Multicore Real-Time Scheduling

Bjorn Andersson and Dionisio de Niz

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213
NASA related Roadmaps

C. L. Liu, “Scheduling algorithms for multiprocessors in a hard real-time environment,” JPL Space Programs Summary, pp. 37-60, 1969:
NASA related Roadmaps

NASA/TM-2013-217986/REV1, Flight Avionics Hardware Roadmap, Avionics Steering Committee, January 2014:

page (i):
“The ASC’s specific recommendations for near-term investments are: … Rad Hard Multicore Processor”

page 34:
“CD07: Advanced COTS-Based Instrument Processor…As a follow on to CD3, this C&DH subsystem will utilize future generations of COTS devices.”

Steering Committee for NASA Technology Roadmaps; National Research Council of the National Academies, “NASA Space Technology Roadmaps and Priorities: Restoring NASA’s Technological Edge and Paving the Way for a New Era in Space”:

page S-7 and S-8 in section “TOP TECHNICAL CHALLENGES”:
“C9) Improved Flight Computers: Develop advanced flight-capable devices and system software for real-time flight computing with low power, radiation-hard and fault-tolerant hardware that can be applied to autonomous landing, rendezvous and surface hazard avoidance.
Flight control
Fly to the right position
Avoid collisions with space debris other satellites
Flight control
Feedback controller
1. Sleep until the right time
2. Read sensor
3. Compute actuation command
4. Actuate command
5. Go to 1.
The delay must be at most $x$ milliseconds

Flight control
1. Sleep until the right time
2. Read sensor
3. Compute actuation command
4. Actuate command
5. Go to 1.
Question: How to verify timing of software

Outline:
1. Challenges in verifying timing of software
2. Our track record
3. Specific challenges in verifying timing of software in autonomous systems
Challenges in verifying timing of software
Challenge 1: One processor, many threads

Thread 1  Thread 2
Challenge 1: One processor, many threads

Thread 1 arrives
Deadline of Thread 1
Thread 2 arrives
Deadline of Thread 2

Thread 1
Thread 2
Challenge 1: One processor, many threads

Thread 1 arrives

Deadline of Thread 1

Thread 2 arrives

Deadline of Thread 2

Thread 1 misses its deadline.
Challenge 1: One processor, many threads

Thread 1 arrives
Deadline of Thread 1

Thread 2 arrives
Deadline of Thread 2

All deadlines are met
Challenge 1: One processor, many threads

Good idea: priority of a thread is a function of its deadline. Deadline-Monotonic (DM)
Challenge 2: Priority inversion, critical sections

Thread 1 and Thread 3 use critical section S

Assign priorities so a thread with short deadline has high priority (DM)
Challenge 2: Priority inversion, critical sections

Thread 1 and Thread 3 use critical section S

Assign priorities so a thread with short deadline has high priority (DM)
Challenge 2: Priority inversion, critical sections

Thread 1 and Thread 3 use critical section S

Assign priorities so a thread with short deadline has high priority (DM)

Thread 3: Lock S

Thread 1 arrives
Deadline of Thread 1
Thread 2 arrives
Deadline of Thread 2
Thread 3 arrives
Deadline of Thread 3

Thread 1
Thread 2
Thread 3
Challenge 2: Priority inversion, critical sections

Thread 1 and Thread 3 use critical section S

Assign priorities so a thread with short deadline has high priority (DM)

Thread 1 arrives
Deadline of Thread 1
Thread 2 arrives
Deadline of Thread 2
Thread 3 arrives
Deadline of Thread 3

Thread 1
Thread 2
Thread 3
Challenge 2: Priority inversion, critical sections

Thread 1 and Thread 3 use critical section S

Assign priorities so a thread with short deadline has high priority (DM)

Thread 1 arrives
Deadline of Thread 1

Thread 2 arrives
Deadline of Thread 2

Thread 3 arrives
Deadline of Thread 3
Challenge 2: Priority inversion, critical sections

Thread 1 and Thread 3 use critical section $S$

Assign priorities so a thread with short deadline has high priority (DM)

Thread 1: Try to Lock $S$, failed, Thread 1 is blocked

Thread 1 arrives  Deadline of Thread 1

Thread 2 arrives  Deadline of Thread 2

Thread 3 arrives  Deadline of Thread 3
Challenge 2: Priority inversion, critical sections

Thread 1 arrives
Deadline of Thread 1

Thread 2 arrives
Deadline of Thread 2

Thread 3 arrives
Deadline of Thread 3

Thread 1 and Thread 3 use critical section S

Assign priorities so a thread with short deadline has high priority (DM)
Challenge 2: Priority inversion, critical sections

Thread 1 and Thread 3 use critical section S

Assign priorities so a thread with short deadline has high priority (DM)

Thread 2 has finished; Thread 3 executes

Thread 1 arrives

Deadline of Thread 1

Thread 2 arrives

Deadline of Thread 2

Thread 3 arrives

Deadline of Thread 3

time
Challenge 2: Priority inversion, critical sections

Thread 1 and Thread 3 use critical section S

Assign priorities so a thread with short deadline has high priority (DM)

Thread 3 has finished; Thread 1 executes

Thread 1 arrives
Thread 2 arrives
Thread 3 arrives

Deadline of Thread 1
Deadline of Thread 2
Deadline of Thread 3

Thread 1
Thread 2
Thread 3
Challenge 2: Priority inversion, critical sections

Thread 1 and Thread 3 use critical section S

Assign priorities so a thread with short deadline has high priority (DM)

Thread 3 has finished; Thread 1 executes

Thread 1 arrives

Deadline of Thread 1

Thread 1 misses its deadline.

Thread 2 arrives

Deadline of Thread 2

Thread 3 arrives

Deadline of Thread 3

Thread 3 has finished; Thread 1 executes
Challenge 2: Priority inversion, critical sections

Thread 1 and Thread 3 use critical section S

Assign priorities so a thread with short deadline has high priority (DM)

Thread 1 waits for both lower priority thread

Thread 1 misses its deadline.
Challenge 2: Priority inversion, critical sections

Thread 1 and Thread 3 use critical section S

Assign priorities so a thread with short deadline has high priority (DM)

Thread 1 waits for a lower priority thread with whom it does not share a critical section

This situation almost caused a mission failure of an autonomous system (see NASA Mars Pathfinder 1997).
Challenge 3: Memory interference in multicore processors

Let us consider a system with a single processor first.
Challenge 3: Memory interference in multicore processors

Thread 1 arrives

Thread 2 arrives

Deadline of Thread 1

Deadline of Thread 2

All deadlines are met
Challenge 3: Memory interference in multicore processors

Let us migrate this software to a multiprocessor with two processors.
Challenge 3: Memory interference in multicore processors

Thread 1  Thread 2

Processors share memory bus
Processors share last-level cache

Last-level cache

Memory bank 0  Memory bank 1
Challenge 3: Memory interference in multicore processors

Thread 1
Thread 2

Processors share memory bus
Processors share last-level cache

Thread 1 can evict a cache block that Thread 2 brought into the last-level cache.
⇒ slowdown of execution
Challenge 3: Memory interference in multicore processors

Processors share memory bus
Processors share last-level cache

Thread 1 and Thread 2 may request the memory bus simultaneously but only one can be served at a time ⇒ slowdown of execution
Challenge 3: Memory interference in multicore processors

Assuming no memory contention: All deadlines are met
Challenge 3: Memory interference in multicore processors

Thread 1 arrives
Deadline of Thread 1

Thread 2 arrives
Deadline of Thread 2

With contention for resources in the memory system:
Thread 1 misses its deadline
Challenge 3: Memory interference in multicore processors

Upgrading a software system to multicore hardware can cause a deadline miss.
Challenge 4: Execution overruns

Thread 1 arrives

Deadline of Thread 1

Thread 2 arrives

Deadline of Thread 2

This is believed to be the worst-case execution time (WCET) of thread 1
Challenge 4: Execution overruns

Thread 1 arrives

Deadline of Thread 1

Thread 2 arrives

Deadline of Thread 2

This is believed to be the worst-case execution time (WCET) of thread 2
Challenge 4: Execution overruns

If a thread executes for longer than its believed worst-case execution time, then a deadline may be missed.
Challenge 5: Mode change

Thread 1  Thread 2
Challenge 5: Mode change

Autonomous system is requested to adapt
Challenge 5: Mode change

We need to prove that Thread 1 does not miss a deadline during the transition from Mode 1 to Mode 2.
Our track record
Challenge 1,2,3:

Previous work on single processor

\[ R_i = C_i + \sum_{j \in hp(i)} \left\lfloor \frac{R_i}{T_j} \right\rfloor * C_j \]

Priority ceiling protocol and priority inheritance protocol

Our work on multiprocessor

\[ R_i = C_i + \frac{1}{m} \sum_{j \in hp(i)} \left( \left\lfloor \frac{R_i}{T_j} \right\rfloor + 1 \right) * C_j \]

RM, 0.69

RM-US(0.33), 0.33*m

First analysis of priority inheritance protocol for (global) multiprocessor (RTSS’09)

First method for analyzing contention on memory bus (RTSS-WIP’09)

First coordinated cache and bank coloring (ICESS’13)

First method for analyzing contention on memory bus considering bank sharing (RTAS’14)
Challenge 4,5:

First implementation of mixed-criticality scheduler in OS-kernel VX Works, under evaluation by NASA

First locking protocol for mixed-criticality scheduling (RTAS’11)

First mode change protocol and analysis for EDF (OPODIS’08)

First mode change protocol with mode-independent tasks on a multiprocessor (ECRTS’11)
Timing challenges specific to autonomous systems
Challenge 6: The execution time of a thread is highly variable.

Challenge 7: A thread may not even terminate.

Challenge 8: The environment is unknown and hence the number of events that the software needs to process is not known before run-time.

Challenge 9: The execution of the software depends on the physical world and the physical world depends on the software.
Sharing of Multiple Hardware Resources

- Core 1
  - L1/L2

- Core 2
  - L1/L2

- Core 3
  - L1/L2

- Core N
  - L1/L2

- Last-Level Cache (L3)

- Memory Bus (and Mem Controller)

- DRAM Bank 0
- DRAM Bank 1
- DRAM Bank 2
- DRAM Bank 3
- DRAM Bank B
Need of Coordinated Protection

Need to constrain interference through each resource type

• CPU cycles
• Cache
• Memory Banks
• Memory Bus / inter-core network

Ensure no inconsistent configuration

• Configuration for one resource does not invalidate configuration of another
Cache Partitioning (Coloring)

Main Mem

Set associativity

Cache

Cache sets

One page

Address bits

16 15 14 13 12

Cache Index

6
Bank Partitioning (Coloring)

Main Mem

Address bits 20 19 18 17 16 15 14 13 12

Bank Index

Cache Index

Bank 0

Bank 1

Bank N
Cache and Bank Address Bits

Address bits

20 19 18 17 16 15 14 13 12

Cache Index

Bank Index

E.g. 2 bank bits
2 cache bits
1 shared bit

<table>
<thead>
<tr>
<th>Cache</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>
Row-Bank Address Bit Xoring Improves Coverage

If two additional bits are xor with bank bits we can get all combinations

<table>
<thead>
<tr>
<th>row</th>
<th>bank</th>
<th>Bank Colors</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>row</th>
<th>bank</th>
<th>Cache Colors</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>01</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>10</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Coordinated Cache and Bank Partitioning & Core Allocation

Avoid conflicting color assignments

Take advantage of different conflict behaviors
  • Banks can be shared within same core but not across cores
  • Cache cannot be shared within or across cores
  • Coordinated core and bank color allocation

Take advantage of sensitivity of execution time to cache
  • Task with highest sensitivity to cache is assigned more cache
  • Diminishing returns taken into account

Two algorithms explored
  • Mixed-Integer Linear Programming
Implementation of Cache+Bank Coloring

Linux / RK : Kernel Memory Manager
Memory reserves with set of bank and cache colors
Pages are classified in cache and bank colors
Added to resource sets that are attached to multiple processes/threads
Experimental Results

Cache coloring only

Our coordinated approach

- PS.canneal
- PS.stream-cluster
- PS.ferret
- PS.fluidanimate
- PS.facesim
- PS.freqmine
- PS.x264
- SPEC.leslie3d
- SPEC.mcf
- SPEC.milc
- SPEC.sphinx3
Limited Number of Private Partitions

Private partitions significantly reduces usable memory

- Number of bank/cache cells in memory
  - Number of cells = \((B*H)\).
  - Size of cell \(C = \frac{M}{B*H}\)
  - With: \(M = \) size of memory, \(B = \) # bank colors, \(H = \) # cache colors
  - E.g. Intel core i7 2600
    - \(M = 4GB, B = 16, H = 32\)
    - \(C = \frac{4GB}{16*32} = 8MB\)

- Private partitions \(\equiv\) one cell per cache color & one cell per bank color
  - Number of private partitions \(PP = \min(B,H)\)
  - E.g. Intel core i7 2600 : \(PP = \min(16,32) = 16\)

- Extreme (using all private partitions) total usable private partition memory
Allowing Sharing

In Partitioned Scheduling OK to share banks within core

- Number of banks are no longer a restriction: \( PP = H \)
- Partitions sharing banks in a core
  - \# Sets of independent partitions \( I = N \); \( N = \) number of cores
  - Memory utilization (uniform partitions) = \( \frac{M}{I} \)
- Intel Core i7 2600: \( I = N = 4 \)
  - Memory utilization (uniform partitions) = \( \frac{4\,GB}{4} = 1\,GB = 25\% \)

Need better utilization

Partitions may not be enough for number of tasks
Predictable Sharing

Exploit different sensitivity
Bounding interference
Policing and enforcement

Share among low sensitive
Isolate extremes

12x increase observed
Bank Partitioning (Coloring) + Timing Analysis

Explicitly considers the timing characteristics of major DRAM resources
- Rank/bank/bus timing constraints (JEDEC standard)
- Request re-ordering effect

Bounding memory interference delay for a task
- Combines request-driven and job-driven approaches

Software DRAM bank partitioning awareness
- Analyzes the effect of dedicated and shared DRAM banks
Response-Time Test

• Memory interference delay cannot exceed any results from the RD and JD approaches
  – We take the smaller result from the two approaches

• Extended response-time test

\[
R_{i}^{k+1} = C_i + \sum_{\tau_j \in hp(\tau_i)} \left[ \frac{R_{i}^k}{T_j} \right] \cdot C_j
\]

Classical iterative response-time test

\[
+ \min \left\{ H_i \cdot RD_p + \sum_{\tau_j \in hp(\tau_i)} \left[ \frac{R_{i}^k}{T_j} \right] \cdot H_j \cdot RD_p, \ J D_p (R_{i}^k) \right\}
\]

Request-Driven (RD) Approach

Job-Driven (JD) Approach
Observations

- Memory interference due to tasks running in other cores
- Tasks running on same core do not interfere with each other
- Collocate memory-intensive tasks on same core

Graph $G = (V_i, E_{i,j})$: $V_i = \tau_i, E_{i,j} = \text{interference}(\tau_i, \tau_j)$,

$$\text{weight}(E_{i,j}) = \frac{R_i - C_i}{T_i} + \frac{R_j - C_j}{T_j}$$

Following BFD:
1. Try to deploy first un-deployed subgraph on bin (core)
2. If cannot
   - break graph with minimum cut (minimize edge weights)
     - One piece that fits largest gap + rest
3. Add to undeployed subgraphs
Minimum-Cut Memory Interference Packing

\[
\frac{R_2 - C_2}{T_2} + \frac{R_3 - C_3}{T_3}
\]

Core 1  
Core 2  
Core 3
Minimum-Cut Memory Interference Packing
Minimum-Cut Memory Interference Packing

Core 1

Core 2

Core 3
Minimum-Cut Memory Interference Packing

Core 1

Core 2

Core 3
Minimum-Cut Memory Interference Packing

Diagram showing the interferences between tasks and their packing onto different cores.
Minimum-Cut Memory Interference Packing

Core 1

Core 2

Core 3
Minimum-Cut Memory Interference Packing
Memory-Interference Aware Task Allocation (MIAA)

Resource Conflicts for Parallelized Workloads

Parallelization

• Computation time > Deadline
  • Must parallelized to meet deadline
  • Guarantee always finish before deadline

Resource interference within a task

• Due to parallel subtasks
  • Need to share memory to communicate

Predictable sharing

• Compatible with efficient parallelized task schedulers
Parallelized Task Scheduling

Developed a staged execution model

Scheduled under Global Earliest-Deadline First

- Most efficient scheduling for staged execution
  - If task schedulable under optimal scheduler our scheduler need at most twice the speed to schedule task
Challenges for Parallelized Task Resource Management

Intra-task partitions
  • Threads with different sensitivities
  • Assign different partitions to different parts of same tasks
    • Down to different colors for each page of a task

Inter-task shared partitions
  • Shared partitions between parts of different tasks

Intra-task memory bus interference
Hardware and Software Profiling

Hardware
- Mapping of memory bits for cache and bank index
- Randomization strategies

Software
- Bound on number of memory accesses
- Temporal and spatial locality of accesses
- Techniques
  - Model checking (better term?)
  - Variable placement and access
  - Control-flow-based temporal and spatial locality
- Profiling
  - Performance counters
  - Valgrind
Contact Information

Bjorn Andersson
Senior Member of Technical Staff
Telephone: +1 412.268.9243
Email: baandersson@sei.cmu.edu