Timing Predictability and How to Achieve It

Jan Reineke

Dagstuhl Seminar
“Adaptive Isolation for Predictability and Security”

October 31, 2016
The Timing Analysis Problem

Set of Software Tasks + Microarchitecture = Timing Requirements

```cpp
// Perform the convolution.
for (int i=0; i<10; i++) {
    x[i] = a[i]*b[j-i];
    // Notify listeners.
    notify(x[i]);
}
```
What does the execution time depend on?

- The **input**, determining which path is taken through the program.

![Diagram showing the relationship between Simple CPU and Memory](image-url)
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.

![Diagram showing complex CPU, L1 Cache, and Main Memory connections.]
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)
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.
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
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.
The Need for Models

Predictions about the future behavior of a system are always based on models of the system.

All models are wrong, but some are useful.

George Box (Statistiker)
The Need for Timing Models

The ISA partially defines the behavior of microarchitectures: it abstracts from timing.

How to obtain timing models?
- Hardware manuals
- Manually devised microbenchmarks
- Machine learning

Challenge: Introduce HW/SW contract to capture timing behavior of microarchitectures.
Desirable Properties of Systems and their Timing Models

- Predictability
- Analyzability
Predictability

How precisely can programs’ execution times on a particular microarchitecture be predicted?

Assuming a deterministic timing model and known initial conditions, can perfectly predict execution time.

But: initial state, inputs, and interference unknown.
Timing Predictability

- Possible execution times
- Overestimation
- Analysis
- Guaranteed upper bound

Frequency vs. execution time
- BCET
- WCET
How to Increase Predictability?

1. *Eliminate stateful components:*
   
   Cache → Scratchpad memory
   Regular Pipeline → Thread-interleaved pipeline
   Out-of-order execution → VLIW

**Challenge:** Efficient static allocation of resources.
How to Increase Predictability?

2. *Eliminate interference*: „*temporal isolation*“
   Partition resources:
   - **in time**
     - TDMA bus/NoC arbitration,
     - SW scheduling (e.g. PREM)
   - **in space**
     - shared cache: in HW or SW
     - SRAM banks (e.g. Kalray MPPA)
     - DRAM banks (e.g. PRET DRAM, PALLOC)

**Challenge:**
Determine efficient partitioning of resources.

**Question:** What’s the performance impact?
How to Increase Predictability?

3. Choose “forgetful“/„insensitive“ components:
   FIFO, PLRU replacement → LRU replacement
   [Real-Time Systems 2007, WAOA 2015]

Open Problems:

- Is there a systematic way to design „forgetful“ microarchitectural components?
- Can randomization help?
Analyzability

How *efficiently* can programs‘ WCETs on a particular microarchitecture be bounded?

WCET analysis needs to consider all inputs, initial HW states, interference scenarios...

...explicitly or implicitly.
How to Increase Analyzability?

1. Eliminate stateful resources:
   Fewer states to consider

2. Eliminate interference: „temporal isolation“:
   Can focus analysis on one partition

3. Choose „forgetful“/“insensitive“ components:
   Different analysis states will quickly converge

4. Enable efficient *implicit* treatment of states:
   - Monotonicity / Freedom from Timing Anomalies
   - Timing Compositionality
Timing Anomalies

- Cache Miss = Local Worst Case
- Cache Hit

Nondeterminism due to uncertainty about hardware state

Timing Anomalies in Dynamically Scheduled Microprocessors
T. Lundqvist, P. Stenström – RTSS 1999
Timing Anomalies: Example

Scheduling Anomaly

Bounds on multiprocessing timing anomalies
(http://epubs.siam.org/doi/abs/10.1137/0117039)
Timing Compositionality: By Example

\[ \text{Timing Compositionality} = \]

Ability to simply sum up timing contributions by different components

Implicitly or explicitly assumed by (almost) all approaches to timing analysis for multi cores and cache-related preemption delays (CRPD).
Timing Compositionality: Benefit

How does compositionality help?

Efficiency of microarchitectural analysis
Conventional Wisdom

Simple in-order pipeline + LRU caches
→ no timing anomalies
→ timing-compositional
Bad News I: Timing Anomalies

We show such a pipeline has timing anomalies:

*{Toward Compact Abstractions for Processor Pipelines*}

A Timing Anomaly

Hit case:
- Instruction fetch starts before second load becomes ready
- Stalls second load, which misses the cache

Miss case:
- Second load can catch up during first load missing the cache
- Second load is prioritized over instruction fetch
- Loading before fetching suits subsequent execution

**Intuitive Reason:**
Progress in the pipeline influences order of instruction fetch and data access
Bad News II: Timing Compositionality

Maximal cost of an additional cache miss?

**Intuitively**: main memory latency

**Unfortunately**: ~ 2 times main memory latency
- ongoing instruction fetch may block load
- ongoing load may block instruction fetch
Good News

Two approaches to solve problem:
1. Stall entire processor upon „timing accidents“
2. Strictly in-order pipeline
Strictly In-Order Pipelines: Definition

**Definition (Strictly In-Order):**
We call a pipeline *strictly in-order* if each resource processes the instructions in program order.

- Enforce memory operations (instructions and data) in-order (common memory as resource)
- Block instruction fetch until no potential data accesses in the pipeline
Theorem 1 (Monotonicity):
In the strictly in-order pipeline progress of an instruction is monotone in the progress of other instructions.

In the blue state, each instruction has the same or more progress than in the red state.
Strictly In-Order Pipelines: Properties

Theorem 2 (Timing Anomalies):
The strictly in-order pipeline is free of timing anomalies.

By monotonicity
Strictly In-Order Pipelines: Properties

**Theorem 3 (Timing Compositionality):**
The strictly in-order pipeline admits „compositional analysis with intuitive penalties.“

**Open Question:** What’s the performance impact of being strictly in-order?
Conclusions

- Need faithful timing models
- Various approaches to achieve predictability
  - Achieving efficiency is hard
- How to convince industry to build and to buy predictable processors?

Thank you for your attention!
Some References

Enabling Compositionality for Multicore Timing Analysis

MIRROR: Symmetric Timing Analysis for Real-Time Tasks on Multicore Platforms with Shared Resources

A Generic and Compositional Framework for Multicore Response Time Analysis

On the Smoothness of Paging Algorithms

Toward Compact Abstractions for Processor Pipelines

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

PRET DRAM Controller: Bank Privatization for Predictability and Temporal Isolation

Timing Predictability of Cache Replacement Policies