Static Timing Analysis – What is special?*

Jan Reineke and Reinhard Wilhelm

Informatik, Saarland University
Saarbrücken

Abstract. Abstract interpretation is successfully used for determining execution-time bounds of real-time programs. The particular problem it solves is the determination of invariants at all program points that describe the set of all execution states that are possible at these program points. These invariants are then used to exclude some of the possible costly executions of instructions, thereby reducing the execution-time bounds. This article considers the properties of this application of abstract interpretation that differ from those in the standard applications of abstract interpretation in compilation and in verification. It also shows how some particular designs of the underlying abstract domains made efficient timing analysis possible.

1 Introduction

1.1 Timing Analysis

Timing analysis of embedded real-time programs attempts to determine tight upper, and sometimes also lower bounds on the execution times of the programs. Ideally, one would find out the worst-case and best-case execution times. This is possible in principle since real-time programs are programmed in a way that termination is guaranteed, and since the execution platform has only finite resources. However, the complete exploration of the associated state space would take far too long to be practically feasible.

It was therefore clear that *abstraction* would need to be applied to arrive at sound execution-time bounds in acceptable times. Our entry into the timing-analysis area started with the (quite successful) attempt to predict the cache behavior by abstract interpretation [1, 2]. Abstract interpretation had not been applied to the timing-analysis problem. The existing approaches were rather ad hoc and of doubtful correctness. It turned out that using abstract interpretation was the recipe for success.

* Work reported herein was supported by the Deutsche Forschungsgemeinschaft in the Transregional Research Center AVACS, Automatic Verification and Analysis of Complex Systems.

1.2 What is different?

The standard textbook on static program analysis, authored by Flemming Nielson, Hanne Riis Nielson, and Chris Hankin, covers most needs of a designer of abstract interpretations. At least this was what we thought when we started out to design static analyses of the cache behavior of real-time programs. However, it turned out that timing analysis offers a number of challenges that were not foreseen by the existing theory or not used in previous practice. Here is a list, some items more absent from the Nielson/Nielson/Hankin book than others, some that could be covered by adaptations of the theory. Some concern cache analysis, others more general timing analysis.

- All traditional static program analyses we found in the literature were, in the best case, based on a semantics of the programming language that abstracted from the underlying execution platform. However, any timing analysis needs to *talk about* architectural behavior. Hence, the behavior of the execution platform must be an integral part of the semantics of the programming language, on which the static analysis is based.
- Any timing analysis is composed of many component analyses, one for each architectural component contributing to the timing behavior of programs. These component analyses interact in possibly complex ways, essentially originating from the dependencies of the architectural components on each other. Worst are cyclic dependencies since they render separate analyses of component behavior more or less impossible [29]. An adequate design of the individual analyses and of the composition is needed to arrive at overall timing analyses that are both precise and efficient.

  This composition of timing analysis of many component analyses is in contrast with the application of static program analysis in compilation, where typically one static analysis checks the applicability of one program transformation [3]. It is also different from the composition of abstract domains as in [4] used to increase precision of one static analysis by using information from another one.
- Timing analyses need to analyze programs on the executable level since the source level does not contain the information on memory allocation of instructions and data, indispensable for cache analyses, nor the information on when memory is accessed, needed for the analysis of bus-access conflicts.
- The replacement strategy of a cache architecture always needs some bookkeeping mechanism about past memory accesses. The state repre-

sentation of this mechanism is optimized for the speed and the size of the update logic. Cache analysis, however, is interested in the state itself, not its representation. The design of cache analyses therefore starts with a lossless abstraction of the HW implementation of the cache.
This is somewhat comparable to the situation in *shape analysis* [5], which is based on a *storeless semantics* abstracting from actual heap addresses, but keeping connectivity information.

- The cache semantics shows many *indirect effects* of state changes of one object on another independent object, e.g. memory block a is loaded into the cache, thereby replacing memory block b.
  These indirect effects are different from the ones caused by manipulations through pointers: These may also have side-effects, however, only on *aliases*, i.e. pointer expressions reaching the same object. So, these indirect effects result from program execution.
  In contrast, cache loads have indirect effects on objects related by the execution platform, i.e., by the cache-set mapping.
- A particularly hard problem is the static analysis of *write-back caches*. Here a modification of the contents of a memory block a residing in the cache leads to a temporary inconsistency in the value of a in the memory hierarchy. This inconsistency is repaired by a write back, possibly much later, when a is evicted by loading some other memory block b. These delayed cause-effect chains are quite unusual in the semantics of programming languages and therefore also in traditional static program analyses.
- The *invariant* at a program point, computed by some static analysis, may have different expressivity (precision), depending on wether the invariant is to hold for all executions reaching this program point or only for a subset corresponding to a particular context and/or a particular control flow. Traditional static program analyses may therefore be *context-* and/or *flow sensitive* or *insensitive* depending on the desired precision of the results and the required effort. The notion of *context* is defined by some abstraction of the set of call strings.
  Timing analyses must be flow-sensitive in order to obtain any precision at all. In addition, timing analyses need and use a generalized notion of *context* to be precise. Different iterations of a loop may have vastly different execution times. Hence, they have to be considered as contexts for the instructions in the body. This is an instance of trace partitioning, invented before trace partitioning was proposed in the literature [6].
- Static analyses of *concurrent systems* focus on the interaction of the concurrently executed tasks on *global variables*. In contrast to this,

cache analysis, and more general timing analysis, has to determine safe approximations of the *resource-occupancy interaction* [7]. An additional complication, compared to the static analysis of concurrent systems, is the non-transparency of which objects compete with which other objects for resources.

- Cache analysis, and more general timing analysis, determines invariants about execution states at program points and derives safety properties from these invariants, i.e., certain timing accidents like a cache miss will never happen at a program point. The proof of such safety properties allows reducing the execution-time bound by the timing penalty corresponding to the excluded timing accident. This use is different from that in traditional static program analyses used in verification, where such a safety property typically proves the absence of a run-time error.
- The profitability of code optimizations involving static analyses as check for their applicability is seldom clear. In contrast, excluding a timing accident by a strong invariant computed by a timing analysis is often associated with the elimination of a very clear penalty.

2 From Microarchitectures to Abstractions for Timing Analysis

When developing a timing analysis, the first task is to obtain a faithful *model* of the microarchitecture that the analysis is targeted at. This can be very challenging because documentation at the required level of detail is seldom available. One promising approach is to start from cycle-accurate models in hardware description languages like VHDL or Verilog if those are made available by the hardware manufacturer [8]. If such models are not provided by the manufacturer, they have to be constructed manually based on processor manuals and extensive measurements on evaluation boards. For some microarchitectural features, such as caches the modeling process can be partially automated [9]. In the remainder of this section, we assume that a cycle-accurate model of the microarchitecture has already been obtained by one of the ways described above.

Mathematically, a cycle-accurate model is a transition relation $R \subseteq S \times S$ that captures the behavior of the processor in a single execution cycle. Programs and their input data are part of the states S of the processor. So the initial states of a program P under all possible inputs are a subset I_P of S. The goal of timing analysis is then to determine a bound on the number of cycles from any possible state in I_P to the program's

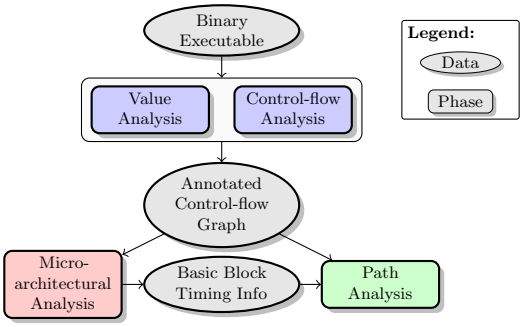

Fig. 1. Main components of a timing-analysis framework and their interaction.

termination, i.e., until it reaches one of its final states F_P. Brute-force exploration of all possible reachable states from the set of initial states is practically infeasible due to its large number. Therefore, a number of abstractions have been introduced to arrive at safe approximations of the worst-case execution time. In the following, we will discuss the two most important such abstractions.

2.1 Analysis Framework

Microarchitectures implement instruction set architectures (ISA). The semantics of binary programs in terms of the computed values in registers and memory are governed by the instruction set architecture. In particular, they are independent of its microarchitectural implementation.

As a consequence, analysis at the ISA level can be separated from analyses specific to a microarchitecture. This separation has led to the high-level structure of WCET analysis tools depicted in Figure 1. For a given instruction set, in a preprocessing step, a *value analysis* [10] determines the possible values of registers and memory locations, usually based on interval and congruence abstractions. The control-flow graph of the program under analysis is annotated with the results of value analysis for use in the subsequent analysis steps. They are required for precise data-cache analysis within microarchitectural analysis, as well as in control-flow analysis. *Control-flow analysis* [11–14] determines loop bounds and other characterizations of the set of semantically-feasible paths through the control-flow graph.

The task of *microarchitectural analysis* [15–19], which we will illuminate further in the following section, is to determine bounds on the execution times of small program fragments such as basic blocks. These bounds,

together with the results of the control-flow analysis are then used in *path analysis* [20, 21] to determine an upper bound on the execution time of the program as a whole. Path analysis is usually performed using integer linear programming formulations.

2.2 Separation into Value and Microarchitectural Analysis

Microarchitectural analysis first computes an overapproximation of the set of all reachable microarchitectural states. From this approximation and the transitions between the different reachable states, execution time bounds for basic blocks can be determined.

A relatively simple microarchitecture may consist of the following components: pipeline control, pipeline datapath, register file, branch predictor, cache, and main memory. The microarchitecture's space can be modeled as the cartesian product of the state spaces of its components:

$$S = \textsc{PipelineControl} \times \textsc{PipelineDatapath} \times \textsc{RegisterFile} \\ \times \textsc{BranchPredictor} \times \textsc{Cache} \times \textsc{Memory}$$

The set of reachable states of program P is the least fixed point of the *next* operator containing the initial states I_P of program P:

$$Col(P) = I_P \cup next(I_P) \cup next^2(I_P) \cup \ldots,$$

where *next* captures the effect of one execution cycle:

$$next(M) = \{s' \mid (s, s') \in R\}$$

The first abstraction, discussed informally in the previous section, is to perform *value analysis* separately, prior to *microarchitectural analysis*. Value analysis can be formalized as abstracting values in the register file and the memory.

Let $\textsc{Value}^\#$ be the abstract domain used in value analysis. A concretization function $\gamma_{VA} : \textsc{Value}^\# \to \mathcal{P}(\textsc{RegisterFile} \times \textsc{Memory})$ provides the meaning of a result of value analysis. Such analyses are flow- or even context-sensitive so that information about registers and memory is available for each program location separately. Formally,

$$\textsc{Value}^\# = (\textsc{Loc} \times \textsc{Context}) \to (\textsc{Register}^\# \times \textsc{Memory}^\#),$$

where Loc and Context are sets of program locations and contexts, and $\textsc{Register}^\#$ and $\textsc{Memory}^\#$ are abstractions of the register file and

memory, respectively. For reasons of brevity we cannot further elaborate on these abstractions. A correct abstract $next^{\#}_{VA} : \textsc{Value}^{\#} \to \textsc{Value}^{\#}$ operator guarantees global correctness of the value analysis.

Given value analysis results, microarchitectural analysis can thus focus on the remaining parts of the microarchitecture, pipeline control, as well as the state of the branch predictor and the cache. The state of the pipeline datapath can be inferred from the state of the pipeline control and the values of registers and memory, and is thus *not explicitly* represented by either value analysis or by microarchitectural analysis.

Let $\mu\textsc{Arch}^{\#}$ be the abstract domain used in microarchitectural analysis, and let $\gamma_{\mu A} : \mu\textsc{Arch}^{\#} \to \mathcal{P}(\textsc{PipelineControl} \times \textsc{BranchPredictor} \times \textsc{Cache})$ be its concretization function. While value analysis does not depend on microarchitectural analysis, the converse is not true. In particular, $next^{\#}_{\mu A}$ depends on the results of value analysis: $next^{\#}_{\mu A} : \mu\textsc{Arch}^{\#} \times \textsc{Value}^{\#} \to \mu\textsc{Arch}^{\#}$. For example, upon a memory access, microarchitectural analysis will query the results of value analysis, which have been annotated to the program's control-flow graph, to determine which memory block is being accessed to be able to classify the access as a cache hit or a cache miss.

For a correctness argument, the abstract operators $next^{\#}_{VA}$ and $next^{\#}_{\mu A}$ can be combined to obtain the abstract $next^{\#}$ operator as follows:

$$next^{\#}(v^{\#}, m^{\#}) := (next^{\#}_{VA}(v^{\#}), next^{\#}_{\mu A}(m^{\#}, v^{\#})).$$

Given correctness of $next^{\#}_{VA}$ and $next^{\#}_{\mu A}$, it can be shown that $next^{\#}$'s least fixed point,

$$Col^{\#}(P) = (i^{\#}_v, i^{\#}_m) \sqcup next^{\#}(i^{\#}_v, i^{\#}_m) \sqcup {next^{\#}}^2(i^{\#}_v, i^{\#}_m) \ldots,$$

overapproximates the set of reachable states $Col(P)$, with the combined concretization function

$$\gamma(v^{\#}, m^{\#}) := \{(pc, pd, rf, bd, c, m) \in S \mid (rf, m) \in \gamma_{VA}(v^{\#}) \wedge (pc, bd, c) \in \gamma_{\mu A}(m^{\#}), \}$$

given that $\gamma(i^{\#}_v, i^{\#}_m) \supseteq I_P$.

From the formalization it is apparent that value analysis can be performed in a preprocessing step, as it does not depend upon the results of microarchitectural analysis. This preprocessing step produces a control-flow graph annotated with the results of value analysis, which is then used by microarchitectural analysis.

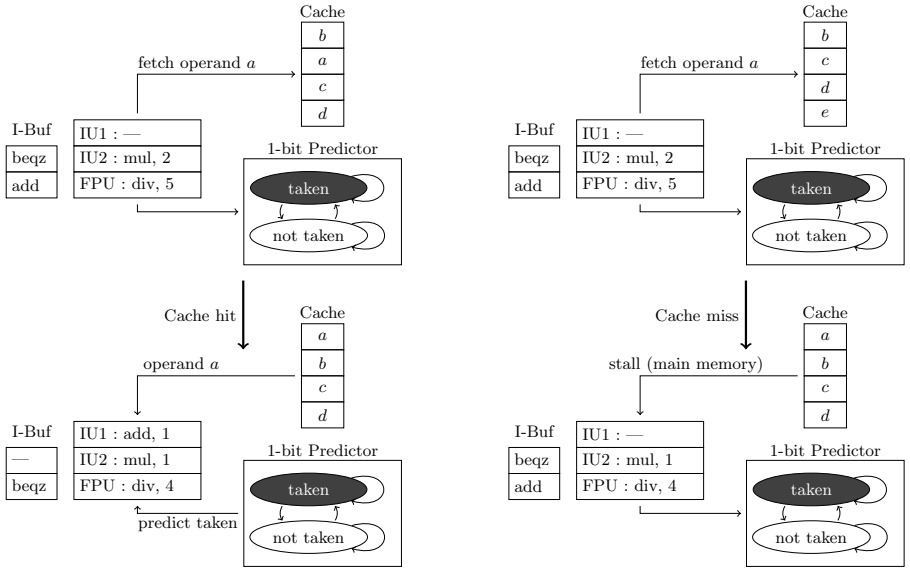

Fig. 2. Transitions from two different initial states of a simple processor consisting of an in-order pipeline, a 2-way fully-associative cache, and a 1-bit branch predictor.

2.3 Microarchitectural Analysis

Now let us turn to the internal structure of μARCH$^{\#}$. Can the pipeline control, the branch predictor, and the cache be analyzed independently of each other? Unfortunately, this is not the case due to the mutual dependencies of the three components. This is best explained with the help of Figure 2. At the top, we see two microarchitectural states of a simple processor, consisting of an in-order pipeline containing an instruction fetch buffer, two integer units, one floating-point unit, a cache, and a 1-bit branch predictor. The two processor states initially only differ in their cache states. The pipeline is about to dispatch the *add* instruction from the instruction fetch buffer to integer unit 1. Assume this *add* instruction adds the contents of a memory address to the contents of a register. Then, the instruction can be dispatched as soon as the memory operand is available. In the state on the left, the operand a is in the cache, and so *add* is dispatched immediately. On the other hand, in the state on the right, operand a needs to be fetched from memory, as it is not in the cache, and the *add* instruction cannot be dispatched yet. So the cache state has an influence on the pipeline state.

Now consider the successor states. On the left, the next instruction from the instruction fetch buffer to execute is a *branch equal zero* instruction. As

the condition upon which the branch depends has not yet been evaluated, the pipeline queries the branch predictor to decide in which direction to speculate. The prediction influences which instruction to fetch next, which in turn will affect the cache contents. So the future cache state depends on the current state of the pipeline and the branch predictor. Due to this tight coupling of the three components, they need to be analyzed relationally to obtain reasonably precise results.

The example also demonstrates that pipeline states cannot easily be ordered in terms of "progress": Intuitively, the successor state on the left has progressed further than the state on the right, as the *add* instruction has already been dispatched in this case. However, if speculative execution proceeds in the wrong direction, the pipeline state on the left may result in a longer execution time than the state on the right, which has no potential to speculate. Due to this lack of a natural ordering, which are usually the basis of abstractions[1], no efficient and precise abstractions are known so far for sets of pipeline states. In Section 3, we speculate about the design of abstractable pipelines and its abstraction.

The analysis essentially operates on the power-set domain of sets of concrete pipeline states, where only the datapath is abstracted away, as discussed earlier. For the cache, however, precise and efficient abstractions have been found.

2.4 Two abstractions for caches

The abstraction described in the following applies to caches with least-recently-used (LRU) replacement and was originally proposed by Ferdinand and Wilhelm [18]. For simplicity we assume a fully-associative cache, i.e., the cache consists of a single cache set, as the example cache in Figure 2. The extension to set-associative caches is straightforward, as set-associative caches can be seen as cartesian products of multiple independent fully-associative caches, each of which can be abstracted independently of the others.

A lossless logical abstraction The first abstraction to perform in the analysis of caches is a cache's physical implementation to a formal, logical model of its behavior. In physical implementations, caches consists of multiple memories containing data, tags, and status bits. In particular, each cache line is associated with a tag to keep track of which memory

[1] In interval analysis for example a set of values is abstracted by its least and their greatest element.

block is cached in the respective line. In addition, a number of status bits are maintained in each cache set to record the "logical" state of the replacement policy.

For instance, an implementation of least-recently-used replacement needs status bits to remember in which order the cache lines of each set have been used. Abstracting from the data stored in the cache, a model of a fully-associative cache with LRU replacement fairly close to the physical implementation might thus consist of two functions: 1) a function $cl : \{1, \ldots, k\} \to \mathcal{B}$ that captures which memory block is stored in each of the k cache lines, and 2) a function $age_{cl} : \{1, \ldots, k\} \to \{0, \ldots, k-1\}$ that maintains the "age" of each cache line, i.e., the number of distinct cache lines that have been accessed since the last access to the given cache line.

For cache analysis it is irrelevant in which physical cache line a memory block is stored; only the relative ages of different cached memory blocks are required to predict the future cache hit behavior. Thus, a lossless abstraction can be applied that captures the age of each memory block $age : Cache = \mathcal{B} \to \{0, \ldots, k-1, k\}$, where uncached blocks assume age k. One can relate the two models by an abstraction function α defined as follows:

$$\alpha(cl, age_{cl}) := \lambda b \in \mathcal{B} : \begin{cases} age_{cl}(i) & : if\ cl(i) = b \\ k & : if\ cl(i) \neq b \forall i \in \{1, \ldots, k\} \end{cases}$$

Upon a load of memory block b, the ages are updated as follows:

$$up(age, b) := \lambda b' \in \mathcal{B} : \begin{cases} 0 & : if\ b' = b \\ age(b') & : if\ age(b) \leq age(b') \\ age(b') + 1 & : if\ age(b) > age(b') \end{cases}$$

An interval abstraction Cache analysis needs to represent sets of cache states. In particular, at program start no knowledge about the cache state may be available, and so cache analysis needs to represent all possible cache states. Obviously, explicit representations are practically infeasible in such cases. A further abstraction is required to compactly represent large sets of cache states with little precision loss.

Fortunately, such abstractions are possible in the case of LRU. This is because LRU exhibits a form of *monotonicity*. Intuitively, the "younger" a memory block, i.e., the lower its age, the better. Thus, it is sufficient to maintain upper and lower bounds on the age of each memory block

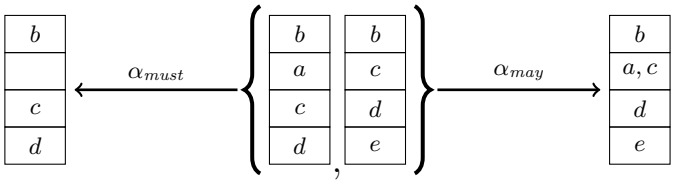

Fig. 3. Must and may cache abstractions.

independently of the ages of the other memory blocks. This yields the following abstract domain

$$\widehat{CacheInterval} = \{(l,u) \mid l,u \in \mathcal{B} \to \{0,\ldots,k-1,k\} \\ \wedge \forall b \in \mathcal{B} : l(b) \leq u(b)\}$$

storing a lower and an upper bound on the age of each memory block.

In the literature, the two analyses have been proposed separately, where lower bounds are maintained in what is called *may analysis* and upper bounds are maintained in *must analysis*. Lower bounds can be used to reason about which memory blocks *may* be cached, whereas upper bounds can be used to reason about which memory blocks *must* be cached.

$\widehat{CacheInterval}$ forms a join semi-lattice with the following order:

$$(\widehat{a}_{may}, \widehat{a}_{must}) \sqsubseteq (\widehat{a}'_{may}, \widehat{a}'_{must}) :\Leftrightarrow \widehat{a}_{may} \sqsubseteq_{may} \widehat{a}'_{may} \wedge \widehat{a}_{must} \sqsubseteq_{must} \widehat{a}'_{must}$$
$$\widehat{a}_{may} \sqsubseteq_{may} \widehat{a}'_{may} :\Leftrightarrow \forall b \in \mathcal{B} : \widehat{a}_{may}(b) \geq \widehat{a}'_{may}(b)$$
$$\widehat{a}_{must} \sqsubseteq_{must} \widehat{a}'_{must} :\Leftrightarrow \forall b \in \mathcal{B} : \widehat{a}_{must}(b) \leq \widehat{a}'_{must}(b)$$

Abstract cache states are related to sets of concrete cache states by a *Galois connection* via the following abstraction and concretization functions:

$$\alpha(C) := (\alpha_{may}(C), \alpha_{must}(C)), \mathit{with}$$
$$\alpha_{may}(C) := \lambda b \in \mathcal{B} : \min_{age \in C} age(b)$$
$$\alpha_{must}(C) := \lambda b \in \mathcal{B} : \max_{age \in C} age(b)$$

and

$$\gamma(\widehat{age}_{may}, \widehat{age}_{must}) := \gamma_{may}(\widehat{age}_{may}) \cap \gamma_{must}(\widehat{age}_{must}), \mathit{with}$$
$$\gamma_{may}(\widehat{age}) := \{age \mid \forall b \in \mathcal{B} : \widehat{age}(b) \leq age(b)\}$$
$$\gamma_{must}(\widehat{age}) := \{age \mid \forall b \in \mathcal{B} : age(b) \leq \widehat{age}(b)\}$$

In Figure 3, the two abstractions are illustrated at the example of the set of cache states found in the two initial states of Figure 2. In the concrete cache states the i^{th} row contains the memory block with age i. In the abstract cache states the i^{th} row contains all memory blocks with age bound i.

The abstract update functions for the lower and upper bounds closely resemble the concrete update function and can be proven correct rather easily:

$$up_{may}(\widehat{age}, b) := \lambda b' \in \mathcal{B} : \begin{cases} 0 & : \mathit{if}\ b' = b \\ age(b') & : \mathit{if}\ \widehat{age}(b) < \widehat{age}(b') \\ age(b') + 1 & : \mathit{if}\ \widehat{age}(b) \geq \widehat{age}(b') \neq k \\ k & : \mathit{if}\ \widehat{age}(b') = k \end{cases}$$

$$up_{must}(\widehat{age}, b) := \lambda b' \in \mathcal{B} : \begin{cases} 0 & : \mathit{if}\ b' = b \\ age(b') & : \mathit{if}\ \widehat{age}(b) \leq \widehat{age}(b') \\ age(b') + 1 & : \mathit{if}\ \widehat{age}(b) > \widehat{age}(b') \end{cases}$$

Integration of cache analysis within microarchitectural analysis As discussed earlier, no good abstractions for sets of pipeline control states are known, and so they are analyzed using a power-set domain. How can the analysis of the pipeline be integrated with the analysis of the cache behavior? Due to their mutual dependencies they need to be analyzed in a relational manner. The idea is to associate with each possible pipeline state, *one* cache state:

$$\mu\widehat{\textsc{Architecture}} := \textsc{PipelineControl} \rightarrow (\widehat{\mathit{CacheInterval}} \cup \{\bot\}),$$

where $\bot$ is used to express that the respective pipeline state is not possible. For simplicity, here, we omit branch predictors, which can be treated similarly to caches.

Other replacement policies We have seen a precise and efficient abstraction for caches with LRU replacement. For other replacement policies, similarly efficient abstractions have been developed [22–27]. However, they do not reach the same level of precision as the replacement policies are less predictable [28].

3 An Abstractable Pipeline

We have seen in the previous section that pipelines in modern high-performance microprocessors don't provide for compact abstractions similar to abstract cache states. This forced the pipeline analysis to work with a power-set domain [17]. We now speculate about an abstractable instruction pipeline, i.e. an instruction pipeline that has a compact abstract domain, simple update and join functions, and thereby admits efficient and precise pipeline analysis. The goal is to have an abstract instruction pipeline which looks much like a concrete instruction pipeline. The concrete state of an instruction pipeline contains a set of instructions of a given program, each in one particular pipeline stage. In addition, the pipeline is connected to a set of queues, buffers, and functional units holding instructions to be fetched next, stores to still be executed, or operations under execution.

The progress in executing a given program, as given by a particular concrete pipeline state, consists in

- which instructions of the program have already retired from the pipeline,
- how far other instructions of the program have progressed in the pipeline,
- how many instructions to be executed next have been prefetched into prefetch queues,
- how far operations dispatched by instructions currently in the pipeline have progressed in the pipelined functional units,
- how many outstanding stores are still in the store buffer.

An abstract state of an instruction pipeline, as we envision it, should look much like a concrete pipeline state. However, the interpretation (*concretization*) is different:

- Any progress of an instruction in the abstract pipeline state is a guaranteed minimal progress of the instruction.
- the contents of the abstract prefetch queue is a sequence of instructions, guaranteed to have been prefetched,
- the progress of dispatched operations in the functional units is guaranteed progress, and
- the stores removed from the store buffer have definitely been performed, the ones still in the store buffer may be still outstanding.

This notion of progress is the basis for defining a partial order of the abstract pipeline domain.

Let us discuss the implication for the pipeline architecture. It means that the pipelines should be an *in-order pipeline*, i.e., without reordering of instructions. An out-of-order pipeline admits several dynamically selected schedules of a given sequence of instructions. The join function would be applied to the different schedules resulting in an abstract pipeline state where each instruction is recorded with its slowest possible progress. Thus the effect of out-of-order execution would be completely lost in the pipeline analysis.

A first step towards *abstractable* pipelines has been done in [29]. We proposed a *strictly in-order* pipeline, i.e., one where no phase of a later instruction can block execution of a phase of an earlier instruction. This restriction excludes timing anomalies, which were still possible in in-order pipelines, against common beliefs. The simple pipeline design admits a compact abstract domain based on the maximally guaranteeable progress.

Pipeline analysis is typically performed on basic blocks. For each predecessor block of a basic block to be analyzed it has produced a final abstract state. These final states need to be combined to an initial state by applying the join function of the abstract domain. Different predecessor basic blocks will consist of different instruction sequences, such that their final abstract states will have different subsequences of instructions in the pipeline. Joining the set of abstract final states would roughly correspond to flushing the pipeline, a costly approach if the pipeline is deep. The efficiency gain of overlapping execution across basic-block boundaries would always get lost. A way out of this dilemma could consist in delaying the join at the beginning of basic blocks until the remaining instructions of the predecessor blocks have retired.

4 Conclusions

We have shown how the architectural basis of static timing analysis influences the character of static timing analysis, which includes a particular instance of abstract interpretation as its most important component. In particular, we have described how two important transformations of the underlying complex cartesian-product domain were needed and successfully used to arrive at efficient analyses.

References

1. Alt, M., Ferdinand, C., Martin, F., Wilhelm, R.: Cache behavior prediction by abstract interpretation. In: Static Analysis, Third International Symposium, SAS'96. (1996) 52–66

2. Ferdinand, C.: Cache Behavior Prediction for Real-Time Systems. PhD thesis, Saarland University, Saarbruecken, Germany (1997) ISBN: 3-9307140-31-0.
3. Seidl, H., Wilhelm, R., Hack, S.: Compiler Design - Analysis and Transformation. Springer (2012)
4. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In Aho, A.V., Zilles, S.N., Rosen, B.K., eds.: Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, ACM Press (1979) 269–282
5. Sagiv, M., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. **24** (2002) 217–298
6. Mauborgne, L., Rival, X.: Trace partitioning in abstract interpretation based static analyzers. In: Programming Languages and Systems, 14th European Symposium on Programming. (2005) 5–20
7. Abel, A., Benz, F., Doerfert, J., Dörr, B., Hahn, S., Haupenthal, F., Jacobs, M., Moin, A.H., Reineke, J., Schommer, B., Wilhelm, R.: Impact of resource sharing on performance and performance prediction: A survey. In D'Argenio, P.R., Melgratti, H.C., eds.: CONCUR 2013 - Concurrency Theory. Volume 8052 of Lecture Notes in Computer Science., Springer (2013) 25–43
8. Schlickling, M., Pister, M.: Semi-automatic derivation of timing models for WCET analysis. In: Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems, ACM (2010) 67–76
9. Abel, A., Reineke, J.: Measurement-based modeling of the cache replacement policy. In: RTAS. (2013)
10. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, New York, NY, USA, ACM Press (1977) 238–252
11. Ermedahl, A., Gustafsson, J.: Deriving annotations for tight calculation of execution time. In: Euro-Par. (1997) 1298–1307
12. Healy, C., Sjödin, M., Rustagi, V., Whalley, D., van Engelen, R.: Supporting timing analysis by automatic bounding of loop iterations. Real-Time Sys. (2000) 129–156
13. Stein, I., Martin, F.: Analysis of path exclusion at the machine code level. In Rochange, C., ed.: 7th International Workshop on Worst-Case Execution Time Analysis (WCET'07). OpenAccess Series in Informatics (OASIcs) (2007)
14. Cullmann, C., Martin, F.: Data-flow based detection of loop bounds. (In: 7th International Workshop on Worst-Case Execution Time Analysis (WCET'07))
15. Ferdinand, C., Heckmann, R., Langenbach, M., Martin, F., Schmidt, M., Theiling, H., Thesing, S., Wilhelm, R.: Reliable and precise WCET determination for a real-life processor. In: International Conference on Embedded Software. Volume 2211 of LNCS. (2001) 469–485
16. Engblom, J.: Processor Pipelines and Static Worst-Case Execution Time Analysis. PhD thesis, Uppsala University, Sweden (2002)
17. Thesing, S.: Safe and Precise WCET Determinations by Abstract Interpretation of Pipeline Models. PhD thesis, Saarland University, Saarbrücken, Germany (2004)
18. Ferdinand, C., Wilhelm, R.: Efficient and precise cache behavior prediction for real-time systems. Real-Time Systems **17** (1999) 131–181
19. Cullmann, C.: Cache persistence analysis: Theory and practice. ACM Trans. Embed. Comput. Syst. **12** (2013) 40:1–40:25
20. Li, Y.T.S., Malik, S.: Performance analysis of embedded software using implicit path enumeration. In: Proceedings of the 32nd ACM/IEEE Design Automation Conference. (1995) 456–461

21. Theiling, H.: ILP-based interprocedural path analysis. In: International Conference on Embedded Software. Volume 2491 of LNCS., Springer (2002) 349–363
22. Reineke, J., Grund, D.: Relative competitive analysis of cache replacement policies. In: LCTES '08: Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems. (2008) 51–60
23. Grund, D., Reineke, J.: Abstract interpretation of FIFO replacement. In: Proceedings of the 16th International Symposium on Static Analysis. SAS '09 (2009) 120–136
24. Grund, D., Reineke, J.: Toward precise PLRU cache analysis. In Lisper, B., ed.: 10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010). (2010) 23–35
25. Grund, D., Reineke, J.: Precise and efficient FIFO-replacement analysis based on static phase detection. In: Proceedings of the 22nd Euromicro Conference on Real-Time Systems (ECRTS). (2010) 155–164
26. Guan, N., Yang, X., Lv, M., Yi, W.: FIFO cache analysis for WCET estimation: A quantitative approach. In: Proceedings of Design, Automation Test in Europe Conference Exhibition (DATE), 2013. (2013) 296–301
27. Guan, N., Lv, M., Yi, W., Yu, G.: WCET analysis with MRU cache: Challenging LRU for predictability. ACM Trans. Embed. Comput. Syst. **13** (2014) 123:1–123:26
28. Reineke, J., Grund, D., Berg, C., Wilhelm, R.: Timing predictability of cache replacement policies. Real-Time Systems **37** (2007) 99–122
29. Hahn, S., Reineke, J., Wilhelm, R.: Toward compact abstractions for processor pipelines. In Meyer, R., Platzer, A., Wehrheim, H., eds.: Correct System Design. Volume 9360 of LNCS., Springer (2015) 205–220