Toward compact abstractions for processor pipelines*

Sebastian Hahn, Jan Reineke
Informatik, Saarland University
Saarbrücken

Reinhard Wilhelm
Informatik, Saarland University and
AbsInt Angewandte Informatik GmbH
Saarbrücken

Abstract

Hard real-time systems require programs to react on time. Static timing analysis derives timing guarantees by analyzing the behavior of programs running on the underlying execution platform. Efficient abstractions have been found for the analysis of caches. Unfortunately, this is not the case for the analysis of processor pipelines. Pipeline analysis typically uses an expensive powerset domain of concrete pipeline states. Therefore, pipeline analysis is the most complex part of timing analysis. We propose a compact abstract domain for pipeline analysis. This pipeline analysis determines the minimal progress of instructions in the program through the pipeline.

We give a concrete semantics for an in-order pipeline, which forms the basis for an abstract semantics. On the way, we found out that in-order pipelines are not guaranteed to be free of timing anomalies, i.e. local worst decisions do not lead to the global worst case. We prove this by giving an example. A major problem is how to find an abstract semantics that guarantees progress on the abstract side. It turns out that monotonicity on the partial progress order is sufficient to guarantee this.

1 Introduction

In state-of-the-art timing analysis, microarchitectural analysis, i.e. the part dealing with the influence of the underlying hardware platform on the execution time behavior, is the most complex part. There are two main reasons for this.

*This work was supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the Transregional Collaborative Research Centre SFB/TR 14 (AVACS) and by the Saarbrücken Graduate School of Computer Science which receives funding from the DFG as part of the Excellence Initiative of the German Federal and State Governments.

The first one is the complexity of modern microprocessors, which feature (cyclic) interdependencies between components. These interdependencies make it hard to impossible to decompose the analysis into several more efficient sub-analyses. Additionally, they often result in so called timing anomalies (i.e. following local worst cases does not lead to global worst cases) that complicate the analysis.

The second, not quite independent, reason is that no compact abstract domain for pipeline analysis has been found, yet, making it necessary to fall-back to a very expensive powerset domain of concrete pipeline states. The resulting state space exploration has to investigate all possible transitions of instructions through the microarchitecture in order not to miss the worst-case behavior. In this context, the aforementioned timing anomalies prevent state space reduction based on local decisions.

There are approaches that tackle the efficiency problem. Wilhelm [11] uses a symbolic representation of the elements of the powerset domain. Other approaches almost exclusively try to overcome the complexity of modern microprocessors. They range from stepping-back towards very simplistic pipeline designs [7] to limiting processor features such that decomposition into more efficient analyses are possible. One example is the PRET architecture [5] that features a thread-interleaved pipeline basically leading to sequential execution w.r.t. a single thread.

Not much research has so far been undertaken to develop compact abstract domain for pipelines. In the following, we present (speculative) ideas and thoughts on how such a compact domain could look like. We also determine sufficient conditions on the concrete pipeline behavior that admit compact domains while leading to precise results. Enforcing these conditions in hardware can lead to degradation of the system's overall performance. Although it cannot be expected that the efficiency problem is ultimately "solved", i.e. compact domains for arbitrarily complex architectures are found, it is a step in this new direction. In any case, it will provide a better understanding of how to model the microarchitectural timing behavior.

2 Background

2.1 Pipelines

We consider a normal RISC-like 5-stage in-order pipeline as depicted in Figure 1, i.e. program instructions are executed in an overlapped fashion, but in the order they occur in the program. First, an instruction is fetched from memory. Second, the instruction is decoded and operands are fetched from the register file. Next, the instruction is executed and potentially a memory address is generated and the corresponding memory access is initiated. In the next stage, pending data memory operations are finished. Last, the results computed by the instruction are written back to the register file. The fetch stage and the memory stage

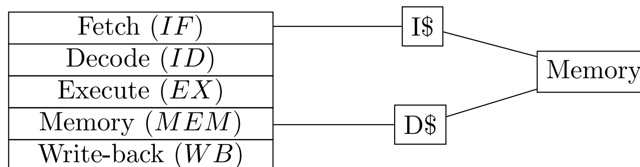

Figure 1: 5-stage RISC pipeline. Fetch and memory stage access a common background memory through separate caches.

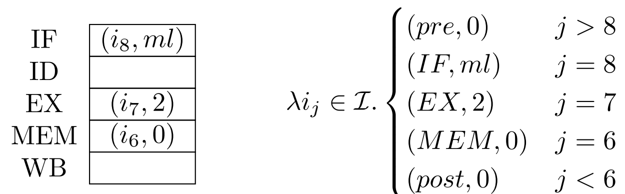

Figure 2: A concrete pipeline state in stage-centric and instruction-centric representation. The numbers denote the individual latencies, i.e. the cycles needed for the instruction to become ready in its current stage.

access a common background memory, possibly via separate instruction and data caches. The progress of an instruction in the pipeline is stalled when data dependencies would be violated or when an instruction is waiting for a memory access to be serviced.

Definition 1. *We call a pipeline* in-order *if each stage processes the instructions in the order they occur in the program.*

There are more advanced pipelining techniques that feature dynamic scheduling to reorder instruction and execute them out-of-(program)-order, speculation across branches, and additional buffers to decouple the pipeline and the memory hierarchy. We discuss the influence of these features on a compact representation for pipeline analysis in Section 5. For the remainder of this article, we focus on the presented in-order processor architecture.

2.2 Concrete Semantics of an In-Order Pipeline

In the following, we give a concrete semantics of an in-order pipeline. The remainder of this article is based on this concrete semantics.

As depicted in Figure 2, there are two equivalent views of a concrete pipeline state: A stage-centric view describing which stage is occupied by which instruction and an instruction-centric view describing which instruction occupies which stage. We select the second view because the abstract semantics, to be presented later, will represent the guaranteed progress of instructions through the pipeline. We use the first view for visualization purposes only.

Domain An instruction in the pipeline can occupy one of the stages IF, ID, EX, MEM, and WB. We further distinguish between instructions that have not yet entered the pipeline, which are in the conceptual stage "pre", and instructions that have already left the pipeline, which are in the conceptual stage "post". Together, we obtain the following set of stages:

$$\mathcal{S} := \{pre, IF, ID, EX, MEM, WB, post\}.$$

Some of the pipeline stages are multi-cycle, e.g. IF and MEM in case of a cache miss and EX in the case of expensive arithmetic operations like floating point division. Thus, we introduce counters that capture how many cycles an instruction needs to remain in its current stage until being able to advance to the next stage.

The concrete domain is then defined as

$$Pipe := \mathcal{I} \to \mathcal{S} \times \mathbb{N},$$

where $\mathcal{I}$ denotes the set of instruction instances that form the instruction sequence $i_1, i_2, \ldots, i_n$ occurring during program execution.

Cycle Update The cycle update $cycle : Pipe \to Pipe$ describes the concrete behavior of the pipeline informally described above, i.e. how a pipeline state changes during the execution of one processor cycle. The structure is quite generic and can be adapted to different pipeline designs: An instruction can advance in the pipeline if the instruction is ready to move to the next pipeline stage and this next pipeline stage would be free in the next cycle. An instruction might not be ready if there are unsatisfied data dependencies or it needs to wait for a memory transfer. In this case the instruction stays in the same stage, but its counter of remaining wait cycles might be decremented. If the next pipeline stage is still occupied in the next cycle, the instruction is stalled and stays unmodified in its current stage.

The next pipeline stage will be free in the next cycle if it is already free or if the instruction occupying it can move on to the next stage. An instruction in the WB stage is considered to always find its (fictive) next stage in the next cycle.

$$cycle(p : Pipe) :=$$
$$\lambda i \in \mathcal{I}. \begin{cases} (stage(i), cnt'(i)) & : \neg ready(i) \\ (stage'(i), latency(stage'(i), i)) & : ready(i) \wedge willbefree(stage'(i)) \\ (stage(i), cnt(i)) & : ready(i) \wedge \neg willbefree(stage'(i)) \end{cases}$$

where $(stage(i), cnt(i)) := p(i)$ and $cnt'(i)$, $stage'(i)$, $ready(i)$, $willbefree(s)$, and $latency(s, i)$ are defined as follows:

$$cnt'(i) := \begin{cases} cnt(i) - 1 & : cnt(i) > 0 \\ 0 & : cnt(i) = 0 \end{cases}$$

$$stage'(i) := \begin{cases} post & : stage(i) = WB \lor (stage(i) = ID \land i = nop) \\ WB & : stage(i) = MEM \\ MEM & : stage(i) = EX \\ EX & : stage(i) = ID \\ ID & : stage(i) = IF \\ IF & : stage(i) = pre \land tofetch(i) \\ pre & : stage(i) = pre \land \neg tofetch(i) \end{cases}$$

$$\begin{aligned} busfree &:= \neg\exists i.(stage(i) = IF \lor stage(i) = MEM) \land cnt(i) > 0 \\ ready(i) &:= (cnt(i) = 0) \land (stage(i) = ID \Rightarrow \text{data dependencies satisfied}) \\ &\quad \land (stage(i) = EX \land i = load/store \Rightarrow (dcachehit \lor busfree) \\ &\quad \land (stage(i) = pre \Rightarrow (icachehit \lor (busfree \land \neg dcachemiss)) \end{aligned}$$

$$\begin{aligned} willbefree(s) &:= (s = post) \lor (\neg\exists i.stage(i) = s) \\ &\quad \lor (\exists i.stage(i) = s \land ready(i) \land willbefree(stage'(i))) \end{aligned}$$

$$latency(s, i) := \begin{cases} 0 & : s \in \{ID, WB, pre, post\} \\ lat(i) & : s = EX \\ 0 & : (s = IF \land icachehit) \lor (s = MEM \land dcachehit) \\ ml & : (s = IF \land icachemiss) \lor (s = MEM \land dcachemiss) \end{cases}$$

where ml denotes the cache miss latency, and $lat(i)$ denotes the execution latency of instruction i that can depend on the operand values (e.g. in case of a division). Some of the values, i.e. lat, $icachehit$, $icachemiss$, $dcachehit$, $dcachemiss$, and $tofetch(i)$, depend on the environment of the pipeline that is known during an actual execution. A latency of 1 cycle is assumed for the access to an L1 cache. We omitted the state of the environment, such as caches, in this formulation for the sake of readability.

2.3 State-of-the-art Pipeline Analysis

Current pipeline analyses rely on expensive powerset domains as abstract domains. An abstract state is a set of concrete pipeline states. The abstract domain is thus given by

$$Pipe^{\#}_{ps} := (2^{Pipe}, \subseteq, \cup).$$

Basically, state-of-the-art pipeline analysis [4, 9] computes the so-called collecting semantics of pipeline states, i.e. it computes for each program point the set of concrete pipeline states that reach the program point during execution. The transfer function $f^{\#}_{ps} : Pipe^{\#}_{ps} \to Pipe^{\#}_{ps}$ can thus be defined using the concrete *cycle*-function applied element-wise. The (abstract) transfer of a single concrete pipeline state can nevertheless lead to several successors due to uncertainty in the environment. E.g. it might be uncertain whether a memory access hits or misses the caches, or what the values of operands are. In these cases, the necessary

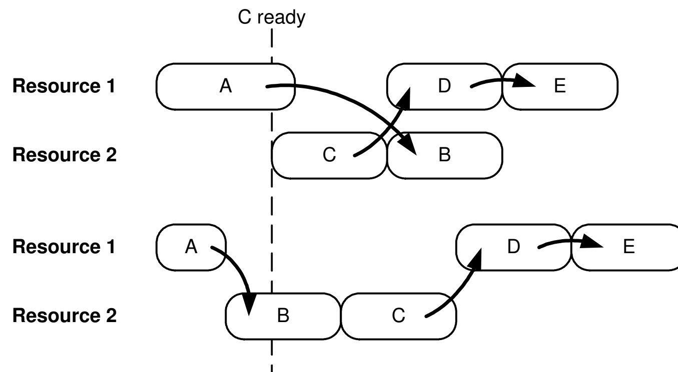

Figure 3: Timing Anomaly: The local worst case does *not* lead to the global worst case [6].

parts of the environment are concretized resulting in several environments. The *cycle* of the concrete state is then applied with each such (partially concretized) environment.

3 Are In-Order Pipelines Interesting? Or: What About Timing Anomalies?

Microarchitectural analysis has to cope with uncertainty about (components of the) execution states. First, it does not generally know the initial execution state, e.g. the initial occupation of the pipeline stages with old instructions. Secondly, the analysis works with an abstract model of the architecture, potentially omitting some details in a safe way, thereby introducing uncertainty in the pipeline and the environment (e.g. caches). Thirdly, it typically combines information about the execution states resulting from different paths leading to one program point. Uncertainty about some components of the execution state may result in non-deterministic decisions to be made by the analysis based on the powerset domain.

A (timing) anomaly is a scenario where the local worst-case in a non-deterministic decision does not lead to the global worst case. A classical example depicted in Figure 3 is a cache miss (locally worse than a cache hit) that leads to a shorter overall execution time, because in the hit case operations are re-ordered in a way better suiting the subsequent program. This is often referred to as scheduling anomaly which is typically present in architectures featuring out-of-order execution. These anomalies hinder local state space reductions that would simplify the state-exploring pipeline analysis described in the previous section.

In-order pipelines with timing anomalies have been discussed earlier in literature. The anomalous behavior was exclusively triggered by odd cache

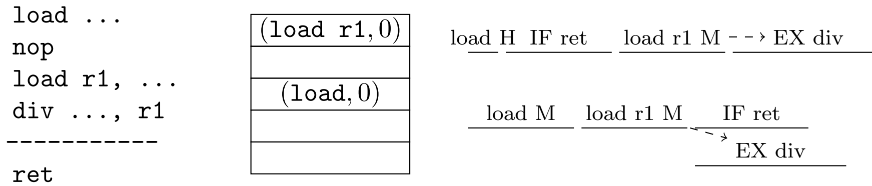

Figure 4: **Left**: The program to execute that leads to an anomalous behavior. **Middle**: The pipeline state serving as starting point for the anomaly. **Right**: The ongoing execution of the program demonstrating the anomaly.

behavior such as pseudo round-robin replacement [9] or partial cache line fills [3]. In-order pipelines with LRU caches have (implicitly) been considered anomaly-free as they execute instructions in a very regular fashion. As an example, Wenzel et al. [10] identified *resource allocation decisions* as necessary condition for timing anomalies. A resource allocation decision describes a situation where a latency variation causes instructions to execute in different orders in the functional units of an superscalar processor.

The pipeline described in Section 2.2 does not allow *resource allocation decisions*, but nevertheless features a timing anomaly. All instructions enter the (sole) functional unit in program order – independent of any latency variation. The definition of a resource allocation decision does not capture, that a latency variation at the beginning of an instruction sequence can have an anomalous impact on the *latency* (not the ordering) of later instructions. We will demonstrate that a timing anomaly is possible anyway and thus the property of in-order execution is not sufficient. However, the definition of Wenzel et al. [10] could be extended to also exclude reorderings of memory accesses.

Let us consider an in-order pipeline with separate caches, but common background memory as described earlier in Figure 1. The anomaly is based on the observation that in-order pipelines still allow the fetch of a instruction to be scheduled before the data access of a preceding instruction. In the following, we discuss the anomaly in more detail.

We make use of the following assumptions on the pipeline behavior that partially extend the formal description in Section 2.2. The pipeline has the ability to eliminate instructions from the pipeline that have no effect, e.g. nop instructions or predicated instructions whose condition is false. Furthermore, the pipeline features a long-lasting instruction such as floating-point division whose latency is at least as long as an L1 cache miss. This seems unrealistic at first glance, however when second-level caches are used to serve first-level misses it might very well be realistic.

Consider the program and (part of) its execution in Figure 4. The division instruction is data-dependent on the second load denoted by the dashed arrow in the right column. The dashed line in the left column denotes the beginning of a new cache line.

The pipeline state depicted in the middle column arises during execution of

the program and serves as a starting point for the timing anomaly. We consider the following cache environment: The first load instruction might hit (H) or miss (M) the data cache, resulting in a non-deterministic decision that triggers the anomalous behavior. The second load instruction misses the data cache. The fetch of the return instruction misses the instruction cache. The right column shows the ongoing execution of the program, demonstrating the possibility of a timing anomaly. The stage between the two loads is free resulting from the eliminated nop. The anomaly is due to the fact, that the second load can advance to the EX stage while waiting for the first load to complete its miss. Thus it can be started before the fetch of the return instruction which suits the execution of the data-dependent division instruction. In the hit case, the second load becomes ready too late and is blocked by the already ongoing instruction fetch.

This example demonstrates that in-order pipelines are not a priori anomaly-free. The existence of timing anomalies thus hinders local state space reductions even for in-order pipelines. So, they are interesting candidates for a compact abstract representation—getting rid of the expensive powerset domain and state space exploration.

4 Compact Abstract Pipeline Domain based on Minimal Progress

One idea for a compact representation of pipelines is inspired by our efficient cache analysis [2]. In this cache analysis, must and may analyses are employed to under-/overapproximate the cache content by tracking the maximal/minimal age of a cache block. In analogy, the idea for pipeline analysis is to track minimal/maximal progress of the instructions in the pipeline. The minimal progress metric can be used to guarantee that an instruction has eventually finished execution. One concern with minimal progress is, that – despite being conservative – some progress must be guaranteed for each call of the abstract domain's transfer function. Otherwise no bound can be derived.

4.1 Abstract Domain

First, we consider the pipeline behavior for one fixed instruction sequence $i_1 i_2 \ldots i_n$ with $i_j \in \mathcal{I}$. This eliminates uncertainty about the program's control-flow. We briefly discuss how to handle diverging and merging control-flow later.

Thus, the variation in execution times stems from cache uncertainty and variable-execution-latency instructions.

Minimal Progress The abstract domain maps each instruction in the sequence to its minimal progress

$$Pipe^{\#} := \mathcal{I} \rightarrow \mathcal{P}_{min},$$

where $\mathcal{P}_{min} := \mathcal{S} \times \mathbb{N}$.

Note, that the domain is identical to the concrete domain, however the interpretation of a domain element is different. An element $ap \in Pipe^{\#}$ describes for each instruction its *minimal progress*, i.e. the pipeline stage that the instruction reached *at least*. Thereby, we establish an ordering on concrete pipeline states.

Defining the partial order First, we define a progress order on the stages an instruction can be in. The idea behind the ordering is, that later stages are "better" in the sense that execution should not take longer starting from later stages. The order $\sqsubseteq_{\mathcal{S}}$ is then given by

$$post \sqsubseteq_{\mathcal{S}} WB \sqsubseteq_{\mathcal{S}} MEM \sqsubseteq_{\mathcal{S}} EX \sqsubseteq_{\mathcal{S}} ID \sqsubseteq_{\mathcal{S}} IF \sqsubseteq_{\mathcal{S}} pre.$$

Some of the stages are multi-cycle, so we extend this to an ordering on *progress* $\sqsubseteq_{\mathcal{P}_{min}}$ as follows

$$(s, n) \sqsubseteq_{\mathcal{P}_{min}} (s', n') \Leftrightarrow s \sqsubseteq_{\mathcal{S}} s' \vee (s = s' \wedge n \leq n').$$

As the ordering $\sqsubseteq_{\mathcal{P}_{min}}$ is total, the induced join is

$$p \sqcup_{\mathcal{P}_{min}} p' = (p \sqsubseteq_{\mathcal{P}_{min}} p') \text{ ? } p' : p.$$

The minimal-progress order on individual pipeline stages can then be extended to whole pipeline states. Two abstract pipeline states are ordered, if the minimal progress of all instructions is ordered:

$$s \sqsubseteq s' \Leftrightarrow \forall i \in \mathcal{I}. s(i) \sqsubseteq_{\mathcal{P}_{min}} s'(i).$$

Note, that the order respects the "has left the pipeline" property, i.e. whether an instruction has left the pipeline and thus finished its execution. Formally, if $s(i) = post$ for $i \in \mathcal{I}$ and $s \in Pipe^{\#}$, then

$$\forall s' \in Pipe^{\#}.\ s' \sqsubseteq s \Rightarrow s'(i) = post.$$

The join function $\sqcup$ is induced by the partial order $\sqsubseteq$ and corresponds to taking the minimal progress for each instruction:

$$s \sqcup s' = \lambda i \in \mathcal{I}. s(i) \sqcup_{\mathcal{P}_{min}} s'(i).$$

As an example consider the illustration in Figure 5.

Concretization/Abstraction Function As already noted, the concrete domain and the abstract domain based on minimal progress are structurally equivalent, yet their interpretations are different. Therefore, an abstract minimal-progress pipeline state can also be viewed as a concrete state and vice versa. Note, that we can use the partial order and join defined above for concrete pipeline states as well. Exploiting this, we give the concretization function γ :

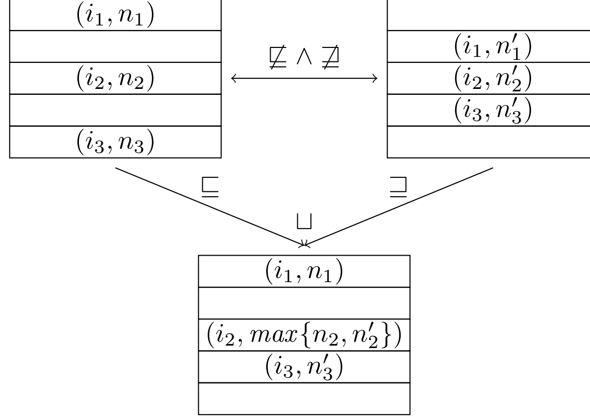

Figure 5: Example of minimal-progress based join function. Take the minimum of the progress of individual instructions.

$Pipe^{\#} \rightarrow 2^{Pipe}$ and abstraction function $\alpha : 2^{Pipe} \rightarrow Pipe^{\#}$ that relate our abstract domain to the collecting semantics domain and vice versa

$$\gamma(ap) := \{cp \in Pipe \mid cp \sqsubseteq ap\}$$
$$\alpha(CP) := \bigsqcup_{cp \in CP} cp$$

An abstract minimal-progress pipeline state ap thus describes all concrete pipeline states that have *at least* the progress of ap. The concretization and the abstraction function form a Galois connection [1].

4.2 Transfer Function

Before we discuss the transfer function, we present the general correctness criterion.

Definition 2 (Local Consistency and Best Abstract Transformer, [1])**.** *Let C and A be the concrete and abstract domains, and let $\gamma : A \rightarrow C$ and $\alpha : C \rightarrow A$ be the concretization and abstraction functions, and let $f : C \rightarrow C$ be the concrete transformer. An abstract transformer $f^{\#} : A \rightarrow A$ is* locally consistent *if and only if*

$$\forall a \in A.\ \gamma(f^{\#}(a)) \sqsupseteq_C f(\gamma(a)).$$

Let $f^{\#}_{best} = \alpha \circ f \circ \gamma$. If α and γ form a Galois connection, $f^{\#}_{best}$ is the best abstract transformer*. The best abstract transformer is locally consistent.*

Local consistency implies global consistency of the analysis results, i.e. the correctness of the overall analysis. Thus, it is sufficient to demonstrate local consistency of our abstract transformer as correctness proof.

Next, we try to come up with an abstract transformer for the abstract pipeline domain based on minimal progress.

The Easy Part

The transfer function takes a minimal progress abstract pipeline state from $Pipe^{\#}$ and computes the effect of the execution for a certain amount of time, e.g. one cycle. To be useful, the transfer function must always be able to guarantee strict progress in the sense of our partial order $\sqsubseteq$ defined in Section 4.1.

The question to answer is: On what does the progress of an instruction depend? Clearly, the progress of an instruction depends on whether the next stage will be free – which in turn depends on the progress the instructions in these stages will make in the *current* cycle. This can be observed directly as the function *willbefree* in Section 2.2 is recursive. As a consequence, the transfer function should proceed backwards through the pipeline, i.e. the progress of instructions in later stages should be determined first. An instruction at least in stage *write back* will be at least in stage *post* after one cycle – no further dependencies.

Next there are data dependencies that cause hazards, so it also matters how far the dependent instructions have at least advanced in the pipeline (see *ready* in Section 2.2). Most of the data dependencies can be removed by employing techniques like forwarding; however some remain. Consider a `load` instruction followed by an arithmetic operation depending on the loaded value. If the arithmetic operation is *at least* in the decode stage, progress can only be guaranteed if the load is *at least* in the write-back phase. Observe that data dependencies have the same "direction" – upstream instructions depend on downstream instructions – as "resource dependencies" discussed in the previous paragraph.

The Hard Part

Unfortunately, the progress dependencies can be bidirectional in general, i.e. the progress of an instruction may also depend on the progress of an instruction further upstream in the pipeline. As an example, consider an instruction in the memory stage just about to request memory as part of a data access that is blocked by an already ongoing instruction fetch. This is caused by the unified background memory with sequential access.

Why are bidirectional dependencies problematic? A downstream instruction may be stalled by an upstream instruction *and* vice versa. If under the abstraction, it cannot be determined which of the two instructions is progressing and which is stalled, then *no* progress is guaranteed for either of the two.

Consider the example abstract state in Figure 6. Recall that the positions of the instructions represent their *minimal* progress, i.e. instructions could be further down the pipeline in a concrete execution. Considering the data access that might just be about to happen: it could be blocked as the instruction fetch

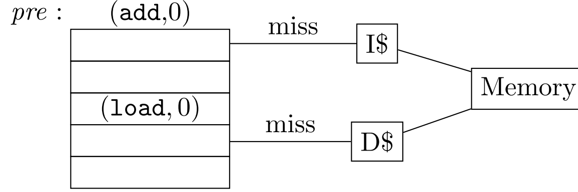

Figure 6: How can progress be guaranteed during cycle transfer of this minimal-progress abstract state?

could already have started. On the other hand, consider the instruction access that might just be about to happen: it could be blocked as the data access could already have started. Combining these two arguments, it follows that with the information available in the abstract domain, no progress can be guaranteed out of this state during one cycle. Essentially the pipeline deadlocks under the abstraction and no execution-time bound can be derived at all.

Note, that the uncertainty of whether instruction or data are scheduled for the bus does not exist within the powerset domain since in each concrete state it is always clear whether instruction fetch or data fetch acquire the bus first.

Monotonicity

A sufficient criterion to guarantee progress on each call of the transfer function $cycle^{\#}$ is monotonicity of the cycle update *cycle*. Monotonicity states that the transfer function preserves the ordering of states:

Definition 3 (Monotonicity)**.** *Let two states $s_1, s_2 \in Pipe$. We call the (concrete) transfer function cycle monotone if and only if*

$$s_1 \sqsubseteq s_2 \Rightarrow cycle(s_1) \sqsubseteq cycle(s_2).$$

Theorem 1. *The abstract transformer $cycle^{\#} := cycle$ is the best abstract transformer of the minimal-progress abstract domain if cycle is monotone.*

Proof. We have to prove that $cycle^{\#} = cycle = \alpha \circ cycle \circ \gamma$. By plugging in the definition of γ and α, the claim becomes

$$cycle(ap) = \bigsqcup_{cp \sqsubseteq ap} cycle(cp).$$

Using the monotonicity property of *cycle* concludes the proof. □

Corollary 1. *The abstract transformer $cycle^{\#} := cycle$ is a sound abstract transfer function of the minimal-progress abstract domain.*

Combined with the property that the order $\sqsubseteq$ respects whether instructions are finished, and given that we start with a correct initial value, it follows that the analysis leads to overall sound results.

However, the transfer function *cycle* as described in Section 2.2 is *not* monotone. This can be derived from the timing-anomalous behavior we described in Section 3. After one cycle, the state in the load-hit case made more progress compared to the load-miss case. But at the end, the load-miss case leads to a state that has progressed more.

Proposition 1 (Absence of Timing Anomalies)**.** *If the transfer function cycle is monotone, the powerset-domain-based analysis can safely follow local worst-cases.*

The terminology local worst-/best-case suggests that, after one cycle, the state following the local best-case should have at least the progress as the state following the local worst-case has. The monotonicity property guarantees, that further cycling will always preserve this progress ordering. Thus, it is safe to follow the state with the minimal progress (arising from the local worst-case) – if the above characterization of local worst-/best-case is appropriate.

Having completely separate instruction and data memory ensures monotonicity. Then, the progress of an instruction solely depends monotonically on the progress of instructions further down in the pipeline. However, this scenario is unrealistic – applications could rely on self-modifying code or need to load data from the instruction memory (e.g. constant pools).

Another "hardware" attempt to ensure monotonicity is to never start a memory request upon an instruction-cache miss as long as an instruction, potentially accessing data memory, could be blocked by this. This way, we enforce a stronger property than in-order execution as we defined it above. In-order execution for example still allows that the fetch (memory access) of a later instruction can be scheduled before the data access of an earlier execution. In some sense, the execution is not in-order w.r.t. to externally visible events such as the acquisition of the memory bus.

Definition 4 (Strictly In-Order)**.** *We call a pipeline* strictly in-order *if each* resource *processes the instructions in the order they occur in the program.*

These resources include the pipeline stages as well as the common background memory. The definition enforces, that all memory accesses of one instruction (i.e. the instruction fetch and potential data accesses) happen before any memory access of a later instruction.

Recall the definition of the concrete pipeline semantics in Section 2.2 that is not strictly in-order. We modify the underlying pipeline such that it becomes strictly in-order as follows:

$$\begin{aligned} \mathit{ready}(i) := \ldots\ \wedge (\mathit{stage}(i) = \mathit{pre} \Rightarrow (\mathit{icachehit} \vee \\ (\mathit{busfree} \wedge \forall \mathit{pr} \in \mathit{prev}(i).\ (\mathit{pr} \neq \mathit{ld/str} \vee \mathit{stage}(\mathit{pr}) \notin \{IF, ID, EX\}))). \end{aligned}$$

An instruction miss that could block the bus for earlier data memory accessing instructions is delayed until no such instructions are in the "critical area" any more. In the case of an instruction cache hit, no such actions are necessary as the caches are separated.

Proposition 2. *The strictly in-order pipeline just described is monotone in the sense of Definition 3.*

The detailed proof of this proposition is quite lengthy and therefore we only present a sketch here. Given two pipeline states s_1, s_2 such that $s_1 \sqsubseteq s_2$. The proof uses a case distinction of the progress of an instruction in s_2. The cases should be considered in a bottom-up fashion starting with *post* and ending with *pre*. This represents the progress dependencies on the progress of instructions further down the pipeline. Then, we exploit that each instruction has at least the same progress in s_1. Using the definition of the concrete semantics, it follows that $cycle(s_1) \sqsubseteq cycle(s_2)$.

Note that in general, even a strictly in-order pipeline may feature timing anomalies, e.g., if it contains multiple incomparable functional units as described by Wenzel et al. [10].

Outlook: Enriched Abstraction

An alternative to enforcing monotonicity of the concrete behavior of the pipeline by hardware modifications is to come up with more expressive abstractions. The idea is to enrich the abstraction with further (instrumented) properties about the time that has been spent at least in a specific stage.

An analogous idea has been successfully used in shape analysis via three-valued logic [8]. Additionally introduced *instrumentation predicates* made it possible to establish and preserve complex statements about heap-allocated data structures.

Transferred to our domain: To ensure that *some* progress is always made in the abstraction, one can instrument the semantics, so that every instruction tracks the number of cycles it has spent in a pipeline stage. Independently of the executed instruction, there is an upper bound on the time needed to pass a stage that can be determined from the concrete microarchitectural behavior. As soon as an instruction exceeds this bound, it is guaranteed to have advanced to the next stage.

Another possibility is to employ additional *relational* information. The example in Figure 6 shows, that progress cannot be guaranteed individually neither for the *add* nor the *load* instruction. However, we know that at least one of the two instructions makes progress in each cycle. Thus, an abstraction that tracks the progress of both instructions in a relational manner is eventually able to guarantee that both instructions have progressed to the next stage.

4.3 Diverging and Joining Control Flow

So far, we considered instruction sequences without branching and joining of control flow. The actual static analysis is performed on a control-flow graph with branches, control joins, and loops. A detailed and formal explanation of the consequences is out of the scope of this article, however we want to give a rough idea of how to extend the domain.

The problem with branching/joining is that one abstract state would need to talk about the behavior of different instruction sequences coming from/going to different branches of the control-flow graph. The obvious solution is to keep several abstract states – namely one per branch/different instruction sequence. After several abstract transformer cycles, the differing instructions will finally leave the pipeline and allow to join the remaining abstract states according to their minimal progress.

An efficient representation could be based on directed, acyclic graphs. Nodes in the graph are the instructions currently processed in the pipeline associated with their minimal progress. An edge points to the preceding instructions. In case of branch, several instructions have the same preceding instruction. In case of a control-flow join, the first common instruction has several preceding instructions and thus several outgoing edges.

4.4 Why Maximal Progress is not so Important

In this article, we focused on the minimal progress of instructions within a pipeline which is sufficient to derive an upper bound on the execution times of a program. In analogy to must-/may-cache analyses, an abstract domain tracking the *maximal* progress can be defined. This is needed e.g. for the calculations of lower bounds on the execution times of a program. Furthermore, in the case of non-monotone transformers it might be useful to prune some cases as infeasible.

5 Open Problems

So far, we examined in-order pipelines with separate caches which are well-suited candidates for compact abstractions due to their regular behavior. Modern processors, however, invest far more complexity to cleverly predict and optimize the executed instruction sequence. Their behavior is sensitive even to small local changes – leading to large global changes. This complicates the search for compact representations.

Branch Prediction and Speculation Speculation techniques allow to execute instructions although it is unclear whether they should be executed at all (e.g. due to an unresolved branch). Speculative execution is known to cause timing anomalies [6] and is also problematic from the point of view of guaranteed progress. Corresponding concrete transformers are non-monotone: Speculatively executed instructions that progress further can turn out to be detrimental for the overall progress. Besides direct effects such as unnecessary and expensive memory accesses (see [6] for an example), speculation can pollute the cache leading to indirect effects due to reloads later.

Buffers such as Store Buffers Additional buffers in the pipeline allow to further decouple the pipeline and the memory. As an example, stores complete to a store buffer such that the pipeline can continue execution while – in parallel

– the store is actually performed in memory. Such behavior introduces additional dependencies, e.g. instruction fetches, data loads, and stores compete for the exclusive bus resource.

Out-Of-Order Execution Data dependencies can hinder the execution of the current instruction in a program. Out-of-order execution allows to reorder instructions and thus to execute subsequent instructions whose dependencies are already satisfied. This complicates the dependencies of an instruction's progress – it might depend on the progress of instructions later in the program. Out-of-order execution is also known to cause timing anomalies [6].

6 Summary and Conclusion

We introduced design principles for pipelines with compact abstractions. We focus on an abstraction that is based on minimal progress of instructions through the pipeline. Any useful abstract transformer should guarantee some progress in each abstract transition. Otherwise, no execution-time bounds can be derived.

We showed that in-order pipelines are not automatically free of timing anomalies. Further, we found that monotonicity of the concrete transformer is sufficient for the absence of timing anomalies. Then, we defined strictly in-order pipelines and showed that these provide for monotone concrete transformers and thus for compact and effective abstractions.

References

[1] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In Alfred V. Aho, Stephen N. Zilles, and Barry K. Rosen, editors, *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 269–282. ACM Press, 1979.

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

[3] Gernot Gebhard. Timing anomalies reloaded. In Björn Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis, WCET*, volume 15 of *OASICS*, pages 1–10. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany, 2010.

[4] Marc Langenbach, Stephan Thesing, and Reinhold Heckmann. Pipeline modeling for timing analysis. In Manuel V. Hermenegildo and Germán Puebla, editors, *9th International Symposium on Static Analysis, SAS*, volume 2477 of *LNCS*, pages 294–309. Springer, 2002.

[5] Isaac Liu, Jan Reineke, and Edward A Lee. A PRET architecture supporting concurrent programs with composable timing properties. In *Conference*

Record of the Forty Fourth Asilomar Conference on Signals, Systems and Computers, pages 2111–2115. IEEE, 2010.

[6] Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In Frank Mueller, editor, *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, volume 4 of *OASICS*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[7] Christine Rochange and Pascal Sainrat. A time-predictable execution mode for superscalar pipelines with instruction prescheduling. In Nader Bagherzadeh, Mateo Valero, and Alex Ramírez, editors, *Proceedings of the Second Conference on Computing Frontiers*, pages 307–314. ACM, 2005.

[8] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 105–118, New York, NY, USA, 1999. ACM.

[9] Stephan Thesing. *Safe and precise WCET determination by abstract interpretation of pipeline models.* PhD thesis, Saarland University, 2005.

[10] Ingomar Wenzel, Raimund Kirner, Peter Puschner, and Bernhard Rieder. Principles of timing anomalies in superscalar processors. In *Fifth International Conference on Quality Software (QSIC 2005)*. IEEE, 2005.

[11] Stephan Wilhelm. Efficient analysis of pipeline models for WCET computation. In Reinhard Wilhelm, editor, *5th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, volume 1 of *OASICS*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.