Exploring and Understanding Multicore Interference from Observable Factors

Benjamin Lesage, David Griffin, Iain Bate and Frank Soboczenski
Critical systems

- Safety is an important concern for critical systems
  - Failures can have catastrophic consequences
- Evidence has to be gathered to verify the system fits its requirements
- Systems are built for robustness
  - The ability to withstand perturbations, faults and variations
  - Evidence of robustness also needs to be gathered
Temporal constraints require the timely completion of tasks
- The behaviour of a task depends on the underlying platform
  - Modeled or measured as part of a timing analysis
- Under correct assumptions, tasks can be analysed independently
  - E.g. assuming worst-case input states
- Push towards multicore platforms for efficiency
  - Off-core resources are shared between cores
  - Multiple cores execute tasks in parallel
Multicore

- Shared resources create new interference channels:
  - Concurrent modifications of a resource state
  - Arbitration delays on concurrent accesses to a resource
- Co-runners cannot be analysed independently without precautions
  - Segregation is costly
  - Segregation can be imperfect
Objective: Assess the impact of interferences on a task
- Identify the interference channels that need to be tackled
- Evaluate the benefits of mitigation schemes

- Rely on existing performance monitoring infrastructure (PMC)
  - Limit requirements on initial platform knowledge

- Feature selection: reduce a dataset by extracting the most important features
  - Capture contributors to execution time variability

- Rely on systematic exploration of interference space
  - Cover a range of interference scenarios
  - Challenge assumptions on worst-case scenarios
Overview

1. Feature Selection
   a. Principal Components Analysis (PCA)
   b. Requisites

2. Evaluation platform

3. Analysis Framework
   a. Synthetic tasks
   b. Data Collection
   c. Analysis process

4. Evaluation
   a. Sources of variability
   b. Impact of interferences
PCA

- Split factors in the dataset into Principal Components (PC)
  - Capture the main axes of variance
- Each PC has a loading
  - Contribution of the PC to the overall variance
  - Orders PC from most to least relevant
- Each factor has a loading on its PC
  - Represent the correlation of factors to a PC

- Focus on high loading PC correlated to execution time
  - Higher loading PC capture the most variance
  - Higher loading factors have the most impact on the PC
PCA

- Split factors in the dataset into Principal Components (PC)
  - Capture the main axes of variance
- Each PC has a loading
  - Contribution of the PC to the overall variance
  - Orders PC from most to least relevant
- Each factor has a loading on its PC
  - Represent the correlation of factors to a PC

- **Focus on high loading PC correlated to execution time**
  - Higher loading PC capture the most variance
  - Higher loading factors have the most impact on the PC
PCA

- Split factors in the dataset into Principal Components (PC)
  - Capture the main axes of variance
- Each PC has a loading
  - Contribution of the PC to the overall variance
  - Orders PC from most to least relevant
- Each factor has a loading on its PC
  - Represent the correlation of factors to a PC

- **Focus on high loading PC correlated to execution time**
  - Higher loading PC capture the most variance
  - Higher loading factors have the most impact on the PC
Requisites

Good quality features are identified from a good quality dataset

- Consider a wide variety of observation scenarios
  - Guarantee that existing variability in the system can be analysed

- Capture all candidate factors in the dataset
  - Allow the identification of interactions between observed factors

- Observe multiple active interference channels
  - Capture correlation between channels
TC27x

- Three asymmetric cores
  - Feature local memories
- Delays on arbitration on Cross Bar slaves
- Ad-hoc instrumentation
  - Capture timer and PMCs
  - Stored in local buffers
  - Fetched through debug interface
- Focus on RAM accesses
  - Round robin arbitration
  - Code in scratchpad
  - Data objects mapped to RAM
Synthetic tasks

- Rely on synthetic tasks to generate contention
  - Control on the level and channel of interferences
  - Requires basic knowledge of the target platform

- Generate an influx of conflicting accesses
  - Run as background tasks, pre-emptible by analysed tasks

- Allow systematic testing of interference channel, level, and patterns
  - Periodic reconfiguration triggered on system idle tick
contender()

0x00: Loop:
0x01:   access mem[01]
0x02:   access mem[02]
0x03:   access mem[03]
0x04:   access mem[04]
<...>
0xFD:   access mem[FD]
0xFE:   access mem[FE]
0xFF:   access mem[FF]

reconfigure()

P := permutation([0x01:0xFF])
I := rand([min_inter:max_inter])
For j in P[0x01:I]
    mem[j] := RAM
For j in P[I:0xFF]
    mem[j] := SCRATCHPAD
contender()

0x00: Loop:
0x01: access mem[01]
0x02: access mem[02]
0x03: access mem[03]
0x04: access mem[04]
  <...
0xFD: access mem[FD]
0xFE: access mem[FE]
0xFF: access mem[FF]

reconfigure()

P := permutation([0x01:0xFF])
I := rand([min_inter:max_inter])

For j in P[0x01:I]
  mem[j] := RAM
For j in P[I:0xFF]
  mem[j] := SCRATCHPAD

Loop over local or RAM accesses
Focus on TC27x RAM arbitration
contender()

0x00: Loop:
0x01: access mem[01]
0x02: access mem[02]
0x03: access mem[03]
0x04: access mem[04]
<...>
0xFD: access mem[FD]
0xFE: access mem[FE]
0xFF: access mem[FF]

reconfigure()

P := permutation([0x01:0xFF])
I := rand([min_inter:max_inter])

For j in P[0x01:I]
    mem[j] := RAM

For j in P[I:0xFF]
    mem[j] := SCRATCHPAD

Dynamic access patterns
Configuration within user bounds
Data collection relies on an automated framework
Tests are driven by the user-provided configuration
  - Selected performance counters, number of observations, etc.
Focus on reproducibility of results and experiments
1. Application configuration and synthetic tasks are generated from a template - This includes the list of tasks and resources in the system
2. The application build tree is generated from the configuration - Insert instrumentation around analysed task
3. Application and contenders are compiled into a single binary
4. Observable events set are generated based on the required ones
   - Each event is counted by a specific register
   - Some combinations of events cannot be captured on the TC27x
5. The board runs the application for each event set, collecting runtime traces
6. Traces are merged by matching corresponding runs of a task
   - Noise between observations verified to be random and negligible
• Evaluated on different applications:
  - TACLeBench benchmarks
  - Automotive case study from the CONCERTO Project http://www.concerto-project.org/
  - Familiarization case study with DENSO

• Evaluated on different software platforms:
  - Erika Enterprise Real-Time operating system
  - Sysgo PikeOS
  - Real-Time Linux (PREEMPT_RT)
  - Barebone

• Evaluated on different hardware:
  - TC27x
  - Freescale P4080
  - Raspberry Pi3
  - Cobham Gaisler Leon3 Multicore
Evaluation - Factor selection

Aurix Tricore TC27x Running on Core 0

**bitcount**

**Benchmark facts**
- Has multiple loops
- Uses little data

**Selected PMC**
- PMEM_STALL
- DMEM_STALL
- IP_DISPATCHSTALL
- LS_DISPATCHSTALL
- LP_DISPATCHSTALL
- MULTI_ISSUE
- PCACHE_MISS
- DCACHE_HIT
- DCACHE_MISS_CLEAN
- DCACHE_MISS_DIRTY
- TOTAL_BRANCH

**Benchmark analysis**
- Max/Min runtime: \( \times 1.21 \)
- Variability resulting from PMC:
  - Data Memory Stalls
  - Stalls in the Arithmetic Unit
  - Stalls in the Load/Store unit
  - Executed branches
- Results suggest that bitcount is:
  - Control flow dependent
  - Independent on shared resources
Evaluation - Factor selection

Aurix Tricore TC27x
Running on Core 0

**bitcount**

Simple kernel of bit counting functions.
- Has multiple loops
- Uses little data

- Max/Min runtime: $\times 1.21$
- Variability resulting from PMC:
  - Data Memory Stalls
  - Stalls in the Arithmetic Unit
  - Stalls in the Load/Store unit
  - Executed branches

- Results suggest that bitcount is:
  - Control flow dependent
  - Independent on shared resources
## Evaluation - Factor selection

**Aurix Tricore TC27x**

*Running on Core 0*

**matmult**

Matrix multiplication kernel
- Fetches input in RAM
- Stores output in RAM
- Follows a single path

<table>
<thead>
<tr>
<th>PMEM_STALL</th>
<th>DMEM_STALL</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP_DISPATCHSTALL</td>
<td>LS_DISPATCHSTALL</td>
</tr>
<tr>
<td>LP_DISPATCHSTALL</td>
<td>PCACHE_HIT</td>
</tr>
<tr>
<td>MULTI_ISSUE</td>
<td>PCACHE_MISS</td>
</tr>
<tr>
<td>PCACHE_HIT</td>
<td>DCACHE_HIT</td>
</tr>
<tr>
<td>DCACHE_MISS_CLEAN</td>
<td>DCACHE_MISS_DIRTY</td>
</tr>
<tr>
<td>TOTAL_BRANCH</td>
<td></td>
</tr>
</tbody>
</table>

- Max/Min runtime: $\times 1.32$
- Variability resulting from PMC:
  - Stalls in data memory on Core 0
  - Stalls in data memory on Core 1
  - Stalls in data memory on Core 2

- Result suggests that matmult is:
  - Data-dependent
  - Sensitive to the behaviour of other cores
Evaluation - Factor selection

Aurix Tricore TC27x
Running on Core 0

dijkstra

Path search in a graph
- Special case for empty paths
- Highly variable runtime
- Fetches input in RAM

- Max/Min runtime: $\times 1577$
- Variability resulting from PMC:
  - Stalls in the Arithmetic unit
  - The number of executed branches
  - Stalls in the Load/Store unit on Core 1
  - Stalls in the Load/Store unit on Core 2

- Result suggests that dijkstra is:
  - Control flow dependent
  - Sensitive to the behaviour of other cores
Evaluation - Interferences

Normalised execution time for **matmult**
Evaluation - Interferences

Normalised execution time for \textit{matmult}

Increased interferences
Evaluation-Interferences

Normalised execution time for `matmul`

Increased runtime
Normalised execution time for **matmult**

Worst-case configurations do not stem from maximised interferences.
Evaluation - Interferences

Both cores do not have the same impact

Normalised execution time for **matmult**
Evaluation Modelling

Interference modelling results on *dijsktra*

- Early work on modelling the impact of interferences on a task
- Forecasting-Based Interference analysis (FBI)
  - Use the selected factors as input to a multi-variate model
  - Interferences modelled as a multiplicative factor on the execution time
Conclusion

- Introduced a framework for the evaluation of the impact of interferences
  - Identify the interference channels relevant to a task
  - Automate the gathering of evidence to support timing arguments

- Evaluated on numerous configurations
  - [Ongoing] Real-Time Linux on Raspberry Pi3
  - Familiarization case study with Denso
  - Sysgo PikeOS on Freescale P4080
  - ...

- A first step towards tackling inter-core interferences
  - Feed the results into further tools, e.g. FBI analysis
  - Assess the robustness of a platform
Conclusion

- A wide exploration of the interference space is required
  - Rely on synthetic tasks to generate controlled contention
  - Resource stressing may not lead to worst-case configurations

- Observability should be supported at the platform level
  - Rely on existing performance monitoring infrastructure
  - Capture a broad view of the system behaviour

- Exercise a variety of interference channels
  - Rely on platform specific knowledge, refined through experimentation
  - Challenge assumptions to increase confidence in the observations
Thank you for your attention
PCA

Principal Components Analysis exhibits the underlying structure in a dataset.
Principal Components Analysis exhibits the underlying structure in a dataset. Both axes exhibit a similar spread of values.
PCA
Principal Components Analysis exhibits the underlying structure in a dataset

A change of axes can cover a larger range of variance
PCA
Principal Components Analysis exhibits the underlying structure in a dataset.

A change of axes can cover a larger range of variance.
PCA
Principal Components Analysis exhibits the underlying structure in a dataset

A change of axes can cover a larger range of variance
PCA
Principal Components Analysis exhibits the underlying structure in a dataset

A change of axes can cover a larger range of variance
Principal Components Analysis (PCA) exhibits the underlying structure in a dataset. Principal Components (PC) capture the main axes of variance:
- Reframe the dataset according to new dimensions
- Correlated factors are part of the same PC
1. Identify the Principal Components (PC) in the dataset
   - Apply PCA to extract the main axes of variance
2. Discard PC not correlated to the analysed task’s execution time
   - Focus on capturing timing variability
3. Discard PC with low contribution to overall variance
   - Remove low impact factors
4. Bound the number of PMC selected per component
   - Compute relative weights of remaining PC
   - Pick more PMC from high variance PC
5. Select best PMC set
   - Maximise the weighted loadings of selected PMC
   - Solved through Integer Linear Programming
   - Include user or platform constraints
TC27x

- Three asymmetric cores
  - Feature local scratchpads
  - Interface for debug
- Delays due to arbitration on Cross Bar slaves
- Focus on RAM accesses
  - Round robin arbitration
  - Code in scratchpad
  - Data mapped to RAM
Software platform

- Erika Enterprise Real-Time operating system
  - Support for multicore platforms
  - OSEK/VDX Compliant
  - Open Source and Free of charge
- Ad Hoc Instrumentation routines
  - Capture timing and PMC values on call
    - Request propagated to all cores
  - Data stored in local scratchpad buffer
  - Interrupt on full buffer to trigger data collection on host
- Instrumentation of code through Rapita Verification Suite
  - Task level, end-to-end observations
Contender()

0x00: while not configure:
0x01: access mem[01]
0x02: access mem[02]
0x03: access mem[03]
0x04: access mem[04]
<...>

0xFD: access mem[FD]
0xFE: access mem[FE]
0xFF: access mem[FF]

reconfigure()

P := permutation([0x01:0xFF])
I := rand([min_inter:max_inter])
For j in [0x01:I]
    mem[j] := RAM
For j in [I:0xFF]
    mem[j] := SCRATCHPAD

- Contenders loop over a sequence of memory accesses
- Contenders are mapped into core local scratchpads
  - No interference from instruction fetch
  - Require some allocated space in the memory map
- Accesses target either local, or uncacheable memory segments
  - Control which ones generate conflicts
  - Dynamically modified code to alter access patterns
- Configurations generated within user-defined bounds