MARACAS: A Real-Time Multicore VCPU Scheduling Framework

Ying Ye, Richard West, Jingyi Zhang, Zhuoqun Cheng

Computer Science Department
Boston University
Overview

1. Introduction
2. Quest RTOS
3. Background Scheduling
4. Memory-Aware Scheduling
5. Multicore VCPU Scheduling
6. Evaluation
7. Conclusion
Motivation

- Multicore platforms are gaining popularity in embedded and real-time systems
  - concurrent workload support
  - less circuit area
  - lower power consumption
  - lower cost
Motivation

- Multicore platforms are gaining popularity in embedded and real-time systems
  - concurrent workload support
  - less circuit area
  - lower power consumption
  - lower cost

- Complex on-chip memory hierarchies pose significant challenges for applications with real-time requirements
Motivation

- Shared cache contention:
  - page coloring
  - hardware cache partitioning (Intel CAT)
  - static VS dynamic
Motivation

- **Shared cache contention:**
  - page coloring
  - hardware cache partitioning (Intel CAT)
  - static VS dynamic

- **Memory bus contention:**
  - bank-aware memory management
  - memory throttling
Contribution

- We proposed the use of foreground (reservation) + background (surplus) scheduling model
  - improves application performance
  - effectively reduces resource contention
  - well-integrated with real-time scheduling algorithms
Contribution

- We proposed the use of foreground (reservation) + background (surplus) scheduling model
  - improves application performance
  - effectively reduces resource contention
  - well-integrated with real-time scheduling algorithms

- We proposed a new bus monitoring metric that accurately detects traffic
Application

- Imprecise computation/Numeric integration
  - MPEG video decoding: mandatory to process I-frames, optional to process B- and P-frames to improve frame rate
Application

- **Imprecise computation/Numeric integration**
  - MPEG video decoding: mandatory to process I-frames, optional to process B- and P-frames to improve frame rate

- **Mixed-criticality systems running performance-demanding applications**
  - machine learning
  - computer vision
VCPU model $(C, T)$ in Quest RTOS
- $C$: Capacity
- $T$: Period
**Quest RTOS**

- VCPU model \((C, T)\) in Quest RTOS
  - \(C\): Capacity
  - \(T\): Period

- Partitioned scheduling using RMS

![Diagram of VCPU and cores](image-url)
**Quest RTOS**

- **VCPU model** $(C, T)$ in Quest RTOS
  - $C$: Capacity
  - $T$: Period

- Partitioned scheduling using RMS

- Schedulability test
  $$\sum_{i=1}^{n} \left( \frac{C_i}{T_i} \right) \leq n(\sqrt{2} - 1)$$
Background Scheduling

- VCPU enters background mode upon depleting its budget \( C \)

![Diagram showing foreground and background scheduling]

Foreground: Budget \( C \)

Background: \( T - C \)

Total Time: \( T \)
### Background Scheduling

- **VCPU enters background mode upon depleting its budget (C)**

- **Core enters background mode when all VCPUs are in background mode**

<table>
<thead>
<tr>
<th>Foreground</th>
<th>Background</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>T - C</td>
</tr>
</tbody>
</table>
Background Scheduling

- VCPU enters background mode upon depleting its budget \((C)\)

- Core enters background mode when all VCPUs are in background mode

- Background CPU Time \((\text{BGT})\): time a VCPU runs when core in background mode
Background Scheduling

- VCPU enters background mode upon depleting its budget (C)

- Core enters background mode when all VCPUs are in background mode

- Background CPU Time (BGT): time a VCPU runs when core in background mode

- Background scheduling: schedule VCPUs when core is in background mode
  - fair share of BGT amongst VCPUs on core
DRAM structure

- Introduction
- Quest RTOS
- Background
- Scheduling
- Memory-Aware Scheduling
- Multicore VCPU Scheduling
- Evaluation
- Conclusion

MARACAS
Ying Ye, Richard West, Jingyi Zhang, Zhuoqun Cheng
Prior work [MemGuard] uses "Rate Metric": number of DRAM accesses over a certain period
Memory-Aware Scheduling

- Prior work [MemGuard] uses "Rate Metric": number of DRAM accesses over a certain period
  - Bank-level parallelism
Memory-Aware Scheduling

Prior work [MemGuard] uses "Rate Metric": number of DRAM accesses over a certain period
- Bank-level parallelism
- Row buffers
Memory-Aware Scheduling

- Prior work [MemGuard] uses "Rate Metric": number of DRAM accesses over a certain period
  - Bank-level parallelism
  - Row buffers
  - Sync Effect
Sync Effect

Each task reduces its access rate by a factor of $\frac{T-t}{T}$.

Contention in $[0, t]$ remains the same.
Each task reduces its access rate by a factor of \((T-t)/T\)

Contention in \([0, t]\) remains the same
Latency Metric

- \( \text{requests} = 3, \text{occupancy} = 10 \)
Latency Metric

- requests = 3, occupancy = 10
- latency = $\frac{10}{3} = 3.3$
Latency Metric

- **UNC_ARB_TRK_REQUEST.ALL** *(requests)*: counts all memory requests going to the memory controller request queue

- **UNC_ARB_TRK_OCCUPANCY.ALL** *(occupancy)*: counts cycles weighted by the number of pending requests in the queue
Latency Metric

- **UNC_ARB_TRK_REQUEST.ALL** *(requests)*:
  counts all memory requests going to the memory controller request queue

- **UNC_ARB_TRK_OCCUPANCY.ALL** *(occupancy)*:
  counts cycles weighted by the number of pending requests in the queue

- Average latency:
  \[ \text{latency} = \frac{\text{occupancy}}{\text{requests}} \]
Memory Throttling

- When core gets throttled, background scheduling is disabled
Memory Throttling

- When core gets throttled, background scheduling is disabled

- Latency threshold: \( \text{MAX\_MEM\_LAT} \)

\[
\text{if \ latency} \geq \ \text{MAX\_MEM\_LAT} \ \text{then} \\
\text{num\_throttle} \ += +
\]
Memory Throttling

- When core gets throttled, background scheduling is disabled

- Latency threshold: MAX_MEM_LAT
  
  \[
  \text{if } \text{latency} \geq \text{MAX_MEM_LAT} \text{ then }
  \]
  
  \[
  \text{num_throttle}++
  \]

- Proportional throttling
  - Every core is throttled at some point
  - Throttled time proportional to core’s DRAM access rate
Predictable Migration

- Predictable Migration
  - Run migration thread with highest priority on each core:
    - Pushing local VCPUs to other cores (starts from highest utilization ones)
  - Only one migration thread active during a migration period
  - Its execution of its entire capacity $C$ does not lead to any other local VCPUs missing their deadlines
  - Constraint on $C$:
    \[ C \geq 2 \times E_{lock} + E_{struct} \]
Predictable Migration

- Run migration thread with highest priority on each core: pushing local VCPUs to other cores (starts from highest utilization ones)
Predictable Migration

- Run migration thread with highest priority on each core: pushing local VCPUs to other cores (starts from highest utilization ones)

- Only one migration thread active during a migration period
Predictable Migration

- Run migration thread with highest priority on each core: pushing local VCPUs to other cores (starts from highest utilization ones)

- Only one migration thread active during a migration period

- Its execution of its entire capacity \( C \) does not lead to any other local VCPUs missing their deadlines
Predictable Migration

- Run migration thread with highest priority on each core: pushing local VCPUs to other cores (starts from highest utilization ones)

- Only one migration thread active during a migration period

- Its execution of its entire capacity $C$ does not lead to any other local VCPUs missing their deadlines

- Constraint on $C$:

$$C \geq 2 \times E_{lock} + E_{struct}$$
For every core, define Slack-Per-VCPU (SPV):

$$SPV = \frac{1 - \sum_{i=1}^{n} (C_i / T_i)}{n}$$
VCPU Load Balancing

For every core, define Slack-Per-VCPU ($SPV$):

$$SPV = 1 - \frac{\sum_{i=1}^{n} (C_i/T_i)}{n}$$

<table>
<thead>
<tr>
<th>Core</th>
<th>10%</th>
<th>30%</th>
<th>60%</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCPU1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VCPU2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Slack</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$$SPV = \frac{1-(10\%+30\%)}{2} = 30\%$$
VCPU Load Balancing

- Balance Background CPU Time (BGT) used by every VCPU across cores: equalize SPVs of all cores
VCPU Load Balancing

- Balance Background CPU Time (BGT) used by every VCPU across cores: equalize SPVs of all cores
  - BGT fair sharing
VCPU Load Balancing

- Balance Background CPU Time (BGT) used by every VCPU across cores: equalize SPVs of all cores
  - BGT fair sharing
  - balanced memory throttling capability on each core
Balance Background CPU Time (BGT) used by every VCPU across cores: equalize SPVs of all cores
- BGT fair sharing
- balanced memory throttling capability on each core
Cache-Aware Scheduling

- Static cache partitioning amongst cores
  - page coloring
Cache-Aware Scheduling

- Static cache partitioning amongst cores
  - page coloring

- New API:
  
  ```
  bool vcpu_create(uint C, uint T, uint cache);
  ```
Cache-Aware Scheduling

- Static cache partitioning amongst cores
  - page coloring

- New API:
  
  ```c
  bool vcpu_create(uint C, uint T, uint cache);
  ```

- Extension of VCPU Load Balancing:
  destination core meets the cache requirement
Evaluation

MARACAS running on the following hardware platform:

<table>
<thead>
<tr>
<th>Processor</th>
<th>Intel Core i5-2500k quad-core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caches</td>
<td>6MB L3 cache, 12-way set associative, 4 cache slices</td>
</tr>
<tr>
<td>Memory</td>
<td>8GB 1333MHz DDR3, 1 channel, 2 ranks, 8KB row buffers</td>
</tr>
</tbody>
</table>
Rate VS Latency

- Micro-benchmark `m_jump`:
  ```c
  byte array[6M];
  for (uint32 j = 0; j < 8K; j += 64)
    for (uint32 i = j; i < 6M; i += 8K)
      < Variable delay added here >
      (uint32)array[i] = i;
  ```
**Rate VS Latency**

- Micro-benchmark `m_jump`:
  
  ```c
  byte array[6M];
  for (uint32 j = 0; j < 8K; j += 64)
    for (uint32 i = j; i < 6M; i += 8K)
      < Variable delay added here >
      (uint32)array[i] = i;
  ```

- Three `m_jump` (task 1,2,3) running on separate cores without memory throttling, utilization (C/T) 50%
Rate VS Latency

- Micro-benchmark m_jump:
  ```
  byte array[6M];
  for (uint32 j = 0; j < 8K; j += 64)
    for (uint32 i = j; i < 6M; i += 8K)
      < Variable delay added here >
      (uint32)array[i] = i;
  ```

- Three m_jump (task 1,2,3) running on separate cores without memory throttling, utilization (C/T) 50%

- Each run, insert a different time delay in task1 and task2, task3 has no delay
Rate VS Latency

- Record the total memory bus traffic, average memory request latency and task3’s instructions retired in foreground mode:
Record the total memory bus traffic, average memory request latency and task3’s instructions retired in foreground mode:

<table>
<thead>
<tr>
<th></th>
<th>Bus Traffic (GB)</th>
<th>Latency</th>
<th>task3 Instructions Retired ($\times 10^8$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>1128</td>
<td>228</td>
<td>249</td>
</tr>
<tr>
<td>M</td>
<td>1049</td>
<td>183</td>
<td>304</td>
</tr>
<tr>
<td>L</td>
<td>976</td>
<td>157</td>
<td>357</td>
</tr>
</tbody>
</table>
Record the total memory bus traffic, average memory request latency and task3's instructions retired in foreground mode:

<table>
<thead>
<tr>
<th></th>
<th>Bus Traffic (GB)</th>
<th>Latency</th>
<th>task3 Instructions Retired (×10^8)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>H</td>
<td>1128</td>
<td>228</td>
</tr>
<tr>
<td></td>
<td>M</td>
<td>1049</td>
<td>183</td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>976</td>
<td>157</td>
</tr>
</tbody>
</table>

Setting comparable thresholds:
- rate-based: derived from Bus Traffic (1128/time)
- latency-based: from Latency (228)
Rate VS Latency

- Record the total memory bus traffic, average memory request latency and task3’s instructions retired in foreground mode:

<table>
<thead>
<tr>
<th></th>
<th>Bus Traffic (GB)</th>
<th>Latency</th>
<th>task3 Instructions Retired ($\times 10^8$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>1128</td>
<td>228</td>
<td>249</td>
</tr>
<tr>
<td>M</td>
<td>1049</td>
<td>183</td>
<td>304</td>
</tr>
<tr>
<td>L</td>
<td>976</td>
<td>157</td>
<td>357</td>
</tr>
</tbody>
</table>

- Setting comparable thresholds:
  - rate-based: derived from Bus Traffic (1128/time)
  - latency-based: from Latency (228)

- Last column serves as reference, showing the expected performance of task3 using the corresponding thresholds
Rate VS Latency

- Repeat experiment with memory throttling enabled and fixed delay for task1/task2
Repeat experiment with memory throttling enabled and fixed delay for task1/task2
Conclusion

- MARACAS uses background time to improve task performance; when memory bus is contended, it gets disabled through throttling
MARACAS uses background time to improve task performance; when memory bus is contended, it gets disabled through throttling.

MARACAS uses a latency metric to trigger throttling, outperforming prior rate-based approach.
Conclusion

- MARACAS uses background time to improve task performance; when memory bus is contended, it gets disabled through throttling.

- MARACAS uses a latency metric to trigger throttling, outperforming prior rate-based approach.

- MARACAS fairly distributes background time across cores, for both fairness and better throttling.
Thank you!

Ying Ye
Boston University