SOUND AND EFFICIENT WCET ANALYSIS IN THE PRESENCE OF TIMING ANOMALIES[1]

Jan Reineke[2] and Rathijit Sen[3]

Abstract

Worst-Case-Execution-Time (WCET) analysis computes upper bounds on the execution time of a program on a given hardware platform. Abstractions employed for static timing analysis can lead to non-determinism that may require the analyzer to evaluate an exponential number of choices even for straight-line code. Pruning the search space is potentially unsafe because of "timing anomalies" where local worst-case choices may not lead to the global worst-case scenario. In this paper we present an approach towards more efficient WCET analysis that uses precomputed information to safely discard analysis states.

1. Introduction

Embedded systems as they occur in application domains such as automotive, aeronautics, and industrial automation often have to satisfy hard real-time constraints. Timeliness of reactions is absolutely necessary. Off-line guarantees on the worst-case execution time of each task have to be derived using safe methods.

Static worst-case execution time (WCET) tools employ abstract models of the underlying hardware platforms. Such models abstract away from parts of the hardware that do not influence the timing of instructions, like e.g. the values of registers. In addition to the abstraction of data, such models further abstract components like branch predictors and caches, that have a very large state space. Good abstractions of hardware components can predict their precise concrete behavior most of the time. However, in general, abstractions introduce non-determinism: whenever the abstract model cannot determine the value of a condition, it has to consider all possibilities, to cover any possible concrete behavior.

Considering all possibilities can make timing analysis very expensive. Even on straight-line code the timing analyzer might have to consider an exponential number of possibilities in the length of the path. Often, the different possibilities can be intuitively classified as local worst- or local best-cases: e.g. cache miss vs. cache hit, pipeline stall or not, branch misprediction or not, etc. In such cases, it is tempting to only follow the local worst-case if one is interested in the WCET. Unfortunately, due to *timing anomalies* [12, 15] this is not always sound. A timing anomaly is a situation where the local worst-case does not entail the global worst-case. For instance, a cache miss—the local worst-case—may result in a shorter execution time, than a cache hit, because of scheduling effects. See Figure 1 for an example. Shortening task A leads to a longer overall schedule, because task B can now block the "more" important task C, which may only run on Resource 2. Analogously, there are

[2]Universität des Saarlandes, Saarbrücken, Germany, reineke@cs.uni-saarland.de

[3]University of Wisconsin, Madison, USA, rathijit@cs.wisc.edu

[1]This work has profited from discussions within the ARTIST2 Network of Excellence. It is supported by the German Research Foundation (DFG) as part of SFB/TR AVACS and by the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement n° 216008 (Predator).

cases where shortening a task leads to an even greater decrease in the overall schedule. Reineke et al. [15] formally define timing anomalies and give a set of examples for timing anomalies involving caches and speculation.

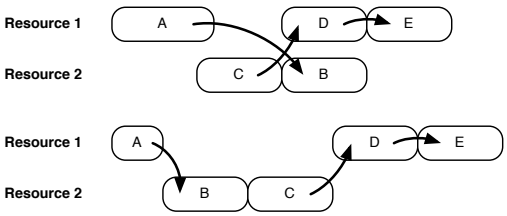

Figure 1. Scheduling Anomaly.

In the presence of timing anomalies it is unsound to simply discard non local-worst-cases. We assume that most if not all modern processors exhibit timing anomalies and that there is little hope to construct processors with good performance that do not exhibit them. To obtain an efficient timing analysis, we would still like to discard non-local-worst-case states, yet in a sound way. Our idea is to perform a precomputation on the abstract model of the hardware, that is used for timing analysis. The precomputation computes for each pair of analysis states a bound on the future difference in timing between the two states. This precomputation might be very expensive, but it only has to be done once for a particular abstract hardware model. Timing analyses using this hardware model can then safely discard analysis states based on the precomputed differences, even in the presence of timing anomalies. Assuming that timing anomalies occur seldomly and can be excluded using the precomputation most of the time, the proposed analysis will be much more efficient than previous exhaustive methods.

2. Related Work

Graham [7] shows that a greedy scheduler can produce a longer schedule, if provided with shorter tasks, fewer dependencies, more processors, etc. Graham also gives bounds on these effects, which are known as scheduling anomalies today.

Lundqvist & Stenström first introduced timing anomalies in the sense relevant for timing analysis. In [12] they give an example of a cache miss resulting in a shorter execution time than a cache hit. A timing anomaly is characterized as a situation where a positive (negative) change of the latency of the first instruction results in a global decrease (increase) of the execution time of a sequence of instructions. Situations where the local effect is even accelerated are also considered timing anomalies, i.e. the global increase (decrease) of the execution time is greater than the local change.

In his PhD thesis [3] and a paper with Jonsson [4], Engblom also discusses timing anomalies. He translates the notion of timing anomalies of the Lundqvist/Stenström paper [12] to his model by assuming that single pipeline stages take longer, in contrast to whole instructions. Both Lundqvist and Engblom claim that, in processors that contain in-order resources only, no timing anomalies can occur. This is not always true unfortunately, as corrected in Lundqvist's thesis [11]. Schneider [17] and Wenzel et al. [21] note that if there exist several resources that have overlapping, but unequal capabilities, timing anomalies can also occur.

Reineke et al. [15] provide the first formal definition of timing anomalies. The definition of timing anomalies in this paper is a slight relaxation of that of [15]. In recent work, Kirner et al. [9] introduce the notion of parallel timing anomalies. Such anomalies arise if a WCET analysis is split into the analysis of several hardware components that operate in parallel. Depending on how the analysis results are combined the resulting WCET estimate may be unsafe. The authors identify conditions that guarantee the safety of two combination methods. This work is orthogonal to ours.

3. Static WCET Analysis Framework

Over the last several years, a more or less standard architecture for static timing-analysis tools has emerged [8, 19, 5]. Figure 2 gives a general view on this architecture. One can distinguish three major building blocks:

1. Control-flow reconstruction and static analyses for control and data flow.
2. Micro-architectural analysis, which computes upper and lower bounds on execution times of basic blocks.
3. Global bound analysis, which computes upper and lower bounds for the whole program.

Micro-architectural analysis [6, 11, 3, 20, 10, 16] determines bounds on the execution time of basic blocks, taking into account the processor's pipeline, caches, and speculation concepts. Static cache analyses determine safe approximations to the contents of caches at each program point. Pipeline analysis analyzes how instructions pass through the pipeline accounting for occupancy of shared resources like queues, functional units, etc. Ignoring these average-case-enhancing features would result in very imprecise bounds.

In this paper, we consider micro-architectural analyses that are based on abstractions of the concrete hardware and that propagate abstract states through the control-flow-graph [3, 20, 6]. In such analyses, the micro-architectural analysis is the most expensive part of the WCET analysis. Due to the introduction of non-determinism by abstraction and the possibility of *timing anomalies* it is necessary to consider very many cases. For complex architectures, this may yield hundreds of millions of analysis states, and may thus be very memory and time intensive.

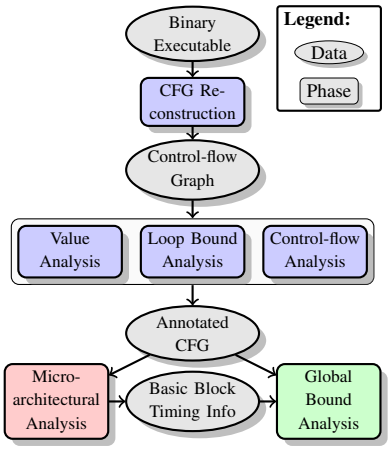

Figure 2. Main components of a timing-analysis framework and their interaction.

In the following, we will describe a simple, formal model of such a micro-architectural analysis. Later, we will describe how this model can be extended to enable more efficient analyses.

3.1. A formal model of micro-architectural analysis

Micro-architectural analysis determines bounds on the execution times of basic blocks. Using an abstract model of the hardware, it "simulates" the possible behavior of the hardware in each cycle. Due to uncertainty about the concrete state of the hardware or about its inputs, abstract models are—unlike usual cycle-accurate simulators—non-deterministic. Therefore, an abstract state may have several possible successor states.

Cycle semantics. Let $State$ be the set of states of the abstract hardware model, and let $Prog$ be the set of programs that can be executed on the hardware. Then the *cycle semantics* can be formally modeled by the following function:

$$cycle : State \times Prog \to \mathcal{P}(State).$$

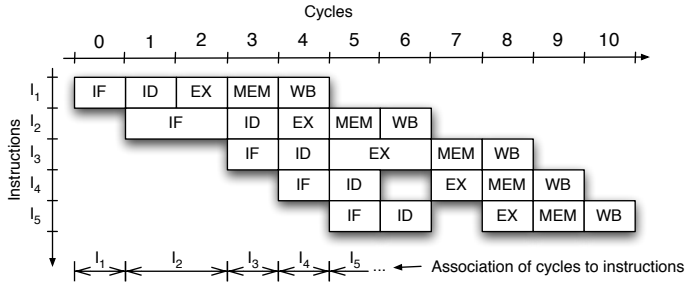

Figure 3. Pipelined execution of several instructions and association of instructions with execution cycles (bottom).

It takes an abstract state and a program and computes the set of possible successor states. Such an abstract hardware model can be proved correct by showing that it is an abstract interpretation [2] of the concrete hardware model. To bound the execution time of a basic block, one needs to associate execution cycles with instructions in the program. In pipelined processors several instructions are executed simultaneously, see Figure 3. There is no canonical association of execution cycles to instructions. One of several possibilities is to associate a cycle with the last instruction that was fetched. This is exemplified in the lower part of Figure 3.

Instruction semantics. Based on such a connection between execution cycles and instructions, one can lift the cycle semantics to an instruction semantics. This instruction semantics takes an abstract hardware state and an instruction that is to be fetched and computes the set of possible abstract hardware states until the next instruction can be fetched. Each of the resulting abstract hardware states is associated with the number of cycles to reach this state:

$$exec_I : State \times I \to \mathcal{P}(State \times \mathbb{N}).$$

For a formalization of the connection of the two semantics, see [20]. As an example of the two semantics in the analysis of the execution of a basic block, see Figure 4.

Instruction semantics can be easily lifted to the execution of basic blocks, which are simply sequences of instructions $\in I^*$:

$$exec_{BB} : State \times I^* \to \mathcal{P}(State \times \mathbb{N})$$

$$\begin{array}{rcl} exec_{BB}(s, \epsilon) & := & \{(s, 0)\} \\ exec_{BB}(s, \iota_0 \ldots \iota_n) & := & \{(s'', t' + t'') \mid \begin{array}{l} (s', t') \in exec_I(s, \iota_0) \wedge \\ (s'', t'') \in exec_{BB}(s', \iota_1 \ldots \iota_n)\} \end{array} \end{array}$$

As a shortcut, we will also write $s \xrightarrow[\iota]{t'} s'$ for $(s', t') \in exec_I(s, \iota)$ and similarly $s \xrightarrow[\iota_0 \ldots \iota_n]{t'} s'$ for $(s', t') \in exec_{BB}(s, \iota_0 \ldots \iota_n)$.

Bounds on the execution times of basic blocks. Given an instruction semantics on a finite set of states $State$, one can compute bounds on the execution times of the basic blocks of a program. To that end, one can compute the set of abstract states that might reach the start of each basic block through a fixed-point iteration. For each of the states that might reach a basic block, the instruction semantics can then be used to compute the maximum execution time of the block starting in that state:

$$\max(s, \iota_0 \ldots \iota_n) := \max\{t \mid s \xrightarrow[\iota_0 \ldots \iota_n]{t} s'\}. \tag{1}$$

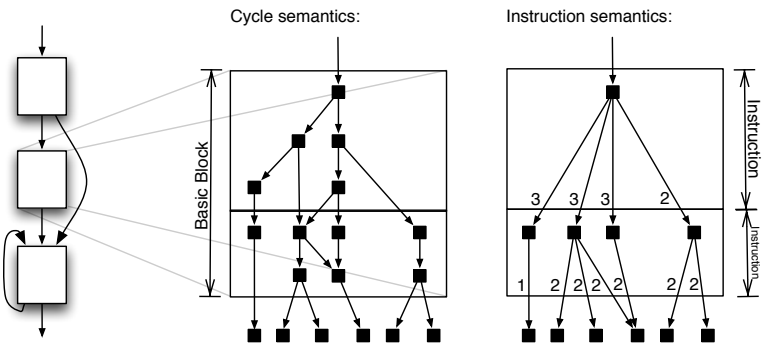

Figure 4. Cycle semantics and instruction semantics of a basic block.

Finally, the maximum of these times for each state reaching a basic block is an upper bound on the execution time of that basic block in the given program. Analogously to the computation of upper bounds one can also compute lower bounds on the execution time of a basic block by the $\min$ function:

$$\min(s, \iota_0 \dots \iota_n) := \min\{t \mid s \xrightarrow[\iota_0 \dots \iota_n]{t} s'\}. \tag{2}$$

4. Timing Anomalies, Domino Effects, and How to Safely Discard States

The approach described in the previous section can be very expensive. Due to non-determinism introduced by abstraction, the set of states to be considered can be extremely large. It is therefore tempting to discard states that are non local-worst-case, e.g., states resulting from a cache hit, if both a cache hit and a cache miss may happen.

4.1. Timing anomalies

Unfortunately, due to so-called *timing anomalies* [12] this is not always sound. The following is a definition of timing anomalies that is slightly relaxed compared with that of [15].

Definition 1 (Timing anomaly)**.** *An instruction semantics has a* timing anomaly *if there exists a sequence of instructions $\iota_0\iota_1 \dots \iota_n \in I^*$, and an abstract state $s \in State$, such that*

- *there are states $s_1, s_2 \in State$, with $s \xrightarrow[\iota_0]{t_1} s_1$ and $s \xrightarrow[\iota_0]{t_2} s_2$, and $t_1 < t_2$, such that*
- $t_1 + \max(s_1, \iota_1 \dots \iota_n) > t_2 + \max(s_2, \iota_1 \dots \iota_n)$.

In an instruction semantics with timing anomalies it is unsound to discard a non-local-worst-case state (like s_1 in the definition). The execution of non-local-worst-case states may be so much slower than the execution of local-worst-case states, that the non-local-worst-case state yields the global worst-case timing. Experience shows that reasonable abstract instruction semantics for most modern processors exhibit timing anomalies.

4.2. How to safely discard analysis states

Our approach is to precompute a function $\Delta : State \times State \to \mathbb{N}^\infty$, where $\mathbb{N}^\infty = \mathbb{N} \cup \{\infty\}$, that bounds the maximal difference in worst-case timing between the two states on any possible instruction sequence. Sometimes the difference in timing between two states cannot be bounded by any constant. That is why we augment $\mathbb{N}$ with ∞. The order of natural numbers is lifted to $\mathbb{N}^\infty$ by adding $\infty \geq n$ for all $n \in \mathbb{N}^\infty$.

Definition 2 (Valid Δ)**.** *A Δ function is* valid*, if for all pairs of states $s_1, s_2 \in State$ and for all instruction sequences $\iota_0 \dots \iota_n \in I^*$:*

$$\Delta(s_1, s_2) \geq \max(s_1, \iota_0 \dots \iota_n) - \max(s_2, \iota_0 \dots \iota_n).$$

Given such a Δ function it is possible to safely discard analysis states. If the analysis encounters two states s_1 and s_2 with execution times t_1 and t_2, respectively, it may discard s_2 if $t_1 - t_2 \geq \Delta(s_2, s_1)$ and s_1 if $t_2 - t_1 \geq \Delta(s_1, s_2)$. In these cases, the discarded state can never overtake the other state. So Δ can be used to locally exclude the occurrence of a timing anomaly. It is expected that this is often the case, as timing anomalies are not the common case. This optimization does not influence the precision of the analysis.

Even if $t_1 - t_2 < \Delta(s_2, s_1)$, one can safely discard s_2, by adding the penalty $\Delta(s_2, s_1) - (t_1 - t_2)$ to t_1. For the resulting $t_1' = t_1 + \Delta(s_2, s_1) - (t_1 - t_2)$, it holds that $t_1' - t_2 \geq \Delta(s_2, s_1)$. In contrast to the first optimization, this of course comes at the price of decreased precision. This optimization offers a way of trading precision for efficiency.

4.3. Domino effects

Unfortunately, the difference in timing between two states cannot always be bounded by a constant. This situation is known as a *domino effect*[1].

Definition 3 (Domino effect)**.** *An instruction semantics has a* domino effect *if there are two states $s_1, s_2 \in State$, such that for each $\Delta \in \mathbb{N}$ there is a sequence of instructions $\iota_0 \dots \iota_n \in I^*$, such that*

$$\max(s_1, \iota_0 \dots \iota_n) - \max(s_2, \iota_0 \dots \iota_n) \geq \Delta.$$

In other words, there are two states whose timing may arbitrarily diverge. Such effects are known to exist in pipelines [17] and caches [1, 14]. If such domino effects exist for many pairs of states, valid Δ functions will often have value ∞. In that case, the Δ function is rather useless; it can rarely be used to discard states.

Although the difference in execution times may not be bounded by a constant, the ratio between the two execution times is always bounded, i.e., $\frac{\max(s_1, \iota_0 \dots \iota_n)}{\max(s_2, \iota_0 \dots \iota_n)} < \rho$ for some constant ρ. An alternative to computing Δ functions is to compute two functions $\rho : State \times State \to \mathbb{Q}$ and $\delta : State \times State \to \mathbb{Q}$ that together bound the maximal difference between the two states in the following way:

Definition 4 (Valid ρ and δ functions)**.** *A pair of functions ρ and δ is* valid*, if for all pairs of states $s_1, s_2 \in State$ and instruction sequences $\iota_0 \dots \iota_n \in I^*$:*

$$\max(s_1, \iota_0 \dots \iota_n) \leq \rho(s_1, s_2) \cdot \max(s_2, \iota_0 \dots \iota_n) + \delta(s_1, s_2).$$

If there are no domino effects, there are valid ρ and δ functions in which ρ is 1 everywhere. In that case, δ is valid in the sense of Definition 2. Otherwise, ρ and δ can still be used to safely discard states: Say the analysis wants to discard a state s_1 with execution time t_1, but keep another state s_2 with execution time t_2, s.t. $\rho(s_1, s_2) = 1.05$ and $\delta(s_1, s_2) = 5$. Then, the analysis could discard s_1, but remember to multiply the future execution time of s_2 by 1.05. Also, similarly to the case of Δ functions, if $t_2 - t_1 < 5$, it would have to add $5 - (t_2 - t_1)$ to t_2.

[1] Domino effects are also known as *unbounded timing effects* [11].

5. Computation of Valid Δ Functions

Given an instruction semantics, how to compute a valid Δ function? Of course, one cannot simply enumerate all sequences of instructions. However, it is possible to define a system of recursive constraints whose solutions are valid Δ functions. These recursive equations correspond to the execution of no instruction at all or to the execution of a single instruction. Therefore, a finite number of constraints will suffice.

As Δ needs to bound the difference in timing for all instruction sequences, it must do so in particular for the empty sequence. This implies the constraints

$$\Delta(s_1, s_2) \;\geq\; 0 \tag{3}$$

for all $s_1, s_2 \in State$. Longer sequences can be covered by the recursive constraints

$$\Delta(s_1, s_2) \;\geq\; t_1' - t_2' + \Delta(s_1', s_2') \tag{4}$$

for all $s_1, s_2, s_1', s_2' \in State$ such that $s_1 \xrightarrow[\iota]{t_1'} s_1' \wedge s_2 \xrightarrow[\iota]{t_2'} s_2'$ for some ι.

Theorem 1. *Any solution Δ to this set of constraints is a valid Δ function.*

Proof. A Δ function that satisfies the above constraints actually fulfills a stronger condition than validity. The constraints guarantee that $\Delta(s_1, s_2) \geq \max(s_1, \iota_0 \ldots \iota_n) - \min(s_2, \iota_0 \ldots \iota_n)$ for all $s_1, s_2 \in State$ and $\iota_0 \ldots \iota_n \in I^*$. Proof by induction over the length of the sequence $\iota_0 \ldots \iota_n$:

Base case: We have to show that $\Delta(s_1, s_2) \geq \max(s_1, \epsilon) - \min(s_2, \epsilon)$. Since $\max(s_1, \epsilon) - \min(s_2, \epsilon) = 0$, this is trivially fulfilled by satisfaction of the $\Delta(s_1, s_2) \geq 0$ constraints.

Inductive step: We have to show that $\Delta(s_1, s_2) \geq \max(s_1, \iota_0 \ldots \iota_{n+1}) - \min(s_2, \iota_0 \ldots \iota_{n+1})$ given that $\Delta(s_1, s_2) \geq \max(s_1, \iota_1 \ldots \iota_{n+1}) - \min(s_2, \iota_1 \ldots \iota_{n+1})$. The recursive constraints guarantee that

$$\Delta(s_1, s_2) \geq \max\{t_1' - t_2' + \Delta(s_1', s_2') \mid s_1 \xrightarrow[\iota_0]{t_1'} s_1' \wedge s_2 \xrightarrow[\iota_0]{t_2'} s_2'\}.$$

Plugging the induction hypothesis (I.H.) for $\Delta(s_1', s_2')$ into this yields

$$\begin{aligned}
& \Delta(s_1, s_2) \\
\overset{\text{I.H.}}{\geq}\; & \max\{t_1' - t_2' + \max(s_1', \iota_1 \ldots \iota_{n+1}) - \min(s_2', \iota_1 \ldots \iota_{n+1}) \mid s_1 \xrightarrow[\iota_0]{t_1'} s_1' \wedge s_2 \xrightarrow[\iota_0]{t_2'} s_2'\} \\
\overset{\text{Eq. 1,2}}{=}\; & \max\{t_1' - t_2' + t_1'' - t_2'' \mid s_1 \xrightarrow[\iota_0]{t_1'} s_1' \wedge s_1' \xrightarrow[\iota_1 \ldots \iota_{n+1}]{t_1''} s_1'' \wedge\; s_2 \xrightarrow[\iota_0]{t_2'} s_2' \wedge s_2' \xrightarrow[\iota_1 \ldots \iota_{n+1}]{t_2''} s_2''\} \\
=\; & \max\{t_1' + t_1'' \mid s_1 \xrightarrow[\iota_0]{t_1'} s_1' \wedge s_1' \xrightarrow[\iota_1 \ldots \iota_{n+1}]{t_1''} s_1''\} - \min\{t_2' + t_2'' \mid s_2 \xrightarrow[\iota_0]{t_2'} s_2' \wedge s_2' \xrightarrow[\iota_1 \ldots \iota_{n+1}]{t_2''} s_2''\} \\
\overset{\text{Eq. 1,2}}{=}\; & \max(s_1, \iota_0 \ldots \iota_{n+1}) - \min(s_2, \iota_0 \ldots \iota_{n+1})
\end{aligned}$$

□

The lower the function value of Δ, the more states can be discarded using Δ. We are therefore computing the *least* solution to the set of constraints.

Our constraints fall into the class of *difference constraints*. The least solution of a system of difference constraints can be found by solving a shortest path problem [13]. To compute this efficiently, one can use Tarjan's algorithm of subtree disassembly and FIFO selection rule [18]. Negative cycles in the

constraint graph correspond to domino effects. Tarjan's algorithm allows for an efficient detection and elimination of these cycles. The least solution for pairs of states on these cycles is ∞.

We have also implemented a solution to compute valid ρ and δ functions, which allows to bound the effect of domino effects. However, due to space constraints, we cannot present the theoretical background and the results of these computations here.

6. Case Study

We used the analysis technique described above on a processor model that includes a decoupled fetch pipeline, multi-cycle functional units and variable memory access latencies. Figure 5 shows a diagrammatic representation of this model. `E0` through `Ek` represent functional units that execute instructions. A particular instruction may be scheduled to be executed on any one of a subset of the available functional units. The `L/S` unit is for load-store instructions only. Our model currently does not consider delays due to reorder-buffer unavailability or bus contention cycles. However, stalls due to data dependencies, functional unit occupancy, and limited fetch buffer space are considered. Speculative execution is not modeled.

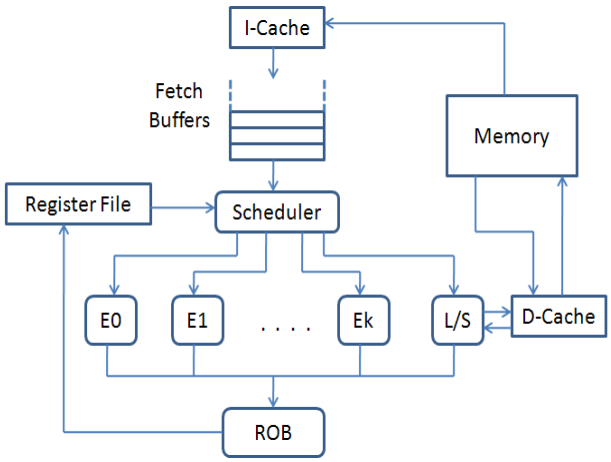

Figure 5. Processor Architecture

Specifications of instances of this model are used as inputs to our analysis engine that computes the Δ function as described in Section 5 A simple architecture description language is used to describe the processors. The size of the state space is reduced by exploiting symmetries in the specification. For example, instructions are classified into types according to their execution latencies. The analyzer first builds a transition system corresponding to this specification. Each state in the transition system comprises of the states of the functional units, fetch buffers, and fetch and data accesses in transit. The edge between any two adjacent states, S_1 and S_2 in this transition system is annotated with the pair $(t, \mathtt{I})$ that indicates a transition from state S_1 to S_2 after t cycles due to execution of instruction `I`. Once we have built this transition system we can construct the constraint system described in the previous section in a straightforward way.

Figure 6 shows part of the transition system for a processor with a domino effect. In this processor, functional unit `E0` is optimized for instruction type `I1` whereas `E1` is optimized for `I2`. `I1` needs 2 cycles to execute on `E0`, but 4 cycles on `E1`. The timing requirements for `I2` are opposite. There are 2 fetch buffers that are drained in order and Icache hit time is 1 cycle. In the figure, `Ij:m` indicates that instruction type `Ij` is in its $\mathtt{m}^{th}$ cycle of execution on that functional unit. A shaded functional unit indicates a new instruction dispatch (`m`=1) that cycle. The figure shows that the timing difference between paths starting from states S_0 and S_4 is 1 clock cycle per loop. The timing difference is thus unbounded on the instruction sequence $(\mathtt{I1.I2})^*$. It is useful to use the cycle ratio (4/3) in this case.

The transition system for a simple processor example with 2 instruction types, 2 functional units, execution times ranging from 2 to 6 cycles, 4 fetch buffers and no domino effects had 555 states and generated 97340 constraints. The Δ function ranged from 0 through 7 with 0s forming 88.1% of

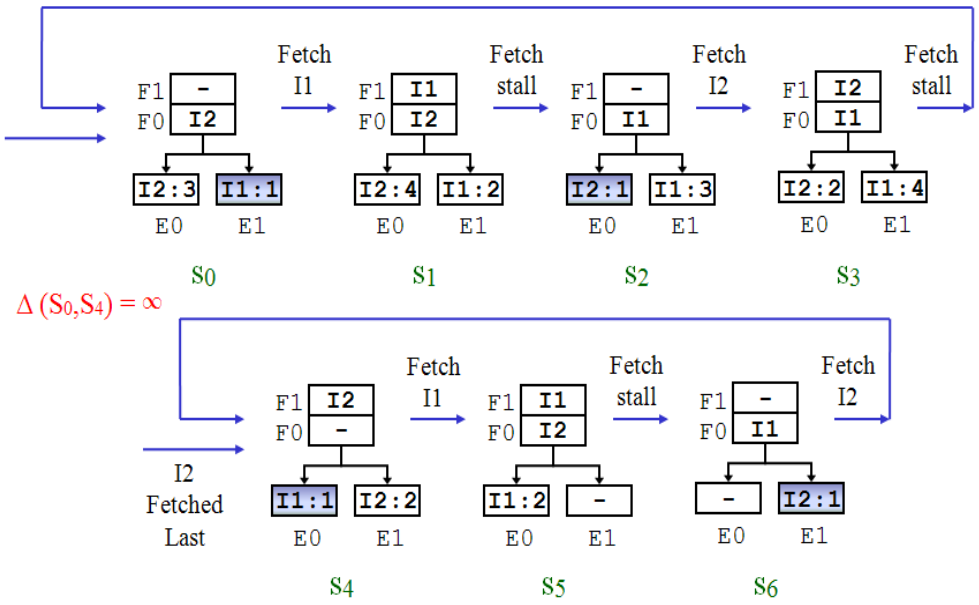

Figure 6. Example with Domino Effects

the distribution. Currently we have not experimented with full specifications of real processors. Our preliminary investigations with scaled down models of processors suggest that the constraint graphs are usually sparse: the number of inequations for most of the toy models typically lie well within 0.1% of the theoretical bound of $|S|^4$, where S is the set of states in which a new instruction is fetched.

7. Conclusions and Future Work

We presented a simple model of micro-architectural analysis which resembles several existing analyses. Timing anomalies in abstract hardware models prevent sound *and* efficient micro-architectural analysis. To enable efficient and yet sound analysis, we introduced Δ functions, which allow to safely prune analysis states even for architectures that exhibit timing anomalies. We have shown how to compute valid Δ functions for a hardware model and evaluated our approach on two example architectures. We have also introduced valid ρ and δ functions, which allow to prune states even in the presence of domino effects. On the way, we arrived at a slightly simpler definition of timing anomalies than in [15] and the first formal definition of domino effects.

In future work, we plan to apply our approach to real architectures, like the MOTOROLA POWERPC 75X or the ARM9, and evaluate the improvement in analysis efficiency. We have seen that valid Δ functions also allow to trade precision for additional efficiency. It will be interesting to study this trade-off on real architectures.

Acknowledgements The authors would like to thank Daniel Grund and Claire Burguière for valuable comments on drafts of this paper.

References

[1] C. Berg. PLRU cache domino effects. In *WCET '06*. Schloss Dagstuhl, Germany, 2006.

[2] P. Cousot and R. Cousot. *Building the Information Society*, chapter Basic Concepts of Abstract Interpretation, pages 359–366. Kluwer Academic Publishers, 2004.

[3] J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, 2002.

[4] J. Engblom and B. Jonsson. Processor pipelines and their properties for static WCET analysis. In *EMSOFT'02*, pages 334–348, London, UK, 2002. Springer-Verlag.

[5] A. Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, 2003.

[6] C. Ferdinand and R. Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2-3):131–181, 1999.

[7] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*, 17(2):416–429, 1969.

[8] C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *RTSS'95*, pages 288–297, Dec. 1995.

[9] R. Kirner, A. Kadlec, and P. Puschner. Precise worst-case execution time analysis for processors with timing anomalies. In *ECRTS'09*, July 2009.

[10] X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for WCET analysis. *Real-Time Systems*, 34(3):195–227, November 2006.

[11] T. Lundqvist. *A WCET Analysis Method for Pipelined Microprocessors with Cache Memories*. PhD thesis, Chalmers University of Technology, Sweden, June 2002.

[12] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *RTSS'99*, Washington, DC, USA, 1999. IEEE Computer Society.

[13] V. R. Pratt. Two easy theories whose combination is hard. Technical report, Massachusetts Institute of Technology, 1977.

[14] J. Reineke. *Caches in WCET Analysis*. PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, November 2008.

[15] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. A definition and classification of timing anomalies. In *WCET'06*. Schloss Dagstuhl, Germany, July 2006.

[16] C. Rochange and P. Sainrat. A context-parameterized model for static analysis of execution times. *Trans. on HiPEAC*, 2(3):109–128, 2007.

[17] J. Schneider. *Combined Schedulability and WCET Analysis for Real-Time Operating Systems*. PhD thesis, Saarland University, Germany, Saarbrücken, Germany, December 2002.

[18] R. E. Tarjan. Shortest paths. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, 1981.

[19] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18(2/3):157–179, May 2000.

[20] S. Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, Saarbrücken, Germany, 2004.

[21] I. Wenzel, R. Kirner, P. Puschner, and B. Rieder. Principles of timing anomalies in superscalar processors. In *Proc. 5th International Conference on Quality Software*, Sep. 2005.