effort
  – Prediction based dynamic reclaiming for efficient utilization of guaranteed bandwidth
  – Maximize throughput by utilizing best-effort bandwidth whenever possible

• Goal
  – Minimum memory performance guarantee
  – A dedicated (slower) memory system for each core in multi-core systems
Memory Bandwidth Reservation

- Idea
  - Control interference by regulating per-core memory traffic
  - OS monitors and enforces each core’s memory bandwidth usage
    - Using per-core HW performance counter (PMC) and scheduler
    - current tick granularity: 1msec (replenishment period)
Guaranteed Bandwidth: $r_{\text{min}}$

• Definition
  – Minimum memory transfer rate: $r_{\text{min}}$
    • when requests are back-logged in the DRAM controller
    • worst-case access pattern: same bank & row miss

• System-wide reservation rule
  – up to the guaranteed bandwidth $r_{\text{min}}$

$$\sum_{1}^{m} B_i \leq r_{\text{min}}$$

m: #of cores

• Example (PC6400-DDR2*)
  – Peak B/W: 6.4GB/s
  – Measured minimum B/W: 1.2GB/s

(*) PC6400-DDR2 with 5-5-5 (RAS-CAS-CL latency setting)
Memory Bandwidth Reclaim

• Key objective
  – Utilize guaranteed bandwidth efficiently

• Regulator
  – Predicts memory usage based on history
  – Donates surplus to the reclaim manager at the beginning of every period
  – When remaining budget (assigned – donated) is depleted, tries to reclaim from the reclaim manager

• Reclaim manager
  – Collects the surplus from all cores
  – Grants reclaimed bandwidth to individual cores on demand
Hard/Soft Reservation on MemGuard

• Hard reservation (w/o reclaiming)
  – Guarantee memory bandwidth $B_i$ regardless of other cores
  – Selectively applicable on per-core basis

• Soft reservation (w/ reclaiming)
  – Does not guarantee reserved bandwidth due to potential misprediction
  – Error cases can occur due to misprediction
  – Error rate is small (shown in evaluation)

• Best-effort bandwidth
  – After all cores use their budgets, and before the next period begins, MemGuard broadcasts all cores to continue to execute
Evaluation Platform

- Intel Core2Quad 8400, 4MB L2 cache, PC6400 DDR2 DRAM
- Modified Linux kernel 3.6.0 + MemGuard kernel module
  - [https://github.com/heechul/memguard/wiki/MemGuard](https://github.com/heechul/memguard/wiki/MemGuard)
- Used the entire 29 benchmarks from SPEC2006 and synthetic benchmarks
Isolation Effect of Reservation

- Sum b/w reservation ≤ $r_{min}$ (1.2GB/s) → Isolation
  - 1.0GB/s (X-axis) + 0.2GB/s (lbm) = $r_{min}$

Core 0: 1.0 GB/s for X-axis

Core 2: 0.2 – 2.0 GB/s for lbm

Solo IPC@1.0GB/s
Effects of Reclaiming and Spare Sharing

- Guarantee foreground (SPEC@1.0GB/s)
- Improve throughput of background (lbm@0.2GB/s): 368%
Outline

• Motivation

• Memory-centric scheduling theory
  – Background: PRedictable Execution Model (PREM)
  – Multicore memory-centric scheduling

• Single Core Equivalence (SCE)
  – Memory bandwidth Isolation (MemGuard)
  – Cache space management (Colored Lockdown)
LVL3 Cache & Storage Interference

• Inter-core interference
  – The biggest issue wrt software certification
  – Fetches by one core might evict cache blocks owned by another core
  – Hard to analyze!

• Inter-task/inter-partition interference

• Intra-task interference
  – Both present in single-core systems too; intra-task interference is mainly a result of cache self-eviction.
Inter-Core Interference: Options

- **Private cache**
  - It is often not the case: majority of COTS multicore platforms have last level cache shared among cores

- **Cache-Way Partitioning**
  - Easy to apply, but inflexible
  - Reducing number of ways per core can greatly increase cache conflicts

- **Colored Lockdown**
  - Our proposed approach
  - Use coloring to solve cache conflicts
  - Fine-grained assignment of cache resources  (page size – 4Kbytes)
  - Use cache locking instructions to lock “hot” pages of rt critical tasks
    ➔ locked pages can not be evicted from cache

How Coloring Works

- The position inside the cache of a cache block depends on the value of index bits within the physical address.
- Key idea: the OS decides the physical memory mapping of task’s virtual memory pages ➔ manipulate the indexes to map different pages into non-overlapping sets of cache lines (colors)

![Diagram showing task and physical address space mapping to cache lines and indexes with cache conflicts indicated by crosses.]

- Task T1 maps its pages 1, 3, and 1 to cache lines 1, 2, and 3.
- Task T2 maps its pages 3 and 4 to cache lines 4 and 5.
- Note the cache conflicts indicated by crosses at cache lines 1 and 2.
How Coloring Works

• The position inside the cache of a cache block depends on the value of index bits within the physical address.
• Key idea: the OS decides the physical memory mapping of task’s virtual memory pages ➔ manipulate the indexes to map different pages into non-overlapping sets of cache lines (colors)

![Diagram](image)

<table>
<thead>
<tr>
<th>Task</th>
<th>Phys Addr. Space</th>
<th>Cache lines</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₁</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>T₂</td>
<td>5</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

➔ Key idea: manipulate the indexes in order to map different pages into non-overlapping sets of cache lines (colors)

➔ Manipulating indexes means playing with the physical addresses of the tasks. The aim is to obtain something like this:

= Cache conflict
How Coloring Works

• You can think of a set associative cache as an array...

32 ways

16 colors
How Coloring Works

• You can think of a set associative cache as an array...
• Using only cache-way partitioning, you are restricted to assign cache blocks by columns.
• Note: assigning one way turns it into a direct-mapped cache!
How Coloring + Locking Works

- You can think of cache as an array...
- Combining coloring and locking, you can assign arbitrary position to cache blocks independently of replacement policy
Colored Lockdown *Final goal*

- **Aimed model** - suffer cache misses in hot memory regions only **once**:
  - During the startup phase, **prefetch & lock** the hot memory regions
  - Significant **improvement** in terms of inter-core isolation and **schedulability**

![Diagram of Colored Lockdown](image)

The Migration of Safety-Critical RT Software to Multicore
Detecting Hot Regions

• In the general case, the size of the cache is not enough to keep the working set of all running rt critical tasks.

• For each rt critical task, we can identify some high usage virtual memory regions, called: hot memory regions. Such regions can be identified through profiling.

• Critical tasks do NOT color dynamically linked libraries. Dynamic memory allocation is allowed only during the startup phase.
Detecting Hot Regions

• The final memory profile will look like:

<table>
<thead>
<tr>
<th>#</th>
<th>page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1 + 0x002</td>
</tr>
<tr>
<td>B</td>
<td>1 + 0x004</td>
</tr>
<tr>
<td>C</td>
<td>25 + 0x0000</td>
</tr>
<tr>
<td>D</td>
<td>1 + 0x001</td>
</tr>
<tr>
<td>E</td>
<td>25 + 0x0003</td>
</tr>
<tr>
<td>I</td>
<td>3 + 0x0000</td>
</tr>
<tr>
<td>K</td>
<td>4 + 0x0000</td>
</tr>
<tr>
<td>P</td>
<td>6 + 0x0002</td>
</tr>
<tr>
<td>Q</td>
<td>1 + 0x0005</td>
</tr>
<tr>
<td>P</td>
<td>1 + 0x0000</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

➢ Where $A$, $B$, … is the page ranking;

➢ Where “#” is the section index;

➢ It can be fed into the kernel to perform selective Colored Lockdown

➢ How many pages should be locked per process?

➢ Task WCET reduction as function of locked pages has approximately a convex shape; convex optimization can be used for allocating cache among rt critical tasks
EEMBC Results

- **EEMBC Automotive** benchmarks
  - Benchmarks converted into periodic tasks
  - Each task has a 30 ms period

- **ARM**-based platform
  - 1 GHz Dual-core Cortex-A9 CPU
  - 1 MB L2 cache + private L1 (disabled)

- **Tasks** observed on Core 0
  - Each plotted sample summarizes execution of 100 jobs

- **Interference** generated with synthetic tasks on Core 1
EEMBC Results

- Angle to time conversion benchmark (a2time)

- Baseline reached when 4 hot pages are locked / 81% accesses caught
EEMBC Results

- Same experiment executed on 7 EEMBC benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Total Pages</th>
<th>Hot Pages</th>
<th>% Accesses in Hot Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>a2time</td>
<td>15</td>
<td>4</td>
<td>81%</td>
</tr>
<tr>
<td>basefp</td>
<td>21</td>
<td>6</td>
<td>97%</td>
</tr>
<tr>
<td>bitmnp</td>
<td>19</td>
<td>5</td>
<td>80%</td>
</tr>
<tr>
<td>cacheb</td>
<td>30</td>
<td>5</td>
<td>92%</td>
</tr>
<tr>
<td>canrdr</td>
<td>16</td>
<td>3</td>
<td>85%</td>
</tr>
<tr>
<td>rspeed</td>
<td>14</td>
<td>4</td>
<td>85%</td>
</tr>
<tr>
<td>tblook</td>
<td>17</td>
<td>3</td>
<td>81%</td>
</tr>
</tbody>
</table>
EEMBC Results

• One benchmark at the time scheduled on Core 0
• Only the hot pages are locked
Acknowledgements

- Part of this research is joint work with prof. Lui Sha, prof. Rodolfo Pellizzoni, and prof. Heechul Yun
- Graduate students and Postdocs involved in this research: Stanley Bach, Renato Mancuso, Roman Dudko, Emiliano Betti, Gang Yao
- This work was sponsored in part by NSF CNS 13-02563 and by CNS 12-19064, by Navy N00014-12-1-0046, by Lockheed Martin 2009-00524 and by Rockwell Collins C54251.

References

Conclusions

• In a multicore chip, memory controllers, last level cache, memory, on chip network and I/O channels are globally shared by cores. Unless a globally shared resource is over provisioned, it must be partitioned/reserved/scheduled.

• We proposed a set of software based solutions:
  – a new clean-slate real-time scheduling approach called memory-centric scheduling theory.
  – an engineering solution, named Single Core Equivalence (SCE)

• We demonstrated our techniques on different platforms based on Intel and ARM, and tested them against other options.

• Questions?