Challenges for Worst-case Execution Time Analysis of Multi-core Architectures

Jan Reineke

IRIT, Toulouse

September 18, 2014
The Context: Hard Real-Time Systems

Safety-critical applications:
- Avionics, automotive, train industries, manufacturing control
- Side airbag in car
  Reaction in < 10 msec
- Crankshaft-synchronous tasks
  Reaction in < 45 microsec

- Embedded controllers must finish their tasks within given time bounds.
- Developers would like to know the **Worst-Case Execution Time (WCET)** to give a guarantee.
The Timing Analysis Problem

// Perform the convolution.
for (int i=0; i<10; i++) {
    x[i] = a[i]*b[j-i];
    // Notify listeners.
    notify(x[i]);
}

**Embedded Software** + \( ? \) = **Timing Requirements**

*Microarchitecture*
What does the execution time depend on?

- The **input**, determining which path is taken through the program.
- The **state of the hardware platform**: Due to caches, pipelining, speculation, etc.
- **Interference from the environment**: External interference as seen from the analyzed task on shared busses, caches, memory.
What does the execution time depend on?

- The **input**, determining which path is taken through the program.
- The **state of the hardware platform**:
  - Due to caches, pipelining, speculation, etc.
- **Interference from the environment**:
  - External interference as seen from the analyzed task on shared busses, caches, memory.

![Diagram showing CPU, memory, and cache connections]

Complex CPU (out-of-order execution, branch prediction, etc.)  

L1 Cache  

Main Memory
What does the execution time depend on?

- The **input**, determining which path is taken through the program.
- The **state of the hardware platform**:
  - Due to caches, pipelining, speculation, etc.
- **Interference from the environment**:
  - External interference as seen from the analyzed task on shared busses, caches, memory.

![Diagram of a computer system with complex CPUs, caches, and main memory.]
Example of Influence of Microarchitectural State

\[ x = a + b; \]
LOAD r2, _a  
LOAD r1, _b  
ADD r3, r2, r1

PowerPC 755

Execution Time (Clock Cycles)

- Best Case
- Worst Case

Courtesy of Reinhard Wilhelm.
Example of Influence of Corunning Tasks in Multicores

Radojkovic et al. (ACM TACO, 2012) on Intel Atom and Intel Core 2 Quad:

up to 14x slow-down due to interference on shared L2 cache and memory controller
Challenges

1. Modeling
   How to construct sound timing models?

2. Analysis
   How to precisely & efficiently bound the WCET?

3. Design
   How to design microarchitectures that enable precise & efficient WCET analysis?
The Modeling Challenge

Timing model = Formal specification of microarchitecture’s timing

Incorrect timing model → possibly incorrect WCET bound.
Current Process of Deriving Timing Model

Micro-architecture ➔ ? ➔ Timing Model
Current Process of Deriving Timing Model

Micro-architecture → ? → Timing Model

PREcision-Timed processors: Performance & Predictability

Jan Reineke, Saarland 12
Current Process of Deriving Timing Model

Micro-architecture

Timing Model
Current Process of Deriving Timing Model

Micro-architecture → ? → Timing Model
Current Process of Deriving Timing Model

→ Time-consuming, and
→ error-prone.
Current Process of Deriving Timing Model

→ Time-consuming, and
→ error-prone.
1. Future Process of Deriving Timing Model

- Micro-architecture
- VHDL Model
- Timing Model

Our Vision: PRET Machines

PREcision-Timed processors: Performance & Predicability

(Image: John Harrison’s H4, first clock to solve longitude problem)
1. Future Process of Deriving Timing Model

Derive timing model automatically from formal specification of microarchitecture.

→ Less manual effort, thus less time-consuming, and
→ provably correct.
1. Future Process of Deriving Timing Model

Derive timing model automatically from formal specification of microarchitecture.

- Less manual effort, thus less time-consuming, and
- provably correct.
1. Future Process of Deriving Timing Model

Derive timing model automatically from formal specification of microarchitecture.

- Less manual effort, thus less time-consuming, and
- provably correct.
2. Future Process of Deriving Timing Model

- Micro-architecture
- Perform measurements on hardware
- Infer model
- Timing Model
2. Future Process of Deriving Timing Model

Derive timing model automatically from measurements on the hardware using ideas from *automata learning*.

→ No manual effort, and
→ (under certain assumptions) provably correct.
→ Also useful to validate assumptions about microarch.
Proof-of-concept: Automatic Modeling of the Cache Hierarchy

- Cache Model is important part of Timing Model
- Can be characterized by a few parameters:
  - ABC: associativity, block size, capacity
  - Replacement policy

**chi** [Abel and Reineke, RTAS 2013] derives all of these parameters **fully automatically**.
Non-blocking caches:
Out-of-order execution:
Prefetching:
Other optimizations like way-prediction:

To minimize the effects of non-blocking caches and out-of-order execution, we can serialize memory accesses by using a form of “pointer chasing” where each memory location contains the address of the next access (for more details on this see section x).

Figure 4.2 shows the result of this modification. Now, a slight jump in the diagram for the time based approach is visible. But it is hard to find the exact location of the jump. Moreover, there is also already a small jump between 31kB and 32kB in the left diagram.

The following algorithm shows our first approach at inferring the associativity.

The algorithm uses the fact that the cache size is always a multiple of the way size. Thus, when accessing the memory with a stride of cache size many bytes, all accesses map to the same cache set. If $\text{curAssoc}$ exceeds the actual associativity, the cache can no longer store all accessed memory locations, and so we expect to see a jump in the number of misses.

Figures 4.3 and 4.4 show the result of running this algorithm on the same architecture as above, both with and without pointer chasing.
Example: Intel Core 2 Duo E6750, L1 Data Cache

Figure 4.2: Result of running the simple algorithm with pointer chasing on an Intel Core 2 Duo E6750 (32kB L1 Cache)

- Non-blocking caches:
- Out-of-order execution:
- Prefetching:
- Other optimizations like way-prediction:

To minimize the effects of non-blocking caches and out-of-order execution, we can serialize memory accesses by using a form of “pointer chasing” where each memory location contains the address of the next access (for more details on this see section x).

Figure 4.2 shows the result of this modification. Now, a slight jump in the diagram for the time based approach is visible. But it is hard to find the exact location of the jump. Moreover, there is also already a small jump between 31kB and 32kB in the left diagram.

The following algorithm shows our first approach at inferring the associativity.

The algorithm uses the fact that the cache size is always a multiple of the way size. Thus, when accessing the memory with a stride of cache size many bytes, all accesses map to the same cache set. If \( \text{curAssoc} \) exceeds the actual associativity, the cache can no longer store all accessed memory locations, and so we expect to see a jump in the number of misses.

Figures 4.3 and 4.4 show the result of running this algorithm on the same architecture as above, both with and without pointer chasing.
Example: Intel Core 2 Duo E6750, L1 Data Cache

Way Size = 4 KB

|Misses|

|Size|

Capacity = 32 KB

To minimize the effects of non-blocking caches and out-of-order execution, we can serialize memory accesses by using a form of "pointer chasing" where each memory location contains the address of the next access (for more details on this see section x).

Figure 4.2 shows the result of this modification. Now, a slight jump in the diagram for the time based approach is visible. But it is hard to find the exact location of the jump. Moreover, there is also already a small jump between 31kB and 32kB in the left diagram.

Figure 4.3 and Figure 4.4 show the result of running this algorithm on the same architecture as above, both with and without pointer chasing.
Replacement Policy

Approach inspired by methods to learn finite automata. Heavily specialized to problem domain.
Replacement Policy

Approach inspired by methods to learn finite automata. Heavily specialized to problem domain.

Discovered to our knowledge undocumented policy of the Intel Atom D525:

Modeling Challenge: Future Work

Extend automation to other parts of the microarchitecture:
- Translation lookaside buffers, branch predictors
- Shared caches in multicores including their coherency protocols
- Out-of-order pipelines?
The Analysis Challenge

Consider all possible program inputs

Consider all possible initial states of the hardware

\[ WCET_H(P) := \max_{i \in \text{Inputs}} \max_{h \in \text{States}(H)} ET_H(P, i, h) \]
The Analysis Challenge

Consider all possible program inputs

Consider all possible initial states of the hardware

\[ WCET_H(P) := \max_{i \in \text{Inputs}} \max_{h \in \text{States}(H)} ET_H(P, i, h) \]

Explicitly evaluating ET for all inputs and all hardware states is not feasible in practice:
- There are simply too many.
- Need for abstraction and thus approximation!
### The Analysis Challenge: State of the Art

<table>
<thead>
<tr>
<th>Component</th>
<th>Analysis Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caches, Branch Target Buffers</td>
<td>Precise &amp; efficient abstractions, for • LRU [Ferdinand, 1999] Not-so-precise but efficient abstractions, for • FIFO, PLRU, MRU [Grund and Reineke, 2008-2011] Reasonably precise quantitative analyses, for • FIFO, MRU [Guan et al., 2012-2014]</td>
</tr>
<tr>
<td>Complex Pipelines</td>
<td>Precise but very inefficient; little abstraction Major challenge: timing anomalies</td>
</tr>
<tr>
<td>Shared resources, e.g. busses, shared caches, DRAM</td>
<td>No realistic approaches yet Major challenge: interference between hardware threads → execution time depends on corunning tasks → need timing compositionality or isolation</td>
</tr>
</tbody>
</table>
The Design Challenge

**Wanted:**
Multi-/many-core architecture that is timing predictable and delivers high performance.

\[
\text{timing predictable} = \text{admits precise and efficient WCET analysis}
\]

and delivers high performance.
Approaches to Increase Predictability

Inputs = Program inputs
+ Initial state of microarchitecture
+ Tasks on other cores

1. Reduce uncertainty
2. Reduce influence of uncertainty on executions
3. Decouple analysis efficiency from number of executions
1. Reduce Uncertainty

a) Eliminate performance-enhancing features:
   E.g. branch predictor, cache, out-of-order execution…

If done naively: Reverses many micro-architectural developments…
   → Decreases performance…

**Key question**: How to reduce uncertainty without sacrificing performance?
1. Reduce Uncertainty

b) Replace features by alternatives with smaller state spaces:

- Caches $\rightarrow$ Scratchpads,
- Complex Pipeline $\rightarrow$ Simple VLIW or Thread-Interleaved Pipeline
- DRAM Controller w/ Closed-Page Policy
1. Reduce Uncertainty

c) Establish fixed microarchitectural state “once in a while”:

- By microarchitecture: partially flush pipeline at basic-block boundaries
  [Rochange and Sainrat, Computing Frontiers 2005]
- By `sync` instruction, placed by compiler
  [Maksoud and Reineke, RTNS 2014]
Approaches to Increase Predictability

Inputs = Program inputs
+ Initial state of microarchitecture
+ Tasks on other cores

1. Reduce uncertainty
2. Reduce influence of uncertainty on executions
3. Decouple analysis efficiency from number of executions
2) Reducing Influence of Uncertainty on Execution Times

If a particular source of uncertainty has no influence on the execution time of a program under analysis, it is irrelevant for timing analysis.

Principal example: Temporal Isolation
Temporal Isolation

- Temporal isolation between cores = timing of program on one core is independent of activity on other cores
- Formally:
  \[ T(P_1, \langle p_1, c_1, p_2, c_2 \rangle) = T_{isolated}(P_1, \langle p_1, c_2 \rangle) \]

- Can be exploited in WCET analysis:
  \[ WCET(P_1) = \max_{p_1, c_1, p_2, c_2} T(P_1, \langle p_1, c_1, p_2, c_2 \rangle) = \max_{p_1, c_1} T_{isolated}(P_1, \langle p_1, c_1 \rangle) \]
Temporal Isolation
How to achieve it?

Partition resources in **space** and/or **time**
- Resource appears like a slower and/or smaller private resource to each client

Examples:
- Time-division multiple access (TDMA) arbitration in shared busses
- Partitioned shared caches
- PRET DRAM Controller = time division of data/control bus + space division of DRAM internals

Why not simply provide **private** resources then?
Uncertainty about Inputs → Possible Executions → Analysis Efficiency

1. Reduce uncertainty
2. Reduce influence of uncertainty on executions
3. Decouple analysis efficiency from number of executions

Inputs = Program inputs + Initial state of microarchitecture + Tasks on other cores
3) Decouple analysis efficiency from number of executions

1. **Abstraction**: represent many concrete executions by few abstract ones:
   - Genius is in choice of abstraction
   - Usually requires some sort of monotonicity

2. **Disregard most executions**:
   - E.g. follow only **local worst-case**
   - Requires absence of **timing anomalies/domino effects**

3. **Divide and conquer**:
   - Analyze different components separately
   - Requires **timing compositionality**
Timing Anomalies

Timing Anomaly = Counterintuitive scenario in which the “local worst case” does not imply the “global worst case”.

Example: Scheduling Anomaly

Bounds on multiprocessing timing anomalies
(http://epubs.siam.org/doi/abs/10.1137/0117039) Jan Reineke, Saarland 44
Timing Anomalies

Example: Speculation Anomaly

- **Cache Hit**
  - A
  - Prefetch B - Miss
  - C

- **Cache Miss**
  - A
  - C

Prefetching as branch condition has not been evaluated yet

Memory access may induce additional cache misses later on

No prefetching as branch condition has already been evaluated yet
Timing Anomalies
Example: Cache Timing Anomaly of FIFO

Access: b c b d c

Similar examples exist for PLRU and MRU. Impossible for LRU.
Timing Anomalies
Consequences for Timing Analysis

In the presence of timing anomalies, a timing analysis cannot make decisions “locally”: it needs to consider all cases.

→ May yield “State explosion problem”
Timing Anomalies
Open Analysis and Design Challenges

- How to determine whether a given timing model exhibits timing anomalies?
- How to construct processors without timing anomalies?
  - Caches: LRU replacement
  - No speculation
  - Other aspects: “halt” everything upon every “timing accident” \( \rightarrow \) possibly very inefficient
- How to construct conservative timing model without timing anomalies?
  - Can we e.g. add a “safety margin” to the local worst case?
Domino Effects

- Intuitively:
  domino effect = “unbounded” timing anomaly

- Examples:
  - Pipeline (e.g. PowerPC 755)
  - Caches (FIFO, PLRU, MRU, …)
Domino Effects
Example: Cache Domino Effect of FIFO

Access: b  c  b  d  c

Similar examples exist for PLRU and MRU. Impossible for LRU.
Domino Effects
Open Analysis and Design Challenges

Exactly as with timing anomalies:
- How to determine whether a given timing model exhibits domino effects?
- How to construct processors without domino effects?
- How to construct conservative timing model without domino effects?
Some timing accidents are hard or even impossible to statically exclude at any particular program point:

- Interference on a shared bus: depends on behavior of tasks executed on other cores
- Interference on a cache in preemptively scheduled systems
- DRAM refreshes

But it may be possible to make cumulative statements about the number of these accidents.
Timing Compositionality
Intuitive Meaning

- Timing of a program can be decomposed into contributions by different “components”, e.g.
  - Pipeline
  - Cache non-preempted
  - Cache-related preemption delay
  - Bus interference
  - DRAM refreshes
  - ...

- Example, decomposition into pipeline and cache

\[ T_{\text{pipeline, cache}}(P, \langle p, c \rangle) = T_{\text{pipeline}}(P, \langle p \rangle) \oplus T_{\text{cache}}(P, \langle c \rangle) \]

Monotonic
Timing Compositionality
Application in Timing Analysis

Then, the components (here: pipeline and cache) can also be analyzed separately:

\[
WCET_{\text{pipeline, cache}}(P) = \max_{p,c} T_{\text{pipeline, cache}}(P, \langle p, c \rangle)
\leq \max_{p} T_{\text{pipeline}}(P, \langle p \rangle) \oplus \max_{c} T_{\text{cache}}(P, \langle c \rangle)
= WCET_{\text{pipeline}}(P) + WCET_{\text{cache}}(P)
\]

Hahn, Reineke, Wilhelm: 
*Towards Compositionality in Execution Time Analysis - Definition and Challenges.*
In CRTS 2013.
In preemptive scheduling, preempts may "disturb" the cache contents of preempted tasks:

Additional misses due to preemption, referred to as the Cache-Related Preemption Delay (CRPD).
Timing Compositionality
Example: “Cache-aware” Response-Time Analysis

Timing decomposition:
- WCET of T1 without preemptions: C1
- WCET of T2 without preemptions: C2
- Additional cost of T1 preempting T2:
  \[ \text{CRPD}_{1,2} = \text{BRT} \times \text{#additional misses} \]
→ Response time of T2:
  \[ R_2 \leq C_2 + \text{#preemptions} \times (C_1 + \text{CRPD}_{1,2}) \]
Timing Compositionality
Open Analysis and Design Challenges

- How to check whether a given decomposition of a timing model is valid?
- How to compute bounds on the cost of individual events, such as cache misses (BRT in previous example) or bus stalls?
- How to build microarchitecture in a way that permits a sound and precise decomposition of its timing?
  - Beyond simply stalling the whole system
- Evaluate costs and benefits of compositional analyses.
Emerging Challenge: Microarchitecture Selection & Configuration

// Perform the convolution.
for (int i=0; i<10; i++) {
  x[i] = a[i]*b[j-i];
  // Notify listeners.
  notify(x[i]);
}

Embedded Software with Timing Requirements

Family of Microarchitectures = Platform
Emerging Challenge: Microarchitecture Selection & Configuration

Family of Microarchitectures = Platform

Choices:
- Processor frequency
- Sizes and latencies of local memories
- Latency and bandwidth of interconnect
- Presence of floating-point unit
- ...

Timing Requirements

Embedded Software

with

```c
// Perform the convolution.
for (int i=0; i<10; i++) {
    x[i] = a[i]*b[j-i];
    // Notify listeners.
    notify(x[i]);
}
```
Emerging Challenge: Microarchitecture Selection & Configuration

Select a microarchitecture that
a) satisfies all timing requirements, and
b) minimizes cost/size/energy.

Summary: Approaches to Increase Predictability

Inputs = Program inputs
+ Initial state of microarchitecture
+ Tasks on other cores

1. Reduce uncertainty by simplifying microarchitecture, e.g. eliminate cache, branch prediction, etc.

2. Reduce influence of uncertainty on executions, e.g. by temporal isolation

3. Decouple analysis efficiency from number of executions:
   • Eliminate timing anomalies and domino effects,
   • Achieve timing compositionality e.g. by LRU replacement, stalling pipeline upon cache miss
Conclusions

Challenges in modeling, analysis, design remain.
Conclusions

Challenges in modeling, analysis, design remain. Progress based on machine learning, abstraction, partitioning has been made.

Thank you for your attention!
Some References

A Compiler Optimization to Increase the Efficiency of WCET Analysis

Architecture-Parametric Timing Analysis

Selfish-LRU: Preemption-Aware Caching for Predictability and Performance

Towards Compositionality in Execution Time Analysis - Definition and Challenges


Measurement-based Modeling of the Cache Replacement Policy

A PRET Microarchitecture Implementation with Repeatable Timing and Competitive Performance

PRET DRAM Controller: Bank Privatization for Predictability and Temporal Isolation