A Declarative Compositional Timing Analysis for Multicores Using the Latency-Rate Abstraction

Vítor Rodrigues[2,3], Benny Akesson[4], Simão Melo de Sousa[1,3], Mário Florido[2,3]

[1] RELiablE And SEcure Computation Group
Universidade da Beira Interior, Covilhã, Portugal
[2] DCC-Faculty of Science, Universidade do Porto, Portugal
[3] LIACC, Universidade do Porto, Portugal
[4] CISTER-ISEP Research Centre, Polytechnic Institute of Porto, Portugal

Abstract. This paper presents a functional model for timing analysis by abstract interpretation, used for estimation of worst-case execution times (WCET) in multicore architectures using a denotational semantics. The objective aims at surpassing the intrinsic computational complexity of timing analysis of multiple processing units sharing common resources. For this purpose, we propose a novel application of *latency-rate* ($\mathcal{LR}$) servers, phrased in terms of abstract interpretation, to achieve timing compositionality on requests to shared resources. The soundness of the approach is proven with respect to a calculational fixpoint semantics for multicores that is able to express all possible ways in which a shared resource can be accessed. Experimental results show that the loss in precision introduced by the $\mathcal{LR}$ server model is about 10% on average and is fairly compensated by the gain in analysis time, which is above 99%. The system is implemented in Haskell, taking advantages of the declarative features of the language for a simpler and more robust specification of the underlying concepts.

1 Introduction

The timeliness requirement of a software application is defined by the capability of the underlying hardware running the application to assure that execution deadlines are met. In embedded real-time systems, the main timeliness criteria is the worst-case execution time (WCET) of an application [8]. The WCET depends both on the structure of source code, such as loop iterations and function calls, and on hardware factors, such as caches and processor pipelines. In general, the state space of both input data and hardware initial states is too large to be exhaustively explored by measurement approaches. This paper presents a pipeline analysis based on the theory of abstract interpretation [6], aiming at reducing this state space. The design of a proper abstract pipeline domain ensures that the analyzer stabilizes after a finite number of steps over Kleene sequences [10], while exhibiting a trade-off between the precision of the WCET results and the computational time required by the static analyzer.

When compared to single-core architectures, the complexity of the timing analysis in multicore environments does not depend only on the processor features, but also on the predictability of the timing behavior of each processor

when sharing resources, e.g. instruction and data memories [7]. In practice, this means that besides the control flow paths through the program, also the "architectural flows", i.e. the number of ways in which a shared resource can be accessed (also called *interleavings*), must be taken into account. Unless shared resources are shared in a composable manner, the different access interleavings allowed by the scheduling arbiter may produce different intermediate hardware states during analysis and, consequently, affect future timing behavior.

The complexity of the analysis increases exponentially when analyzing architectural flows. Suppose a program that consists of two concurrent processes, P_1 and P_2. The arising conflicts when requesting access to the shared resource are resolved by "interleaving" the execution sequences of the two processes in such a way that either P_1 or P_2 executes by flipping a coin. Hence, for a program with n processes, each one executing a sequence of m instructions, the number of possible interleavings is $(n.m)!/(m!)^n$. The numerator $(n.m)!$ gives all possible interleavings and the denominator $(m!)^n$ restricts this number to the number of allowed sequences, i.e interleavings that preserve the sequential order in the original machine programs. For realistic programs, the number of execution sequences are huge and although their analysis is a decidable problem, it is not feasible to compute in general.

The five technical contributions of this paper are:

1. a static timing analysis for multicore systems, using our previous two-level denotational meta-language [12,14] and an intermediate graph language;
2. the use of the *latency-rate* ($\mathcal{LR}$) server model presented in [17] as an abstraction to achieve compositionality in the temporal domain, so that the analysis of architectural flows can be avoided while preserving the soundness of timing analysis for multicore systems;
3. the formalization and implementation of the $\mathcal{LR}$-server model in the context of data-flow analysis using an abstract interpretation framework based on Galois connections;
4. a method for automatic compilation of dependency graphs, including the new "interleaving" graph operator, into a *meta*-language based on λ-calculus and directly implemented in Haskell;
5. showing that Haskell can be used as a language where the mathematical complex notions underlying the theory of abstract interpretation can be easily, elegantly, and efficiently implemented and applied to the analysis of complex hardware architectures.

The rest of this paper is organized as follows. We start by discussing the related work in pipeline analysis in Section 2, followed by an overview of our approach to the problem of WCET analysis for multicores in Section 3. Section 4 introduces the necessary background on the $\mathcal{LR}$ server model, in particular its ability to abstract timing behavior. Previous work on a two-level denotational meta-language used for static analysis based on abstract interpretation is described in Section 5 and a method to automatically compile fixpoint interpreters using the meta-language is described in Section 6. We then briefly describe our functional approach to a declarative pipeline analysis in Section 7. The formalization of the $\mathcal{LR}$ abstraction in terms of a Galois connection is given in Section 8, followed by a set of Haskell definitions for resource sharing in Section 9. We conclude after a discussion on experimental results in Section 10.

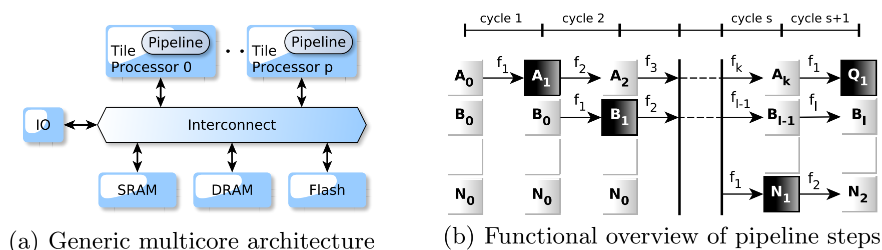
2 Related Work on Pipeline Analysis

The theoretical foundations of our complete WCET timing analysis framework are the methods of static analysis by abstract interpretation [6] combined with path analysis using linear programming [20]. For the particular case of pipeline analysis, we base our approach on the "abstract pipeline semantics" proposed by Schneider et al. [15], where provably sound timing properties are obtained by abstract interpretation. Since there is no abstraction of sets of *concrete* timing properties, the given pipeline analysis is a special case in the abstract interpretation framework where the abstract timing properties are themselves the "sticky collecting semantics" of concrete timing properties.

The concrete pipeline semantics in [15] is a simplified semantics of the processor, focused only on the aspects related to its temporal behavior, and relies on former value analysis and cache analysis. In contrast, our approach combines value and cache analysis with the pipeline analysis in a single data-flow analysis, which implies that the semantic transformers defined for the register and memory domains can be invoked during pipeline analysis. Still, the theoretical formalism given in [15], in particular its definition of *resource association*, can easily cope with our definition of "hybrid" pipeline state, i.e. a state that combines concrete timing properties with abstract cache states and register invariants.

3 Overview of Approach

We consider a tiled multicore architecture with several ARM9 cores, shared memories and IO, as shown in Figure 1(a). Each processor core has an instruction pipeline and an instruction cache memory. By definition, pipelining allows overlapped execution of instructions by dividing the execution of instructions into a sequence of pipeline stages, $k \in PS$, and by simultaneously processing N instructions. We consider a generic ARM9 processor with an instruction cache and a simple in-order pipeline with five pipeline stages: *fetch* (FI), *decode* (DI), *execute* (EX), *memory access* (MEM), and *write back* (WB). Figure 1(b) illustrates a functional view on pipelining.



(a) Generic multicore architecture (b) Functional overview of pipeline steps

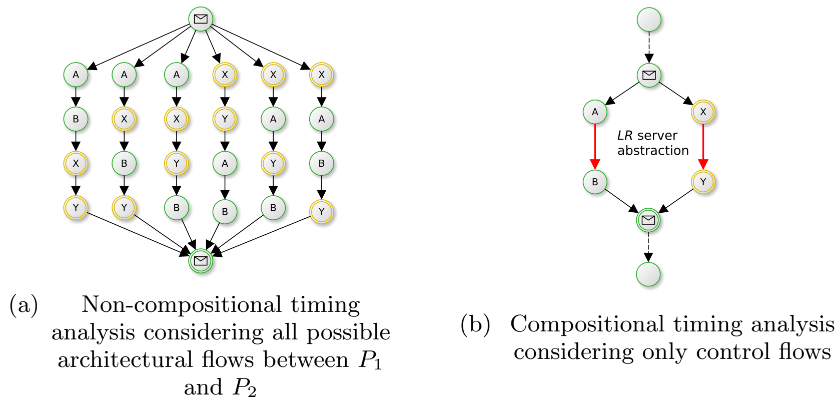
Fig. 1: Functional model of a pipeline in a multicore architecture

The functions $f_1, f_2, \ldots, f_k, \ldots,$ specify the effect of pipeline state transformations across a variable number of pipeline steps, which is greater than

five in the presence of pipeline *bubbles* [15]. For example, the instruction B in Figure 1(b) requires l pipeline steps to complete, where $l > k$. Each pipeline state includes an instruction vector of size N, adjoined with a timing property, $1, 2, \ldots, s, s+1$. This property expresses the relation between the elapsed *cycles per instruction* (CPI) and the current stage of an instruction inside the pipeline.

The absence of an abstraction for the *concrete* CPI values in the abstract interpretation literature [15] implies that the *abstract* pipeline domain must be defined as a *set* of pipeline states. For single-core architectures, this does not constitute a computational problem, because there is only a finite, and therefore manageable, number of pipeline states. Although the same principles could apply to timing analysis in multicore architectures, the major drawback is having the sets of concrete timing values spread across a huge number of architectural flows, which is exponentially bigger than the number of control flows.

Let P_1 and P_2 be two processes running on a homogeneous multicore system comprising two processor tiles. The corresponding number of architectural flows is given in Figure 2(a) and the original control flow is given in Figure 2(b).



(a) Non-compositional timing analysis considering all possible architectural flows between P_1 and P_2

(b) Compositional timing analysis considering only control flows

Fig. 2: Architectural and control flows for two processes P_1 and P_2, where instructions A and B belong to P_1 and instruction X and Y belong to P_2

Assuming composability in the value domain, i.e. there is no application data shared between processes, the need for timing analysis of architectural flows depends on the scheduling made by the arbiter of the shared resource. Composable arbiters, i.e. arbiters providing complete isolation between application in the temporal domain, analysis of interleavings is not required. An example of such an arbiter is non-work-conserving *time-division multiplexing* (TDM), which statically allocates a constant bandwidth to each processor core. However, when replacing the TDM arbiter by a work-conserving *round-robin* arbiter (RR), the system is no longer composable, since the scheduling of requests depend on the presence or absence of requests from other processor cores. In this case, the analysis of every allowed scheduled sequence in Figure 2(a) must be performed.

However, the analysis of the shared resource can be made compositional if the access times are predictable. This implies that upper bounds on the access times

to shared resources are calculated so that the variation in interference between processor cores visible in Figure 2(a) is removed (abstracted). The formal model of $\mathcal{LR}$ servers is particularly suitable for determining these upper bounds, since it provides a timing abstraction applicable to most predictable shared resources and arbiters. Figure 2(b) shows how the number of architectural flows can be reduced to the number of control flows when abstracting the temporal behavior by means of the compositional $\mathcal{LR}$-server model.

4 Latency-Rate Servers

We now introduce the concept of latency-rate ($\mathcal{LR}$) [17] servers as a shared-resource abstraction. In essence, a $\mathcal{LR}$ server guarantees a processor core a minimum allocated rate (bandwidth), ρ, after a maximum service latency (interference), Θ. As shown in Figure 3, the provided service is linear and guarantees bounds on the amount of data that can be transferred during any interval independently of the behavior of other processor cores. The values of the two parameters Θ and ρ depend on the choice of arbiter in the class of $\mathcal{LR}$ servers and its configuration. Examples of well-known arbiters in the class are TDM, *weighted round-robin* (WRR), *deficit round-robin* (DRR) and *credit-controlled static-priority* (CCSP) [1].

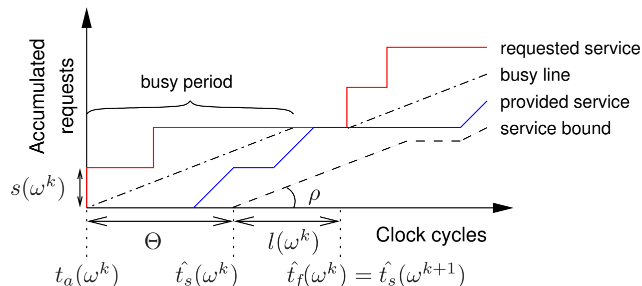


Fig. 3: A $\mathcal{LR}$ server and its associated concepts.

Like most other service guarantees, the $\mathcal{LR}$ service guarantee is conditional and only applies if the processor core produces enough requests to keep the server busy. This is captured by the concept of *busy periods*, which are intuitively understood as periods in which a processor core requests at least as much service as it has been allocated (ρ) on average. This is illustrated in Figure 3, where the processor core is busy when the requested service curve is above the dash-dotted reference line with slope ρ that we informally refer to as the *busy line*.

We proceed by showing how scheduling times and finishing times of requests are bounded using the $\mathcal{LR}$ server guarantee. From [19], the worst-case scheduling time, $\hat{t_s}$ of the kth request from a processor core, c, is expressed according to Equation (1), where $t_a(\omega^k)$ is the arrival time of the request and $\hat{t_f}(\omega^{k-1})$ is the worst-case finishing time of the previous request from processor core c. The

worst-case finishing time is then bounded by adding the time it takes to finish a scheduled request of size $s(\omega^k)$ at the allocated rate, ρ, of the processor core, which is called the *completion latency* and is defined as $l(\omega^k) = s(\omega^k)/\rho$. This is expressed in Equation (2) and is visualized for request ω^k in Figure 3.

$$\hat{t_s}(\omega^k) = \max(t_a(\omega^k) + \Theta, \hat{t_f}(\omega^{k-1})) \tag{1}$$

$$\hat{t_f}(\omega^k) = \hat{t_s}(\omega^k) + s(\omega^k)/\rho \tag{2}$$

5 Meta-Language

This section presents background on our denotational meta-language [12,14]. We develop a constructive fixpoint semantics based on expressions of a two-level denotational meta-language aiming at compositionality in both value and temporal domains. The main advantage is the possibility to generate type safe fixpoint interpreters automatically, and in a flexible way, for a variety of control flow patterns, including the architectural flows originated from shared resources.

Denotational definitions are factored in two stages, which is equivalent to the definition of a *core semantics* at compile-time (ct) and an *abstract interpretation* at run-time (rt). Supported by the *compositionality assumption* of Stoy [18], the core semantics expresses control and architectural flows by means of higher-order relational combinators of the run-time entities.

$$\mathrm{ct} \triangleq \mathrm{ct}_1 * \mathrm{ct}_2 \mid \mathrm{ct}_1 \mid\mid \mathrm{ct}_2 \mid \mathrm{ct}_1 \oplus \mathrm{ct}_2 \mid \mathrm{ct}_1 \oslash \mathrm{ct}_2 \mid \text{split rt} \mid \text{merge rt} \mid \mathrm{rt} \tag{3}$$

$$\mathrm{rt} \triangleq \underline{\Sigma} \mid (\underline{\Sigma} \underline{\times} \underline{\Sigma}) \mid \mathrm{rt}_1 \underline{\rightarrow} \mathrm{rt}_2 \tag{4}$$

Implemented combinatores are the sequential composition ($*$), the pseudo-parallel composition ($||$), the intra-procedural recursive composition ($\oplus$), and the inter-procedural recursive composition ($\oslash$). At the compile-time level, we can only directly talk about transformations on run-time values of type $\mathrm{rt}_1 \underline{\rightarrow} \mathrm{rt}_2$, defined for program states $\underline{\Sigma}$. These run-time types specify functions that can be regarded as state transformers or simply as "code", whose effect can be obtained by executing a piece of code on an appropriate abstract machine.

Therefore, interpretations of the higher-order expressions of the core semantics (ct) can be used to automatically generate the code of a program (designated by *meta*-program), which is *composed* by several state transformers (rt). The meta-programs are then given as input to the static analyzer. Let $b ::= (\cdot * \cdot) \mid (\cdot \mid\mid \cdot) \mid (\cdot \oplus \cdot) \mid (\cdot \oslash \cdot)$ be the syntactical meta-variable for the binary operators in the upper level of the meta-language, and $u ::= \mathsf{id} \mid \mathsf{split} \mid \mathsf{merge}$ be the syntactical meta-variable for the unary operators (interface adapters). Then, fixpoints can be generically defined as the reflexive transitive closure $T^\star$ of the transition relation T [4], where T is the initial program relation:

$$T^\star \triangleq \bigsqcup_{n \geqslant 0} T^n = \bigsqcup_{n \geqslant 0} \left(\bigsqcup_{i \leqslant n} T^i \right) = \bigsqcup_{n \geqslant 0} (\lambda R \bullet ((u\, T)\; b\; (u\, R)))^i (\bot_\Sigma) \tag{5}$$

where $\bot_\Sigma$ is the initial hardware state. In this way, fixpoint semantics can be efficiently computed by using program-specific *chaotic iteration strategies* [5], specified at compile-time level by the type expressions in the meta-language for free. In complement to type checking, the soundness of the abstract state transformers, which have the unified type $\mathrm{rt}_1 \underline{\rightarrow} \mathrm{rt}_2$ and are defined at run-time level, can be proven correct by using the calculational approach proposed in [4].

6 Automatic Generation of Fixpoint Interpreters

Next, we describe the calculation process of obtaining fixpoint interpreters. The generation of fixpoint interpreters is based on the notion of relational semantics of the program [3], defined as a set of transition relations $\tau \subseteq (\underline{\Sigma} \times \mathit{Instr} \times \underline{\Sigma})$, where *Instr* is the set of instructions of the program and $\underline{\Sigma}$ is the set of labeled program states according to a weak topological order (w.t.o) [2]. Moreover, the w.t.o. is used to induce a partial *dominance* order $\preceq$ over program instructions.

To represent all the program paths allowed for a program, an intermediate graph language was defined. The inductive abstract syntax of a dependency graph is represented by the data type G and allows us to represent a mimic of the execution order of a program [4], according the program structure known at compile time. The objective is to abstract the trace semantics [3] of the program into a set of "connected" transition relations τ, which are denoted in Haskell by **Rel** a, where a is a polymorphic variable for the domain $\underline{\Sigma}$.

A dependency graph is either an empty graph, a subgraph consisting of a single relation (**Leaf**), two subgraphs connected in sequence (**Seq**), two intra-procedural subgraphs connected recursively (**Unroll**), two inter-procedural subgraphs connected recursively (**Unfold**), two subgraphs connected pseudo-parallely (**Choice**), representing alternative program paths, or, last but not least, two subgraphs running on different processor cores (**Conc**).

```
data Rel a = (a, Instr, a)
data G a = Empty | Leaf (Rel a) | Seq (G a) (G a) | Unroll (G a) (G a)
         | Unfold (G a) (G a) | Choice (Rel a) (G a) (G a) | Conc (G a) (G a)
```

By taking advantage of the algebraic properties of the higher-order relational combinators, fixpoint interpreters (meta-programs) are "calculated" using the denotational approach. The syntactic phrases of a program are their dependency graphs. The denotations of each component of G are expressed by the combinators in the upper-level of the two-level denotational meta-language. The main advantage of using Haskell for the calculation of fixpoint interpreters is the fact that a definition written in Haskell can be compiled (or interpreted) to give a type-safe interpreter. This guarantees the correctness of a core semantics (ct) parameterized by the abstract state transformers defined at run-time (rt).

Along these lines, abstract interpreters in the form of Equation (5) are automatically compiled into λ-calculus by providing interpretations of the core semantics, in particular for the binary operators b and the unary operators u. For example, the sequential combinator $(*)$ is interpreted as the following:

```
( * ) :: (a → b) → (b → c) → (a → c)
(f * g) = λs → (g ∘ f) s
```

The main advantage of defining the higher-order relational combinators in Equation (3) is that new functions can be obtained throughout the composition of more basic functions. In this way, the calculation of meta-programs, all with the unified type $(a \rightarrow a)$, is defined by means of the function *derive*. For a complete definition with respect to the patterns in G, we refer to [13].

```
derive :: (a → a) → G a → (a → a)
derive f (Leaf r)  = f * abst r
derive f (Seq a b) = derive (derive f a) b
```

```
derive f (Conc a b) = let is = interleavings a b
                          ms = map (derive (create b)) is
                      in f ∗ scatter (length ms) ∗ (distribute ms) ∗ reduce
```

The function *abst* used in the interpretation of the atomic syntactic phrase **Leaf** provides the right-image isomorphism used in the abstraction of the relational semantics to denotation level, as described by Cousot in [3]. By the fact that the structure of dependency graphs is inductive, the type signature of *derive* requires the definition of the actual meta-program f, which is composed in sequence with new interpretations. In this way, the interpretation of (**Seq** a b) is straightforward, stating that the subgraphs a and b are connected in sequence.

The meaning of a subgraph **Conc** a b is given by the composition of the current meta-program f with the whole set of *interleavings* between a and b. The creation and synchronization of these two processes is modeled by the *scatter/reduce* computational pattern, commonly used in parallel computing. Inductively, the derivation of each individual trace is accomplished by using *derive* with the initial meta-program returned by the function *create* [13].

The function *interleavings* is used to obtain the set of architectural flows of Figure 2(a). This function takes two dependency subgraphs and returns a list of subgraphs. Using list comprehensions, the allowed sequences are a subset of all *permutations* of the transition relations belonging to both processes, *main* and *thread*, (which are first converted into lists using *toList*). The illegal sequences are removed by means of the constraint *preserve*, which excludes any sequence *seq*, that, after being filtered from the transition relations belonging to the other process, is not exactly equal to the original sequence *ori*.

```
interleavings :: G a → G a → [G a]
interleavings main thread
  = let preserve ori seq = ori ≡ filter ((flip elem) ori) seq
        (mainL, threadL) = (toList main, toList thread)
        sequences = [is | is ← permutations (mainL ++ threadL),
                              preserve mainL is, preserve threadL is]
        ts = map traces (groups sequences)
    in map (foldl interleave main) ts
```

After the computation of the interleaved sequences, it is necessary to transform these sequences back into dependency graphs. To this end, the functions *groups*, *traces* and *interleave* are defined according to the logics of the data type G, so that each architectural flow can be instantiated as set of connected transition relations, which possibly pertain to different applications.

In summary, the "derivation" of (**Conc** a b) is first to *scatter* the output state taken from the actual meta-program f into an "array" of independent flows, then *distribute* this state through the array of flows (*ms*), and finally combine the corresponding outputs using the function *reduce*. The functions *scatter*, *distribute* and *reduce* are described next.

```
scatter :: Int → a → [a]
scatter = replicate

distribute :: [a → a] → [a]     → [a]
distribute = zipWith (λf a → f a)

reduce :: (Lattice a) ⇒ [a] →   a
reduce = foldl join bottom
```

The function *scatter* is trivially defined by the Haskell function *replicate*. The function *distribute* takes a list of functions $[a \rightarrow a]$, and a list of input values $[a]$

and return a list $[a]$ with the results obtained by applying each input function to each input value. The function *reduce* is applied at merge points and is responsible for computing the least upper bound between the elements of the input list $[a]$, by using the functions *bottom* and *join* defined in the type class *Lattice* [13].

7 Pipeline Analysis

This section describes our functional and declarative approach to the pipeline analysis of ARM9. The pipeline analysis by abstract interpretation presented in [15] introduces the notion of *resource association* as a pair $(s, \{r_{j_1}, \ldots, r_{j_n}\})$, where $s \in PS$ is a pipeline stage and $r_{j_1}, \ldots, r_{j_n} \in R$ is a set of generic resources, such as functional units or cache memories. These resources can be either static, such as the resource demand of an instruction according to its type, or dynamic, when the description of the resource carries its own state. The particularity in our approach is that the state of the dynamically allocated sequences is updated after each pipeline stage. For this reason, we redefine the notion of a concrete pipeline state in [15] and introduce the notion of a hybrid pipeline state P, which combines concrete timing information with the abstract state of resources.

Let $R^\sharp$ be the abstract register domain, $D^\sharp$ be the abstract data memory domain and $M^\sharp$ be the abstract instruction memory domain. Since the states in $R^\sharp$, $D^\sharp$ and $M^\sharp$ can be updated during every pipeline stage and need to be shared by all instructions inside the pipeline, we require the definition of an extra set of store buffers $R'^\sharp$, $D'^\sharp$ and $M'^\sharp$. These domains contain the resource states that are to be allocated during the pipelining of every single instruction. This means that, after analyzing an instruction, it is required to compute the least upper bound between the top-level domains $R^\sharp$, $D^\sharp$ and $M^\sharp$ and the store buffers $R'^\sharp$, $D'^\sharp$ and $M'^\sharp$. The hybrid pipeline state is defined as:

$$P \triangleq (\mathit{Time} \times \mathit{Pc} \times \mathit{Demand} \times R'^\sharp \times D'^\sharp \times M'^\sharp \times \mathit{Coord}) \tag{6}$$

where *Time* is the global number of CPU cycles, *Pc* is *program counter* of the next instruction to fetch, *Demand* is a 32-bit sized word, used to model the dependencies between data registers in such a way that each register is either a blocked or unblocked resource, and *Coord* is a N-sized vector, N being the number of instructions allowed inside the pipeline at a given time.

$$\mathit{Coord} \triangleq [\mathit{TimedTask}]_N \tag{7}$$

A *TimedTask* is defined for one instruction and consists of the current elapsed CPU *Cycles* and the current *Stage* of a given *Task*. A *Task* is associated with an instruction, *Instr*, and holds also local copies of the "context" of a hybrid state:

$$\mathit{TimedTask} \triangleq (\mathit{Cycles} \times \mathit{Stage} \times \mathit{Task}) \tag{8}$$

$$\mathit{Task} \triangleq (\mathit{Instr} \times \mathit{Pc} \times \mathit{Demand} \times R'^\sharp \times D'^\sharp \times M'^\sharp) \tag{9}$$

We now identify semantic transformers required by our *functional approach* to pipeline analysis, as illustrated in Figure 1(b). The analysis is performed at three levels: at the lower level, we define the transformer F_T as a morphism on the composite domain *TimedTask* (for example, the instances $f_1, f_2, \ldots, f_n$ in Figure 1(b)); at the middle level, we define the transformer F_P as a morphism on the composite domain P, which uses F_T to compute the new elements inside

the N-sized vector *Coord*; finally, at the higher level, we define the transformer $F_P^\sharp$ as a morphism on sets of hybrid states $P^\sharp \triangleq 2^P$, which uses F_P to transform the hybrid pipeline states in the input set. The semantic transformers F_P and $F_P^\sharp$ are concisely defined as:

$$F_P \in \mathit{Instr} \mapsto P \mapsto P \tag{10}$$

$$F_P(i)(p) \triangleq \mathit{toContext}(i) \circ [F_T \circ \mathit{fromContext}(p)]_N \tag{11}$$

$$F_P^\sharp \in \mathit{Instr} \mapsto P^\sharp \mapsto P^\sharp \tag{12}$$

$$F_P^\sharp(i)(p^\sharp) \triangleq \{F_P^{5+}(i)(p) \mid p \in p^\sharp\} \tag{13}$$

where F_P^{5+} corresponds to the recursive functional application of F_P at least five times in an ARM9 pipeline. Note that F_P^{5+} does not correspond to the transitive closure of F_P by the fact that *local* worst-case timing properties are always associated with the final pipeline stage of a given task. This is possible because the value and cache analysis are performed simultaneously with the pipeline analysis, thus making the timing analysis a deterministic process for each given input timing property. In this way, the intermediate hybrid pipeline states can be discarded after completion.

However, even in fully timing compositional architectures [7], such as ARM9, the non-determinism introduced by the control flow must be taken into account. Therefore, the soundness can only guaranteed if all hybrid pipeline states arriving at a join point are collected into a set of type $P^\sharp$. The definition of $F_P^\sharp$ naturally supports the non-determinism intrinsic to sets of hybrid states in the sense that F_P^{5+} is applied to every pipeline state $p \in P^\sharp$. Let $\{s_k^i \mid k \in PS, k \geqslant 5\}$ be the set of ordered pipeline stages (including stalled stages) required to complete the instruction i. Then, F_P^{5+} is defined by:

$$F_P^{s_{k+1}^i}(i)(p) \triangleq F_P(i)(F_P^{s_k^i}(i)(p)) \tag{14}$$

$$F_P^{5+} \triangleq F_P^{s_{WB}^i} \tag{15}$$

The purpose of F_T is to compute the effect of pipelining a single instruction. However, since all the N instructions inside the coordinate vector (*Coord*) share the common context defined in P, it is necessary to read/write the state of the resources in P. In particular, the value of the program counter Pc must be known to fetch the next instruction from memory when one instruction inside the pipeline finishes, and the value of *Demand* must be kept updated depending on the blocked/unblocked state of register ports.

As an example, consider the case where the current stage is **FI** (*Fetch*), i.e. there is free space inside the pipeline to fetch a new instruction from instruction memory. Depending on the context of the actual pipeline state P, structural hazards [15] may block the access to memory and, therefore, cause the pipeline to stall. Otherwise, the actual *TimedTask* is updated by means of the function *fetchInstr*, which uses the context information about the next program counter to fetch *pc* and the actual state of the abstract instruction memory *iMem* to calculate the output timing property. Here, a timing property is denoted by the type variable *a* implementing the type class *Cycles*, as defined by Equation (8).

```
fetchInstr :: (Cycles a) ⇒ a → Task → TimedTask a
fetchInstr cycles t@Task {taskNextPc = pc, taskImem = iMem}
```

```
= let (classification, opcode, m') = getAbstMem iMem pc
      i = decode opcode
      pc' = pc + 4
      buffer' = setAbstReg bottom R15 (StdVal pc')
  in if classification ≡ Hit
       then let t' = t {taskInstr = i, taskNextPc = pc', taskImem = m'}
            in TimedTask {property = fetched cycles, stage = DI,
                          task = Fetched t' buffer'}
       else let t' = t {taskInstr = i, taskNextPc = pc', taskImem = m'}
            in TimedTask {property = missed p, stage = FI,
                          task = Stalled Structural t' buffer'}
```

Two different scenarios can occur during a fetch: either the *opcode* of the instruction is contained in the instruction cache, in which case the memory access is classified as a **Hit** and the next stage is set to **DI** (*Decode*); or it must be fetched from instruction main memory, thus causing the pipeline to become **Stalled**. In any case, the abstract cache state is updated by means of function *getAbstMem*. The type class *Cycles a* defines two functions, *fetched* and *missed*, for each of the corresponding scenarios. If the instruction fetch was successful, then the abstract value of the register **R15** (the program counter register in ARM9) is updated in the store buffer using *setAbstReg*. Due to page limitations, the reader can find the complete Haskell definition of the pipeline analysis in [13] (see Section 9 for another example of the use of declarative programming).

8 The $\mathcal{LR}$-Server Model as a Galois Connection

The meaning of the access times to shared resources in the context of timing analysis is the range of its possible values, i.e. the interval from lower bounds to upper bounds. Due to the limited bandwidth of the shared bus, shared accesses introduce additional delays that stall the pipeline. Therefore, the soundness of the timing analysis requires the computation of upper bounds on delays. To cope with this, we redefine *TimedTask* as:

$$\mathit{TimedTask} \triangleq (\mathit{Cycles} \times \mathit{Delay} \times \mathit{Stage} \times \mathit{Task}) \tag{16}$$

As mentioned in Section 7, the pipeline abstract domain is defined as a set of hybrid pipeline states, each including a "concrete" timing property now given by *Cycles* plus *Delay*. The purpose of the $\mathcal{LR}$-server model is to reduce the number of joins and provide, at the same time, upper bounds for delays caused by shared requests. From the observation of Figure 2(a), it is clear that the number of join operations is proportional to the number of architectural flows. However, Figure 2(b) shows that when applying the $\mathcal{LR}$ model to compute *safe* upper bounds for the finishing times of shared requests, the number of joins is determined solely by the control flows of each process independently.

The soundness of the abstraction provided by the $\mathcal{LR}$-server model relies on the fact the all timing properties calculated throughout architectural flows are upper bounded by the finishing times calculated using the $\mathcal{LR}$ model. Here, the objective is to formalize this approximation by means of a Galois connection.

Let *Delay* be an upper semi-lattice equipped with a partial order $\leqslant$ on natural numbers $\mathbb{N}$, describing both concrete and abstract timing properties and let $\mathbb{D}$ be a set of timing properties. A Galois connection $\mathit{Delay}^\natural(\subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathit{Delay}^\sharp(\subseteq)$, where $\mathit{Delay}^\natural = \mathit{Delay}^\sharp = 2^{\mathbb{D}}$, is defined in terms of a *representation function*

$\beta : Delay \mapsto \mathbb{D}$ that maps a concrete value $p \in Delay$ to the best property describing it in $\mathbb{D}$. This property is the canonical extension of Equation (2) to sets. Given a subset $X \subseteq \mathbb{D}$ and an abstract property $p^\sharp \in Delay^\sharp$, the abstraction and concretization maps are defined by:

$$\alpha(X) = \bigcup\{\beta(x) \mid x \in X\} \tag{17}$$

$$\gamma(p^\sharp) = \{p \in P \mid \beta(p) \subseteq p^\sharp\} \tag{18}$$

Let w_c^k be the kth instruction to fetch from the shared memory when there is a cache miss in the processor core c. The best property $p^\sharp$ is the singleton set containing the smallest finishing time given by Equation (2) when applied to w_c^k. Therefore, the $\mathcal{LR}$ abstraction can be formally defined by the representation function β:

$$\beta(t_f(w_c^k)) = \{\max(t_a(w_c^k) + \Theta_c, \hat{t}_f(w_c^{k-1})) + s(w_c^k)/\rho_c\} = \{\hat{t}_f(w_c^k)\} \tag{19}$$

This formally shows that the predictability of $\mathcal{LR}$ servers can be used to abstract the meta-programs corresponding to architectural flows into meta-programs corresponding to control flows only. Since each access time is upper bounded by the $\mathcal{LR}$ server, we have by compositionality that the maximum local timing property given by Equation (17), that would be obtained by joining ($\bigcup$) all abstract pipeline states across the architectural flows in Figure 2(a), is exactly equal to the maximum local timing property when only the control flows are considered.

9 Haskell definitions for resource sharing

This section gives declarative definitions for the temporal behavior of TDM and $\mathcal{LR}$ arbiters. Let the type variable a, defined in the type class *Cycles a*, be instantiated by a concrete timing property denoted by the data type *WCET*.

```
data WCET = WCET {cycles :: Int, ta :: Int, core :: Int, tf :: Int, delay :: Int}
```

The analysis of a TDM arbiter is simplified due to its predictable and composable properties, which makes the delay of a request to a shared resource easily computed using the arrival time, *ta*, and the processor *core* identifier. As mentioned in Section 7, requests to the main instruction memory occur upon cache misses. Thus, the function *missed* belonging to the type class *Cycles* is:

```
missed w@WCET {cycles = c, ta, core}
  = let d = ta `mod` frame
        first = slots * core
        end = first + slots - 1
        ts = if first ⩽ d ∧ d ⩽ end then 0
             else if d < first then (first - d) else (frame - d + first)
    in w {cycles = c + round (ts + 1), tf = ta + ts + 1, delay = ts + 1}
```

The frame size of the TDM bus is given by the variable *frame*. Assuming slots are equally distributed among the processor cores and that they are consecutively allocated in the frame and a completion latency of 1 cycle, the delay time is *ts+1*, where *ts* uses the division remainder of the arrival time *ta* by *frame* in order to check for an allocated slot. If the *core* needs to wait for an allocated slot, the required number of cycles can be statically calculated [9].

Now consider a shared bus with an arbitration protocol that is predictable but not composable, such as work-conserving round robin. In this case, the timing behavior of each application is dependent on the applications running on other cores, which makes analysis of all architectural flows mandatory in order to achieve soundness. In this context, the advantage of the $\mathcal{LR}$-server abstraction is the possibility to guarantee bounds on the starting times and finishing times of the requests so that compositionality in the timing domain is achieved.

The $\mathcal{LR}$-server model requires a timing property to model the guaranteed service rate, which is the finishing time (tf) of the previous request on the same *core*. According to Equation (2), the function *missed* defines the timing behavior of a cache miss in terms of an arrival time, *ta*, and a previous finish time, *tf*. Accordingly, the function *missed* is:

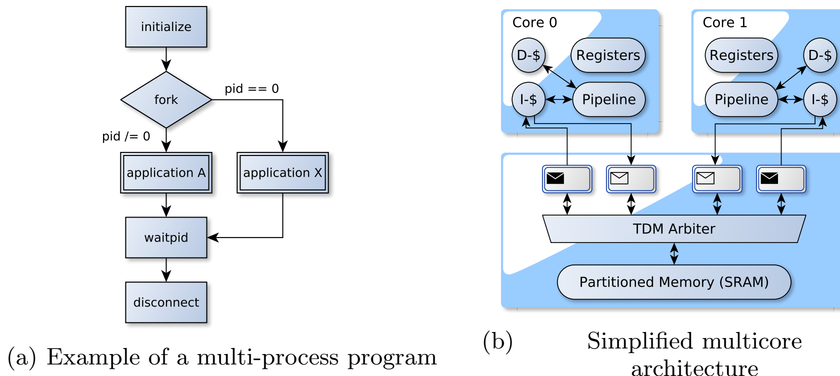
```
missed w@WCET {cycles = c, ta, tf = tf'}
  = let busy = ta + theta < tf'
        d = if busy then 1/rho else theta + (1/rho)
    in w {cycles = c + round d, tf = d + if busy then tf' else ta, delay = d}
```

10 Experimental Results

The discussion of experimental results include two different experimental scenarios. First, we compare the WCET and the analysis time obtained for small programs from the analysis of architectural flows (TDM) versus control flows (TDM) in Table 1. Second, we compare the WCET results of composable TDM versus a $\mathcal{LR}$ abstraction of a composable TDM arbiter for Mälardalen WCET benchmark programs [11] in Table 2. By compositionality of the $\mathcal{LR}$ abstraction and assuming that each processor core has a sufficiently large private data memory (D-$) and a common initial hardware state, each program is analyzed independently from the program configured to run on the second core. We consider the simplified multicore architecture in Figure 4(b), where instructions are shared in a partitioned SRAM memory shared by a TDM arbiter.

By definition, architectural flows cannot be feasibly computed. However, we do compute interleavings for the simple program in Figure 4(a), where "application A" and "application X" have only a few instructions each. Due to its natural composability, the analysis of control flows with TDM arbitration is much faster than the analysis of architectural flows, requiring only 1% of the time. With respect to the WCET estimate, the first line in Table 1 shows a lower WCET (179 CPU cycles) for the interleavings approach compared to composable TDM analysis (185 CPU cycles). This difference in the WCET is a consequence of the actual hardware state of the processor core running "application X" upon the invocation of the *fork* procedure and demonstrates the impact that the intermediate hardware states have on the timing analysis of architectural flows.

In fact, when the number of instructions of "application X" is bigger than the number of instructions of "application A", the worst-case path corresponds to that of "application X". However, since the analysis of "application X" starts with an empty pipeline state, it naturally takes less CPU cycles to complete. After increasing the number of instructions in "application A", this effect is eliminated because the worst-case path becomes that of "application A". Consequently, for the two analyses, the WCET is equal in the last two experiments.



(a) Example of a multi-process program (b) Simplified multicore architecture

Fig. 4: Simple program running on a simplified multicore architecture

Table 1: Comparison results for architectural flows, composable TDM

No. instructions "application A"	No. instructions "application X"	No. of interleavings	Results (CPU cycles/sec.)	Architectural Flows (TDM)	Composable TDM
4	5	126	WCET	179	185
			Analysis Time	57.0	0.17
5	5	252	WCET	188	188
			Analysis Time	140.3	0.18
6	5	462	WCET	195	195
			Analysis Time	588.7	0.43

Next, we compare the WCET results in Table 2 obtained using the $\mathcal{LR}$ abstraction with $\Theta = 1$ and $\rho = 0.5$ (modeling a particular TDM configuration with frame size of 2) to the results obtained with composable TDM. The WCET values presented in Table 2 depend not only on the size of the instruction cache and on the ability of the $\mathcal{LR}$ server to stay busy, but also on the program flow, e.g. number of loop iterations. Since we are considering a blocking multicore architecture, where a request from a processor core cannot be issued before the previous request has been served, every request starts a new busy period by definition. This is the most unfavorable situation possible for the $\mathcal{LR}$ abstraction, since every request requires $\Theta + 1/\rho$ cycles to complete, maximizing the overhead compared to TDM.

Still, our experiments show that this overhead is limited to between 8.7% and 12.1% for the considered arbiter, configuration, and applications. This is partly because the use of a small frame size reduces the penalty of starting a new busy period upon every cache miss through the low $\Theta = 1$ value, but also because the case of an SRAM shared by a TDM arbiter is quite simple and is captured well by the abstraction. A more complex case with DRAM and CCSP arbitration is shown in [16] along with an optimization to reduce the pessimism of the abstraction without loss of generality. In terms of the run-time of the analysis tool, it is approximately ($\approx$) the same for both composable TDM and the $\mathcal{LR}$ abstraction.

From this experiment, we conclude that compositional analysis of control flows using the $\mathcal{LR}$ abstraction is very fast and scalable compared to analysis of architectural flows. The analysis time is similar to compositional analysis based

on composable TDM arbitration, although it incurs a reduction in accuracy of about 8-12% for our configuration and applications. More precise WCET estimates would be obtained for multicore architectures that support high levels of parallelism. For example, architectures including super-scalar pipelines or caches allowing multiple outstanding requests. This would reduce the number of busy periods in the $\mathcal{LR}$ server upon cache misses, but would also increase the overall complexity of the WCET analyzer. Nevertheless, the main benefit of the $\mathcal{LR}$ abstraction is that it is able to perform compositional timing analysis using any arbiter belonging to the class, as opposed to being limited to composable TDM.

Table 2: WCET results for some of the Mälardalen benchmarks

Benchmark	No. Source Loop Iterations	$\mathcal{LR}$-server (WCET)	No. Cache Misses	TDM (WCET)	Overhead (%)	Analysis Time in sec. (≈)
bs	152	1162	111	1036	10.8	2.3
bsort	156	1459	152	1311	10.1	0.9
cnt	145	1309	175	1171	10.5	0.8
cover	111	796	105	707	11.2	3.9
crc	459	3160	304	2826	10.6	15.0
expint	251	2023	233	1818	10.1	1.9
fdct	1011	10897	720	9892	9.2	20.1
fibcall	111	994	59	885	11.0	2.3
matmult	287	2580	188	2343	9.2	5.2
minmax	221	956	263	873	8.7	2.6
prime	232	1079	196	959	11.1	5.2
ud	418	3943	97	3464	12.1	40.0

11 Final Remarks

This paper presents an approach to timing analysis in multicore architectures exclusively based on the declarative frameworks of denotational semantics, abstract interpretation and functional programming. The type system of Haskell is used to define a type safe and parameterizable fixpoint semantics by means of a two-level denotational meta-language. Fixpoint (abstract)-interpreters are automatically generated by providing interpretations to the algebraic combinators of the meta-language, providing a generic and compositional framework for static analysis. A particular abstract interpreter for pipeline analysis is defined for the WCET analysis of programs running on the ARM9 microprocessor.

The WCET analysis of multicores is defined incrementally by extending the intermediate representation language with a new syntactical element, representing programs running on different processing cores, whose denotational interpretation reuses the algebraic combinators used for static analysis in single cores. The complexity of the new fixpoint interpreter is reduced by using the abstraction provided by the $\mathcal{LR}$ server model on the timing behavior of shared resources.

Using declarative programming in Haskell, the temporal behavior of shared resources is in direct correspondence with the mathematical definitions of the TDM and $\mathcal{LR}$ arbiter models. The outcome is the definition of provably sound and compositional timing analysis in multicore environments, with a loss of precision in order of 10% on average that is relatively small compared to the factor 100 reduction in terms of analysis time.

Acknowledgments This work is partially funded by LIACC, through the Programa de Financiamento Plurianual, FCT, by the FAVAS project, PTDC/ EIA-CCO/105034/ 2008, FCT, and by the EU ARTEMIS JU funding, within the RECOMP project, ref. ARTEMIS/0202/2009, JU grant nr. 100202.

References

1. Benny Akesson, Andreas Hansson, and Kees Goossens. Composable resource sharing based on latency-rate servers. In *Proc. DSD*, 2009.
2. François Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proc. of the International Conference on Formal Methods in Programming and their Applications*. Springer-Verlag, 1993.
3. P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Elec. Notes in Theoretical Computer Science*, 6, 1997.
4. P. Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.
5. P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3), 1992.
6. Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. POPL*, 1977.
7. Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza, Jan Reineke, Benoît Triquet, and Reinhard Wilhelm. Predictability considerations in the design of multi-core embedded systems. In *Proc. ERTS*, 2010.
8. Jakob Engblom, Andreas Ermedahl, and Friedhelm Stappert. A worst-case execution-time analysis tool prototype for embedded real-time systems. In *Proc. RT-TOOLS*, 2001.
9. Timon Kelter, Heiko Falk, Peter Marwedel, Sudipta Chattopadhyay, and Abhik Roychoudhury. Bus-aware multicore wcet analysis through tdma offset bounds. In *Proc. ECRTS*, 2011.
10. Stephen Cole Kleene. *Introduction to metamathematics*. Van Nostrand, 1952.
11. Mälardalen WCET research group. `www.mrtc.mdh.se/projects/wcet`.
12. Vítor Rodrigues, João Pedro Pedroso, Mário Florido, and Simão Melo de Sousa. Certifying execution time. In *Proc. FOPARA*, 2012.
13. Vítor Rodrigues. A declarative compositional timing analysis for multicores using the latency-rate abstraction. Technical report, LIACC, Faculty of Computer Science, University of Porto, 2012. link: `www.dcc.fc.up.pt/~vitor.rodrigues`.
14. Vítor Rodrigues, Mário Florido, and Simão Melo de Sousa. A functional approach to worst-case execution time analysis. In *Proc. WFLP*, 2011.
15. Jörn Schneider and Christian Ferdinand. Pipeline behavior prediction for superscalar processors by abstract interpretation. *ACM SIGPLAN Not.*, 34, 1999.
16. Hardik Shah, Alois Knoll, and Benny Akesson. Bounding SDRAM Interference: Detailed Analysis vs. Latency-Rate Analysis. In *Proc. DATE (to appear)*, 2013.
17. Dimitrios Stiliadis and Anujan Varma. Latency-rate servers: a general model for analysis of traffic scheduling algorithms. *IEEE/ACM T. Netw.*, 6(5), 1998.
18. Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.
19. Maarten Wiggers, Marco Bekooij, and Gerard Smit. Modelling run-time arbitration by latency-rate servers in dataflow graphs. In *Proc. SCOPES*, 2007.
20. Reinhard Wilhelm. Why ai + ilp is good for wcet, but mc is not, nor ilp alone. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *LNCS*. Springer Berlin / Heidelberg, 2003.