Certifying Execution Time in Multicores

Vítor Rodrigues[a], Benny Akesson[b], Mário Florido[c], Simão Melo de Sousa[d], João Pedro Pedroso[e], Pedro Vasconcelos[c]

[a]*Rochester Institute of Technology, New York*
[b]*Faculty of Electrical Engineering, Czech Technical University, Prague*
[c]*DCC-Faculty of Sciences & LIACC, University of Porto*
[d]*DI-University of Beira Interior & LIACC, University of Porto*
[e]*DCC-Faculty of Sciences & INESC TEC, University of Porto*

Abstract

This article presents a semantics-based program verification framework for critical embedded real-time systems using the *worst-case execution time* (WCET) as the safety parameter. The verification algorithm is designed to run on devices with limited computational resources where efficient resource usage is a requirement. For this purpose, the framework of *abstract-carrying code* (ACC) is extended with an additional verification mechanism for *linear programming* (LP) by applying the certifying properties of *duality theory* to check the optimality of WCET estimates. Further, the WCET verification approach preserves feasibility and scalability when applied to multicore architectural models.

The certifying WCET algorithm is targeted to architectural models based on the ARM instruction set and is presented as a particular instantiation of a compositional data-flow framework supported on the theoretic foundations of *denotational semantics* and *abstract interpretation*. The data-flow framework has algebraic properties that provide algorithmic transformations to increase verification efficiency, mainly in terms of verification time. The WCET analysis/verification on multicore architectures applies the formalism of *latency-rate* ($\mathcal{LR}$) servers, and proves its correcteness in the context of abstract interpretation, in order to ease WCET estimation of programs sharing resources.

Keywords: abstract interpretation, verification, ACC, WCET, LP, $\mathcal{LR}$-servers

1. Introduction

The design of embedded real-time systems is, in general, guided by some *timeliness* criteria. Timeliness in embedded real-time systems means that programs have operational deadlines, i.e. strict run-time constraints, and that the system must guarantee such requirements to ensure *safety*. Timeliness evaluation is performed at hardware level and is defined as the system ability to assure that execution deadlines are met at all times. Therefore, when the risk of failure, in terms of system responsiveness, may endanger human life or substantial economic values [21], the determination of upper bounds for the execution times of programs becomes a safety requirement.

The timeliness safety criteria is most commonly specified by the *worst-case execution time* (WCET) [74]. This timing property is an over-approximation of the execution time of the path inside the program that takes the longest to execute. In general, the particular input data that causes the exact WCET is unknown at compile time. Therefore, the exclusive use of measurement-based techniques to determine the WCET for *any* possible run of the program is not feasible in terms of computational cost (exception made, for example, to straight-line programs, for which input data never changes execution order). Alternative solutions use *static analysis* methods to guarantee sound estimates of the WCET in finite time [73]. Nonetheless, *state of the art* WCET tool suites like `aiT` [1] and `Otawa` [11] are conservative by nature and may require manual intervention in order to predict tight WCET estimates. Also, end-to-end measurement-based approaches can be combined with game-theoretic learning approaches using SMT solvers in order to generate predictable WCET estimates based on probabilistic guarantees [63].

In addition, embedded real-time systems often require incremental updates, where critical "patches" may be required after deployment [6]. Traditionally, this is done using manual and heavyweight processes, specifically dedicated to a particular modification. However, incremental updates of real-time systems can only be achieved if the system design abandons its traditional monolithic and closed conception. We propose a novel approach to an ideal scenario, where the WCET analysis is complemented with a *verification mechanism*, whose task is to verify whether the computed WCET estimate is compliant with safety requirements of an embedded real-time system.

State of the art WCET analyzers like `aiT` [1] make use of the theoretical foundations of *abstract interpretation* [19] combined with *linear programming* [55]. Although such type of tool suites excel in computing tight and precise WCET estimates using real-world hardware models, they do not easily fit to the task of formal WCET verification. The reason is that, here, the focus needs to be put more on the search of highly efficient mechanisms for WCET *checking*. Although it is not our objective to reproduce the quality of state-of-the-art WCET analysis, we have designed a comprehensive WCET tool prototype[1], based on declarative programming, where the underlying concepts of abstract interpretation can be elegantly implemented [57]. The analysis is restricted to source-to-assembler code compiled for the ARM target architecture.

We perform a *flow-sensitive*, *path-sensitive* and *context-sensitive* timing analysis. Apart from the induction of abstract interpreters that perform *value* and *cache* static analysis based on state-of-the-art domains [57], in this article we focus on a new approach to *pipeline analysis* that can be parametrizable by the timing model of a generic processor. Sound upper bounds of execution time are computed as the combined result of a simultaneous *value*, *cache* and *pipeline*

[1]Available at `https://github.com/esmifro/wcetac`

analysis. Along the lines of [1], estimates of the *worst-case* execution time of the program are computed *a posteriori* by a *path analysis* using linear optimization.

Despite the common use of complex hardware features to increase instruction throughput, most embedded systems have limited computing resources. Therefore, mechanisms for WCET verification face new challenges due to the design complexity of WCET estimation. For this reason, the computational burden resulting from the integration of the complete WCET toolchain into the *trusted computing base* (TCB) of embedded systems with real-time constraints would be unacceptable. Well-known solutions that address this issue are Proof-Carrying Code (PCC) [50], Typed-Assembly Languages (TAL) [48] and Abstraction-Carrying Code (ACC) [9].

The main objective of this article is to present a verification mechanism that efficiently checks if a program satisfies a given safety specification in terms of WCET [60]. Along the lines of the previously mentioned approaches, we propose a lightweight and standalone method, which does not depend on a third-party certifying entity to monitor the transmission of one program through an "untrusted" communication channel. We extend the existent ACC framework with an efficient mechanism to check the solutions of the linear optimization problem. This mechanism is illustrated in Fig. 1 by the component "LP Checker".

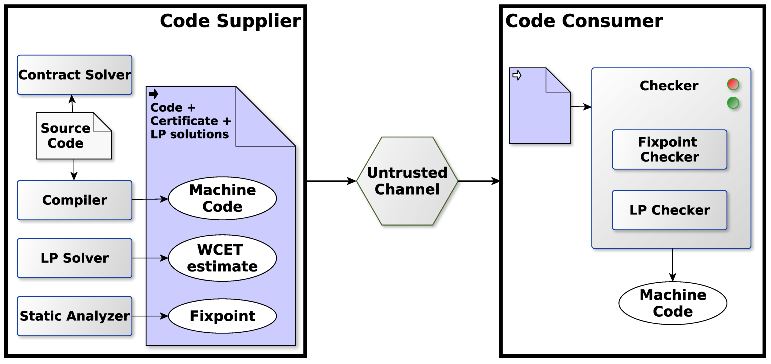


Figure 1: Overview of the extended ACC certifying platform

The verification mechanism uses *fixpoint theory* [40] to check the least fixpoint solution of the static analyzer, complemented with *duality theory* [47] applied to the *simplex method* [36]. The application of the verification mechanism to multicore architectures is based on the sound abstractions of the concrete timing model provided by the formal model of $\mathcal{LR}$-servers [67].

1.1. Abstract-Carrying Code

The Abstract-Carrying Code framework attributes the meaning of programs, $[\![P]\!]$, to least fixpoints in some abstract domain D_α, given the semantic operator S_P. The correctness of the analysis ensures that such an abstraction can be used as *certificates* for verification purposes: $Cert_\alpha = [\![P]\!] = \mathsf{lfp}(S_P^\alpha)$. These certificates contain abstract invariants that can be statically classified as *valid* by abstract interpretation in respect to some safety specification, I_α, such that $Cert_\alpha \subseteq I_\alpha$. The checking mechanism decides if the certificates are indeed fixpoints in a single pass: $(S_P^\alpha(Cert_\alpha) \equiv Cert_\alpha)$.

According to Fig. 1, the use of certificates in distributed embedded systems clearly separates the roles played by a code "supplier" and a code "consumer". The main advantage is that the computational cost associated with verification of safety properties is shifted to the supplier side, where the certificate is generated. To be effective, the *certificate checker* should be an automatic and stand-alone process and much more efficient than the *certificate generator*. Additionally, the size of the certificates is also relevant, since it determines if the verification process can be performed stand-alone and in reasonable time. Techniques to certificate size reduction use specific graph traversal strategies to classify entries in the certificates that are indeed *relevant* for the checker to reproduce the certificate in a single pass [7].

1.2. Fixpoint Semantics

The identification of infeasible program paths and upper bounds for loop iterations on the source code are typically referred to as *program flow* information. The accuracy of this particular type of information is essential to avoid WCET overestimation [28]. In order to automatically compute loop bounds and infeasible paths across fixpoint iterations, the fixpoint semantics proposed in this article is based on *chaotic iteration strategies* [15, 20] without the use of *convergence accelerators* [18, Section 2].

Iteration strategies provide a systematic way to recursively traverse the program syntactical structure until fixpoint stabilization is reached. As opposed to work-list algorithms, chaotic strategies follow the syntactic structure of the program, exactly computing fixpoints as a syntax-direct denotational semantics would do. The design of our *program-flow analysis* is based on an instrumented abstract domain that is used for value analysis and shares its results with the pipeline analysis, where the history of computation is particularly relevant. Hence, iteration strategies automatically provide flow-sensitivity, paving the way for automatic extraction of program flow information [57, Section 6.4].

A path-sensitive analysis is achieved by computing *backward abstract interpretations* of conditional branches [16]. The computation of necessary preconditions enables the static analyzer to combine the abstract domains used for value, cache and pipeline static analysis and to perform the latter within a single data-flow analysis, which can be regarded as a *simulation* in the combined abstract domain [57, Section 6.8].

However, the computational efficiency of our approach to WCET estimation is significantly less efficient when compared to the [1] toolchain, which comprises separate analyses for each abstract domain. For example, the analysis of loops in the context of cache behavior prediction has been greatly improved, notably by the VIVU approach [46]. However, our approach is able to reduce over-approximation of timing properties because abstract cache states are not propagated to the pipeline analysis. Moreover, our requirements to design a stand-alone verification mechanism enforce automatic extraction of program flow information. This implies full loop-unrolling and, naturally, loss of efficiency in the presence of programs with finite, but very high loop bounds.

Formally, the fixpoint algorithm is constructively defined as the *reflexive transitive closure* of a transfer function, which is regarded as an input-output relation [16]. Relations are associated with the syntactic phrases in the assembler program, each one defined between two labeled program states. In order to correlate the relational shape of our fixpoint algorithm, we apply the semantics projection mechanism proposed by Cousot [17] to derive a set of typed data-flow functions in the denotational setting. The isomorphism between the relational and the denotational semantic models is guaranteed by an *intermediate graph language*, which encodes the intensional information contained in the iteration strategy of a particular program.

1.3. Linear Programming

As already mentioned, the proposed WCET analyzer is based on static analysis and linear programming. However, the time necessary to compute the least fixpoint solution is significantly greater than the time necessary to calculate the solution of the linear program. The complexity of linear optimization when using *integer* linear programming (ILP) is NP-hard and cannot be reduced except if some properties specific to WCET analysis can be exploited, in particular, the reduction of the number of linear constraints imposing integral solutions. Otherwise, the inclusion of an ILP solver in the embedded system's TCB for the purpose of verification is likely to be impracticable by design.

We propose the inclusion of the WCET checking phase inside the ACC framework by conjecturing that the integer solution of the ILP program can be soundly found by solving a LP relaxation of the linear constraints. This is proved sound when the coefficient matrix of the integer linear problem is *totally unimodular* [51], allowing the use of *duality theory* to check the LP solutions. The mathematical properties of the *primal* and *dual* solutions of a LP program reduce the complexity of the "LP solver", to be solvable in polynomial time with the best known algorithms. Hence, for verification purposes, the "LP checker" is able to run in linear time and much more efficiently .

This article presents four technical contributions:

i/ A definition of a compositional and generic data-flow analyzer that uses a dependency order over assembler instructions. Efficient and type-safe fixpoint algorithms are automatically derived from an interpretation of an intermediate graph language that encodes the syntax of unstructured assembler code into expressions of a two-level denotational meta-language.

ii/ Inclusion of the WCET checking phase inside the ACC framework using the certifying properties of *duality theory* of LP. The verification of optimal solutions can be performed using simple linear algebra computations, without the need to recompute the simplex method algorithm.

iii/ A definition of a *pipeline analysis* that simultaneously combines a *value analysis* and a *cache analysis* in a single data-flow analysis by abstract interpretation. The number of pipeline states included in ACC certificates are largely reduced without affecting the soundness of the fixpoint checker.

iv/ A Novel approach to timing analysis in multicores by integrating the formal $\mathcal{LR}$-server model in the semantic framework of abstract interpretation. The resulting abstraction of the temporal behavior of shared resources provides a compositional timing analysis compatible with our denotational meta-language and preserves feasibility and scalability.

The rest of the article is organized with the following structure. We start with an overview of the relevant technical areas in Section 2. In Section 3, we describe the generic data-flow framework and how it can be parameterized to perform static analysis by abstract interpretation. A particular instantiation of this framework for the purpose of WCET analysis is presented in Section 4. An extension of the ACC framework for performing a semantics-based program verification using the WCET as the safety parameter is presented in Section 5. Before concluding, we present our WCET analysis in multicores architectures in detail in Section 6, where we also present the experimental results for a comprehensive subset of the Mälardalen benchmark programs.

2. Related Work

In this section, we introduce the technical areas at the foundations of our WCET verification framework. Denotational meta-languages are introduced first, followed by a summary of the related work on certifying algorithms. Then, we briefly present and contrast the design of modern WCET tools with our approach and, finally, we present the analytical model of $\mathcal{LR}$ servers.

2.1. Denotational Meta-Languages

The two-level denotational meta-language (TML) presented in [54] introduces an important level of modularity to denotational definitions that use typed λ-calculus as a meta-language by making the distinction between "compile-time" and "run-time" types. The two-level meta-language paved the way for systematic treatment of data-flow analysis and is important for efficient implementation of programming languages with the objective to automatically generate compilers. Code generation for various abstract machines can be specified by providing different run-time interpretations of the meta-language [52].

We propose a modified TML aiming to express the semantics of programming languages in a unified fixpoint form by means of algebraic relations. At the higher level of the meta-language, *meta-expressions* are defined to encode the dependency graphs of programs whose interpretation is always the same, regardless of knowledge about the domains of interpretation and syntax definitions. The correctness of denotational meta-programs depends on compile-time information, more precisely on their syntactical topological orders [15].

2.2. Abstraction-Carrying Code

The application of ACC to mobile code safety has been proposed by Albert *et al.* in [8] as an enabling technology for PCC, a first-order logic framework initially proposed by Necula in [50]. A partial specification check in PCC is performed using first-order predicates (preconditions and postconditions on safety properties) in order to verify if a given program P meets such a partial specification. The code supplier generates the set of *proof obligations* for P and discharges them using some verification tool that produces proofs. Afterwards, the program P is annotated into another program, P', with the set of proof obligations and their proofs, and sent to the code consumer. Upon reception, the verification condition generator produces new proof obligations and checks if they can be discharged using the proofs contained in P'.

A certified denotational abstract interpreter is presented in [13], where the main objective is to demonstrate that PCC-based techniques can be used to encode both the analyzer and the soundness proofs into the same logic, without any semantic gap between the analysis that is proved correct on paper and the analyzer that actually runs on the machine. Another relevant research project aiming at the certification of resource consumption in Java-enabled mobile devices is *Mobility, Ubiquity and Security* (MOBIUS) [12]. In this project, the *logic-based verification* paradigm of PCC is complemented with a *type-based verification*, whose certificates are derived from typing derivations or fixpoint solutions of abstract interpretations.

In the context of Constraint Logic Programming (C)LP, the verification conditions in ACC [8] are generated from a set of assertions in order to attest the compliance of a program with respect to the safety policy. If an automatic verifier is able to validate the verification conditions, then the fixpoint abstract semantics constitute the *certificate*. The consumer implements a defensive checking mechanism that not only checks the validity of the certificate w.r.t. the program, but also re-generates a set of trustworthy verification conditions to check the abstract properties contained in the certificate.

To improve the time efficiency of the verification mechanism, Albert *et al.* present a fixpoint technique to reduce the size of certificates in [7]. The key observation is that certain entries in a certificate may be irrelevant in the sense that they can be reproduced by the checker in a single pass. The program analysis graph is implicitly represented by means of two data structures, the *answer* table and the *dependency arc table*. Under some queue handling strategy, a table entry is relevant if after being "updated", it imposes the recomputation of other entries, i.e. if dependencies are found in the arc table. Hence, the reduced certificate can be proven valid, i.e. it is either the least fixpoint or a post-fixpoint, even if irrelevant entries are removed.

In the same direction, we propose a methodology that takes into account data-flow dependencies and actualize abstract states at the program labels that have their predecessors modified during the last iteration. As opposed to [7], dependencies between labels are captured by a chaotic iteration strategy. Instead of using an extra field to classify relevant entries, we use the "depth" of the program labels inside the topological order. Since iteration strategies are encoded in our intermediate graph language, we are able to perform algebraic transformation of these graphs by reducing the number of program labels and by removing labels whose *depth* is greater than "0" [15]. In the latter case, the meta-program extracted from such a dependency graph is a purely sequential algorithm that checks if the received certificate is, indeed, the *least* fixpoint point in a single pass. The checker in Fig. 1 is restricted to validate the least fixpoint because the safety policy implies formal verification of the optimality of the linear program solutions.

Even though incremental updates was identified as being a central concept in our application scenario, this article only presents the initial steps required to reach such level of flexibility in managing software updates. Notwithstanding, ACC investigates an incremental approach to PCC by considering any arbitrary program update over the original program [6]. This principle relies on the fact that certain parts of the original certificate may not be affected by the updates. Therefore, the optimized incremental checker not only rechecks those parts that have been directly affected by a change caused by the update, but also the indirect effect of these changes. This approach is not compatible with [7] because information required by the incremental checker may have been removed by the fixpoint reduction.

2.3. Certifying Optimization Methods

Certifying and checking algorithms using linear programming duality to provide a witness of optimality have been recently proposed by McConnell *et al.* [47]. Our framework complements this approach with a formal definition of the certifying algorithm for the WCET optimization problem that extracts a system of linear equations (*flow-conservation constraints*) from a *meta-trace semantics* using a right-image isomorphism on relations of labeled abstract states. This soundness mechanism is based on the constructive design of a hierarchy of program semantics presented in [17, Sec. 8]. In this way, we are able to relax the ILP problem of WCET calculation to an equivalent LP problem under the conjecture that the coefficient matrix of the linear program in *totally unimodular* [51].

Our conjecture that the linear program can discard the integral constraints without compromising precision is based on the fact that the coefficient matrix of the linear program is an $m \times n$ integral matrix such that the determinant of each square submatrix is equal to 0, 1, or -1. Since the infeasible paths and loop bounds are automatically detected, the relaxation of the ILP problem does not impact the (integer) type of the LP solutions. However, the total unimodularity conjecture excludes the class of LP programs where a given *capacity constraint* is capable of expressing loop-bound variables in terms of other loop-bound variables, e.g. for a triangular loop. In this sense, our checking mechanism is not complete with respect to all possible LP programs (consisting of flow-conservation constraints and capacity constraints).

Based on the property of total unimodularity, our automatic loop bound analysis supports nested loops in a way such that the execution counters at every loop header is a constant value. Therefore, the proposed WCET analyzer is a certifying algorithm that is both efficient and sound, for the reason that the maximum timing properties provided as input to the simplex solver are statically computed. The *checkability* of WCET estimates is guaranteed by the generic and automatic process of compiling chaotic fixpoint algorithms from dependency graphs combined with the applicability of the *strong duality theory* to the linear program. Hence, besides a highly efficient fixpoint checker in terms of verification time, the LP checker is designed to run in linear time.

A formally verified WCET estimation tool using the Coq proof assistant [27] is presented in [45]. The parallel with our outlined certifying method is the use of *a posteriori* validation to guarantee a correct ILP result. More specifically, the result of ILP solving is verified by stating that any larger solution (than the WCET) is infeasible. In order to prove the infeasibility of a linear system, easily checkable inclusion certificates based on Farkas lemma are used [14]. The checking is performed by a *validator* that is itself proven correct. The output of the validator is *true* if the certificate confirms that the ILP solution is valid. Compared to LP verification using duality theory, these machine-check proofs may incur non-negligible execution costs. In this aspect, we argue that our solution is more suitable for embedded real-time systems due to the limited computing resources of ACC consumer sites. Therefore, we require a trade-off between having sophisticated program-flow analysis and having a quite simple and efficient checking mechanism.

2.4. State-of-the-art WCET Analyzers

The prominent `aiT` WCET toolchain, performs a *microarchitectural analysis* of a machine program running on a given hardware platform by computing approximations using abstract interpretation of the times that the system takes to execute all allowed sequences of instructions. This requires abstract modeling of the host processor, which most commonly includes the abstract semantics of cache memory configurations [73] and the timing model of the processor pipeline [62]. These tasks are preceded by a value analysis, which depends on the abstract semantics of the processor instruction set and computes the range of abstract values for registers and memory addresses. Solutions for program-flow analysis include: 1) the VIVU approach [46], which transforms loops into procedures and use the call-string approach to interprocedural analysis; 2) an internal loop bound analysis using a combination of interval-based abstract interpretation and pattern matching [25]; and 3) a flow-facts annotation file format called AIS [30]. The estimation of WCET is done using ILP [72].

In contrast to the VIVU (virtual inlining, virtual unrolling) approach in [46], we perform full loop unrolling, rather than distinguishing the first loop iteration from the rest [46], and we apply the functional approach to inter-procedural analysis [57, Sec. 6.5]. However, many other techniques for program-flow analysis have been investigated in the literature. In fact, although flow analysis is undecidable in general, automatic extract of precise flow constraints can be achieve for a range of benchmark programs when using sophisticated abstractions [26, 34]. In particular, SWEET [42] is a tool that derives flow facts automatically using Abstract Execution [34]. This approach is highly flexible in relation to the abstract domains used for value analysis, including both non-relational and relational abstractions. The advantage of the latter is precision, in the sense that some relationships between program states are preserved by the abstraction. For example, the polyhedral abstraction [22] can be used to extract *parametric* flow facts from triangular loops [41].

Abstract Execution has similarities to our approach, but there are also fundamental differences arising from the necessity of "verification" of flow facts. Although Abstract Execution is based on abstract interpretation, it does not use traditional fixpoint analysis, where the abstract state for a program point covers all concrete states in that program point, for all executions. Since our approach uses fixpoint theory to verify the existence of a least fixpoint in the abstract domain, we chose to design our flow analysis based in Church numerals, where loop bounds have a well-defined semantics as a side effect of the value analysis. A common aspect of Abstract Execution and our approach is the explicitly use of time variables.

In [29] a variant of Abstract Execution is presented, instrumented with time variables holding upper and lower bounds for execution times, which eliminate the need for ILP calculation. However, this approach is not compatible with our verification scenario for one main reason: the WCET is a global variable that is incremented until the worklist-based algorithm of Abstract Execution terminates. To the best of our knowledge, such an algorithm has no certifying properties on the time variables themselves. Although our solution also considers collections of time variables, each timing property corresponds to *local* execution times on every program point. Therefore, under the conjecture of total unimodularity, both local execution times and flow facts can be cross-checked using duality theory applied to LP in a very efficient way.

The main drawback in our design is the efficiency of the fixpoint algorithm. However, our main goal is focused on the efficiency of fixpoint "verification" and not on efficiency of fixpoint computation. Nonetheless, our option to combine value, cache and pipeline analysis in a single data-flow analysis simplifies the run-time structure of the WCET analyzer by reducing the sources of non-determinism after determining *infeasible* program paths. In fact, we propose an extension of the abstract pipeline semantics proposed by Schneider et al. [62] in order to combine the intervals provided by the *value analysis* and the "execution facts" provided by the cache analysis in order to obtain reduce the number of program paths. This redesign allows us to analyze the WCET on multicore architecture models, because the access times to shared resources can be deterministically computed (assuming that concurrent programs do not share data, e.g. locks). By integrating the $\mathcal{LR}$ server abstraction into the pipeline analysis, the WCET in multicores is efficiently estimated by computing provably sound timing properties w.r.t. the concrete timing model of one single processor core.

An example of simulation in the abstract domain, where path analysis is integrated with accurate timing analysis is found in [43]. A common aspect with our approach is that automatic extraction of loop bounds during value analysis has high computational costs for loops that iterate many times. However, such a mechanism has the advantage of detecting infeasible paths and, consequently, improving the accuracy of the timing analysis. Alternatively, `GameTime` [63], explores loop unrolling (and inlined function calls) when maximum bounds are statically known. After extracting the control flow graph of the program, a subset of program paths, named *basis paths*, are extracted. The term *basis* is used in the sense of standard linear algebra, where basis paths are those that form a basis of all sets of paths. Satisfiability modulo theories (SMT) formulas are extracted for each candidate basis path. Besides detecting infeasible basis paths, the SMT solver generates test cases in order to perform end-to-end measurements. The `GameTime` algorithm inputs machine programs, executes these under the test cases and generates weighted graph models used to make predictions about timing properties. Sound WCET estimates are obtained even on architectures that include caches, complex pipelines and branch prediction.

Another common component in WCET tools is the existence of an intermediate representation of control flow. In comparison with our approach, the `aiT` toolchain is able to analyze and reconstruct control-flow graphs from disassembled code, whereas the analyzer presented in this article is limited to ARM assembler code. Another intermediate language used in static analysis are those of `Soot` [69], a framework for optimizing Java bytecode. Among several program analyses, a *dominance analysis* is especially useful for labeling program states. In contrast, our intermediate language has a simple inductive definition that expresses basic control-flow patterns and is able to encode dependency graphs consisting of ternary relations, where each relation is an instruction delimited by two program labels. More interesting in the context of our work is the control-flow representation presented in [5], where unstructured control flow of bytecode is transformed into recursion. Along these lines, we have designed a formal method for deriving fixpoint algorithms (abstract interpreters) that traverse the dependency (control-flow) graph.

2.5. Latency-Rate Servers

The $\mathcal{LR}$ server abstraction [67] is a simple linear lower bound on the service provided by a shared resource. The abstraction has two key benefits: 1) it enables resource interference to be bounded for the many arbiters belonging to the class [67], such as Weighted Round-Robin, Time-Division Multiplexing, and several varieties of Fair Queuing, thereby addressing the diversity of arbitration in complex systems in a unified manner; 2) it supports *independent* formal performance analysis per application, enabling WCET estimation to be extended to multicore platforms with shared resources in a scalable manner.

The model was originally developed for analysis of networks, but has gained popularity in the context of real-time embedded systems in recent years. Example uses of the model involve modeling buses [70], networks-on-chips [35], and SRAM and SDRAM memories [2, 64], both in the context of performance analysis using network calculus [23] and data-flow graphs [65]. The $\mathcal{LR}$ abstraction has also previously been used to compute WCET in single-path programs accessing shared resources [64]. However, this is the first time it has been integrated into a WCET static analyzer and proved correct in the context of abstract interpretation, enabling analysis of realistic programs sharing resources in a multicore setting.

3. Generic Data-flow Analysis

This section introduces a generic data-flow analysis [38, 53] used for computing parametrizable abstract interpretations on assembler code. The certifying property of the framework is based on the uniqueness of the least fixpoint solution of a system of data-flow equations [40]. The data-flow equations of a program are instantiated according to the data dependencies found in the program at compile-time, modeled according to the *weak topological order* of the program [15]. In this way, and in order to give a compositional characterization to the fixpoint semantics, the order of the data-flow equations inside a Kleene increasing chain is obtained by induction on the program syntactic structure.

3.1. Fixpoint Semantics

Topological orders, also named dominance orders in control-flow analysis [10], establish a dependency order among syntactic terms. The main difference is that the topological order considers program labels at the entry/exit points of every instruction in the assembler code, reducing a *basic block* to a single syntactic term. In this way, the dependency between data-flow equations can be expressed by a *dependency graph.* In this case, the notions of *weak topological order* and *dominance order* are equivalent and both can be applied to unstructured programs because the static information about relative offsets of "branch", "jump" or "call" instructions are preserved down to assembler level.

One advantage of expressing program semantics in this way is the ability to define a compositional data-flow analysis at the denotational level. The construction of a data-flow analyzer is achieved in two phases: 1) encode the label-based topology of the program into a dependency graph at compile-time; 2) automatically generate the code of an efficient fixpoint algorithm by an interpretation on terms of the dependency graph. To this end, the dependency graph is encoded using an *intermediate graph language* [57, Section 5.3].

Our approach to the computation of least fixpoints follows from the decomposition of fixpoint equations by partitioning as proposed in [20]. If such equations are continuous, the least fixpoint can be computed by Gauss-Seidel's iterative method to accelerate the convergence by continually re-injecting previous results in a specified order. As advocated in [16], the compositional design of an abstract interpreter should involve the choice of a chaotic iteration strategy to mimic the actual program execution, in the sense of denotational semantics, by induction on the program syntactic structure. In particular, our solution applies the *recursive* chaotic strategy presented in [15].

3.2. Abstract Program Semantics

For the purpose of timing analysis, automatic determination of execution times is accomplished by inspecting some assembler program, $P \in \text{Prog}$, consisting of a sequence of instructions, $I \in \text{Instr}$. Program properties are defined in the abstract domain $\mathbb{C}$, denoting all the possible abstract hardware states. The analysis is based on the notion of labeled *program states*, $\langle l, \sigma \rangle \in \Sigma$, that associate a *program label*, $l \in \text{Lab}$, with some *invariant*, $\sigma \in \text{Invs}$, that soundly approximates the dynamic behavior of a program $P \in \text{Prog}$ at that label [18].

$$\begin{aligned} \text{Instrs} &\in \text{Prog} \mapsto \wp(\text{Instr}) \\ \text{Instrs}[\![I_1; \ldots; I_n]\!] &\overset{\mathsf{def}}{=} \{I_1, \ldots, I_n\} \qquad (1) \\ \text{Invs} &\in \text{Prog} \mapsto \wp(\text{Lab} \hookrightarrow \mathbb{C}) \\ \text{Invs}[\![P]\!] &\overset{\mathsf{def}}{=} \text{at}_P[\![P]\!] \mapsto \mathbb{C} \qquad (2) \\ \Sigma &\in \text{Prog} \mapsto \wp(\text{Lab} \hookrightarrow \text{Invs}) \\ \Sigma[\![P]\!] &\overset{\mathsf{def}}{=} \text{at}_P[\![P]\!] \mapsto \text{Invs}[\![P]\!]r \qquad (3) \end{aligned}$$

The type of program invariants $\text{Invs}[\![P]\!]$ is the set of total maps from a program label, l, to an *abstract environment*, $\rho \in \mathbb{C}$. Let $I \in \text{Instrs}[\![P]\!]$ be a program instruction belonging to the program P. Every label l inside a program is contained inside the set in_P, and identifies either the state "at" the beginning or the state just "after" the instruction I.

$$\begin{aligned} \text{in}_P &\in \text{Instrs} \mapsto \wp(\text{Lab}) \\ \text{in}_P[\![I]\!] &\overset{\mathsf{def}}{=} \{\text{at}_P[\![I]\!], \text{after}_P[\![I]\!]\} \qquad (4) \\ \text{in}_P[\![I_1; \ldots; I_n]\!] &\overset{\mathsf{def}}{=} \bigcup_{i=1}^{n} \text{in}_P[\![I_i]\!] \qquad (5) \end{aligned}$$

The program states of Def. (3) are proved to be uniquely identified by their labels [18]. Hence, program states can be ordered according to the notion of weak topological ordering presented in [15], using the total order $\preceq$ on $\text{in}_P[\![\ldots]\!]$. For example, a possible order of a program with four components is given in (6). The elements inside matching parentheses are called *components* and the first element of a component is called the *head*.

$$(l_1^1 \ \cdots \ l_1^{n_1} \ (l_2^i \ \cdots \ l_2^{n_i} \ (l_3^j \ \cdots \ l_3^{n_j}) \ (l_2^u \ \cdots \ l_2^{n_u}))) \qquad (6)$$

The labels of component 1 are ordered by the sequence $l_1^1 \ \cdots \ l_1^{n_1}$, where the identifiers $1 \cdots n_1$ (upper indices), define a set of labels sharing a sequential hierarchy belonging to the same component (lower index). The second component has a first sequential order of labels, $l_2^i \ \cdots \ l_2^{n_i}$, starting with the identifier i and ending with n_i, then is interposed by a third component with a sequential hierarchy of $j \cdots n_j$, and is finally completed with the fourth sequential hierarchy of $u \cdots n_u$. The total order ($\preceq$) is induced by the position of each label.

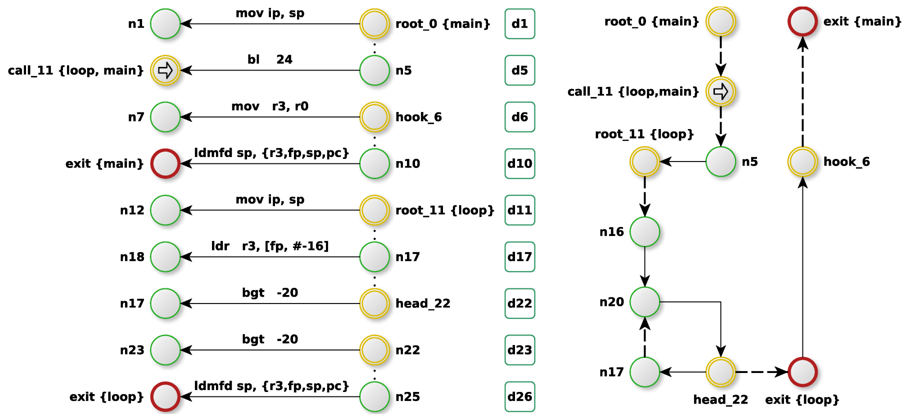
The first elements of topological components define the program points in the assembler from where executions can "branch", "jump", "return", etc. Considering the basic instruction-subset of the ARM platform, heads of components, like the ones upper-indexed by 1, i, j and u in Ex. (6), are necessarily either an entry point of a procedure (e.g. after a *branch-and-link* instruction, '`bl`'), the head of an intraprocedural loop (before a *conditional-branch* instruction, '`bgt`', '`beq`', etc.) or the hook point on the *caller* procedure after a procedure return (next instruction after a *branch-and-link*). The last labels inside a component, n_1, n_i, n_j and n_u in Ex. (6), represent either the label before an intraprocedural loop, the next-to-last label of an intraprocedural loop or the return point of a procedure (after a *load-registers-and-return* instruction, '`ldmfd`', '`ldmfa`', etc.).

Example 1. The weak topological order of a simple program.
As an illustrating example of program state labeling consider the source code in Fig. 2(a) and a fragment of the corresponding labeled assembler program in Fig. 2(b). This example is revisited throughout the rest of the article.

```
int main(void)
{
  int y = loop (3);
  return y;
}
```

```
int loop (int x)
{
  while (x>0)  { x--; }
  return x;
}
```

(a) Source code example



(b) Labeled relational semantics

(c) Meta-trace semantics

Figure 2: Source code example and the corresponding semantic representations

Fig. 2(b) shows two *root* labels, 'root_0' and 'root_11', one for each procedure 'main' and 'loop'. They label the states just before the instruction '`mov ip, sp`' which stores the intra-procedure pointer '`ip`' into the stack pointer '`sp`'. The procedure call is made through the *branch-and-link* instruction '`bl`', which starts with the *intra*-procedural sequential label 'n_5' with the target label 'call_11'.

The start label of the next instruction is a "hook" point ('hook_6'), stating that the execution continues at this point after the return of the procedure call. The intra-procedural loop inside the procedure 'loop' produces the "head" (underlined) label 'head_22' that, in conjunction with the label 'n_17' (feedback arc), states that the instructions between the labels 17 and 22 will be recursively executed by the static analyzer until the conditional instruction `bgt -20` evaluates to "false" in the abstract domain. Finally, each procedure returns with a non-enumerable target label "exit" that can only be determined in function of the caller's "hook" point.

The corresponding weak topological order given by (7) and the corresponding *recursive* iteration strategy by (8):

$$(0 \cdots 5\ (11\ 12 \cdots 16\ 20\ \ldots (\underline{22}\ 17\ \cdots 21)\ 22 \cdots 25\ (6 \cdots 10))) \tag{7}$$
$$0 \cdots 5\ 11 \cdots 16\ 20\ 21\ [\underline{22}\ 17\ \cdots 21]^*\ 22 \cdots 25\ 6 \cdots 10 \tag{8}$$

Label 22 is repeated after the $[\,]^*$ "iterate until stabilization" operator in order to provide a path-sensitive analysis. Note that compared to the weak topological order of (6), label 22 is denoted by both indices j and u. In this way, information dependent on the predicates at conditional branch instructions is taken into consideration. For instance, if a branch instruction represents a condition $x > 0$ in the source code, then it would assume that indeed $x > 0$ holds at the beginning of the target path of the branch (the head of the conditional component), and that $x <= 0$ holds at the fall-through branch. ▲

3.3. The Chaotic Fixpoint Algorithm

Our objective in applying chaotic strategies is to perform a context-sensitive analysis of assembler programs, where the history of a computation needs to be accounted for. This is particularly relevant for *pipeline analysis*, for which the chaotic fixpoint algorithm naturally provides pipeline state traversal. Put simply, the fixpoint algorithm is able to mimic the execution order of the assembly program by simulating the value of the *program counter*.

Fig. 2(b) illustrates how the assembly program is labeled according to a weak topological order where every instruction is delimited by two labels. Intuitively, the representation in Fig. 2(b) uses a total order, $(\preceq)$, on program labels to convey a relational semantics defined by a nondeterministic transition system $\langle \Sigma[\![P]\!], \tau\rangle$, where ternary relations $\tau \subseteq (\Sigma[\![P]\!] \times \mathrm{Instrs}[\![P]\!] \times \Sigma[\![P]\!])$ are defined between program states $\Sigma[\![P]\!]$ that are "connected" by the labeling process. Given a syntactic object $\mathrm{Instrs}[\![P]\!]$, an input-output relation is established between a state $\langle l, \sigma\rangle \in \Sigma[\![P]\!]$ (which we abbreviate as $\sigma_l \in \Sigma[\![P]\!]$ for sake of simplicity) and its possible successors in relation to l.

Let G be any rooted dependency graph. G is denoted by a triple (N, A, r), where N is the set of its nodes, A the set of arcs and r its root. A *path* p in G is said to lead from n_1 to n_k if a sequence of nodes in $(n_1, \ldots, n_i, n_j, \ldots, n_k)$ exists in N such that $\forall (i, j), \exists n_i \to n_j \in A$. Therefore, for each procedure there exists an isomorphism between the dependency graph G and the labeled relational semantics $\langle \Sigma[\![P]\!], \tau\rangle$, where the set of all nodes, $\{r\} \cup N$, corresponds to the labels of program states $\Sigma[\![P]\!]$ and the arcs $A = (N \times N)$ correspond to the pairs of label identifiers present in the set of relations τ.

The data-flow analysis framework is defined to be a pair $\langle \Sigma[\![P]\!], F\rangle$, where F is a space of functions acting in $\Sigma[\![P]\!]$. To each arc (i, j) of G, a data-flow function $f_{(i,j)} \in F$ is associated to represent the effect only at the component of $\Sigma[\![P]\!]$ at label j (for simplicity simply referred to as $\Sigma[j]$), that is, a change of relevant data inside the invariants map, $\mathrm{Invs}[\![P]\!]$, as control passes from the start i, through i, to the start of j. The set of data-flow equations, where for each $n_j \in N$, x_j denotes the data instance available at the start of n_j, and $(\perp_{\mathbb{C}})$

denotes the absence of information at the program entry, is defined as:

$$x_j = \bigsqcup_{(i,j)\in E} f_{(i,j)}(x_i), n_j \in N - \{r\} \tag{9}$$

The solution to these equations is the *merge-over-all-paths* solution (MOP):

$$y_i = \bigsqcup \{f_p(\perp) : p \in \text{path}_G(r, j)\}, j \in N$$

where we define $f_p = f_{(i_k,j_k)} \circ f_{(i_{k-1},j_{k-1})} \circ \cdots \circ f_{(i_1,j_1)}$ for each path $p = (n_{i_1}, n_{j_1} \ldots, n_{i_k}, n_{j_k})$. Since the MOP solution is undecidable in general, an approximating iterative algorithm is required to yield the *least fixpoint* solution. This accomplished by computing joins at those program points that have multiple incoming arcs. The denotational meta-language presented in Section 3.4 defines specific functions for this purpose, namely "split" and "merge".

The finer grain of observation of program execution, i.e. the most precise program semantics, is that of a *trace semantics* [17]. Starting from an initial state, the trace semantics models the execution of a program as a sequence of states, observed at discrete intervals of time, moving from one state to the next by executing an atomic program step. However, since program states $\Sigma[\![P]\!]$ differ from one another in their labels [18], the number of abstract invariants computed across fixpoint iterations is kept equal to the number of program labels. Therefore, the relational semantics of Fig. 2(b) abstracts from all the intermediate states computed during program execution. Further, the iteration strategy allows us to derive a more compact program semantics that we named *meta-trace semantics* [57], exemplified in Fig. 2(c).

The meta-trace semantics is convenient because different aspects of fixpoint semantics are accommodated in a simple way. The advantages are enumerated as: 1) the ordering in Ex. (1) is in direct correspondence with the order of evaluation of program states in Fig. 2(c); 2) weak topological orders have a graph-based representation, whose interpretation by means of functional application and joins computes approximations to the MOP fixpoint solution; and 3) the application of Gauss-Seidel's successive approximations [20] provide a constructive method to compute the least fixpoint of Kleene ascending chains.

At each step, the weak topological order determines which are the indices j of Equation (9) that are updated with the effect produced by what data-flow functions and in what order. In order to solve fixpoint equations like $\Sigma[\![P]\!] = F(\Sigma[\![P]\!])$, the data-flow equations defined in (9) are redefined in terms of an iteration k, where the invariant $\sigma_i = \Sigma[i]$ and the invariant $\sigma_j = \Sigma[j]$ are two invariants such that either $i \prec j$ or $j \preceq i$:

$$\sigma_j^{k+1} = \sigma_j^k \sqcup f_{(i,j)}(\sigma_i^k)$$

Fixpoint computations apply the *recursive strategy* to the iterations k over F. This strategy recursively stabilizes subcomponents of every component in the topological order every time the component is stabilized. For every node $\{n_i, n_j, \ldots\} \in N$ of *depth* 0, i.e. nodes that do not belong to any nested component, we know that for every arc $i \to j$, i is necessarily listed before j, i.e.

$i \prec j$. Hence, the value of i used in the computation of j already has its final value, which implies that the interpretation of sequential statements is made only once and in the right order, yielding a *flow-sensitive* data-flow analysis.

In the case of loops, including nested loops with increasing depth [15, Definition 1], the stabilization of the corresponding component is detected by the stabilization on its *head*. If the value associated with the head of the component remains unchanged after the subsequent iteration, then the argument used to prove the fact that no iteration is necessary over the nodes of depth 0 also proves that the values assigned within the component will not change hereafter [15].

Given the function space $F = \langle f_{(1,h)}, \ldots, f_{(i,j)}, \ldots, f_{(k,n)} \rangle$ and the invariants vector $\langle \sigma_1, \sigma_2, \ldots, \sigma_n \rangle$,, where n is the number of assembler instructions, Kleene first recursion theorem [44] is applied to devise a constructive method to compute the least fixpoint of F, $\mathit{lfp}\, F$. The least fixpoint condition is achieved when, $\forall (i,j) \in E$, the following equality test holds:

$$F^{\delta}(\langle \sigma_1, \ldots, \sigma_{j-1}, \sigma_j, \ldots, \sigma_n \rangle) = \langle \sigma_1, \ldots, \sigma_{j-1}, f_{(i,j)}(\sigma_i), \ldots, \sigma_n \rangle$$

Then, starting with the bottom element in iteration 0 by defining $\Sigma[\![P]\!]^0 = \bot_\Sigma$, the $\mathit{lfp}^{\sqsubseteq}_{\bot_\Sigma} F = \bigsqcup_{\delta \geqslant 0} F^{\delta}$ is computed by the upward abstract iteration sequence:

$$\Sigma[\![P]\!]^{k+1} = \begin{cases} \Sigma[\![P]\!]^k & \text{if } \Sigma[\![P]\!]^k = F(\Sigma[\![P]\!]^k) \\ \sigma_j^{k+1} = \sigma_j^k \sqcup f_{(i,j)}(\sigma_i^k) & \text{depth of } j \neq 0 \\ \sigma_j^{k+1} = f_{(i,j)}(\sigma_i^k) & \text{otherwise} \end{cases}$$

3.4. Meta-Language

We develop a constructive fixpoint semantics based on expressions of a two-level denotational meta-language [60, 59] 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 accesses to shared resources by application running on multicore architectures [58].

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 [68], the core semantics expresses control and architectural flows by means of higher-order relational combinators of the run-time entities. The advantage of this approach is that new programs can be obtained throughout the composition of smaller programs, in analogy to graph-based languages.

Implemented combinators are the sequential composition ($*$), the pseudo-parallel composition ($||$), the intra-procedural recursive composition ($\oplus$), and the inter-procedural recursive composition ($\oslash$). At the lower level, semantic transformers of type $\mathrm{rt}_1 \to \mathrm{rt}_2$ provide the desired denotational effects.

$$\mathrm{ct} ::= \mathbb{B} \mid \mathrm{ct}_1 * \mathrm{ct}_2 \mid \mathrm{ct}_1 \mid\mid \mathrm{ct}_2 \mid \mathrm{ct}_1 \oplus \mathrm{ct}_2 \mid \mathrm{ct}_1 \oslash \mathrm{ct}_2 \mid \text{split} \mid \text{merge} \mid \mathrm{rt} \qquad (10)$$

$$\mathrm{rt} ::= \Sigma \mid (\Sigma \times \Sigma) \mid \mathrm{rt}_1 \to \mathrm{rt}_2 \qquad (11)$$

The two-level meta-language unifies data and control flow in a pure functional language. Control flow is expressed at relational level by the upper level of the meta-language using the *point-free* notation [32]. Using this notation, the input state of a composition of data-flow functions, of type rt, is associated directly with the output state using left-associative parenthesis. In this way, references to the input argument can be removed from the compositional layer (*point-free*), allowing the compositional combinators to become binary relational operators by taking two relations as arguments and producing a new relation. For each combinator, there is only one bound variable that corresponds either to an input state with the type Σ or the product $(\Sigma \times \Sigma)$. We let $\mathbb{B}$ denote the logical boolean values (*True* and *False*), such that $\mathbb{B} \triangleq \{\text{tt}, \text{ff}\}$.

Example 2. Derivation of a meta-program using combinators.
Fig. 2(c) illustrates how a meta-program is derived for the assembler compiled from the source code in Fig. 2(a) and the corresponding iteration strategy (8). Starting with the first input-output relation defined for a syntactical expression, the first state-propagation function is instantiated for the arc given by the pair of label identifiers (0,1) that has the instruction '`mov ip, sp`' as the partially applied syntactic object. Then, the meta-program is automatically compiled by interpreting the syntax terms of the dependency graph that encodes the meta-trace in Fig. 2(c). The benefit of our approach is that only the control-flow combinators in the upper level of the denotational meta-language are used.

```
(mov ip, sp) * ··· * (mov r0, #5) * (bl 24) * ··· * (str r0, [fp, #-16]) * (b 16) *
··· * (ldr r3, [fp, #-16]) * (cmp r3, #0) * ((bgt -20) ⊕ (ldr r3, [fp, #-16]) * ··· *
(sub r3, r3, #1) * ··· * (ldr r3, [fp, #-16]) * (cmp r3, #0)) * (bgt -20) * ··· *
(mov r0, r3) * (ldmfd sp, r3,fp,sp,pc) * (mov r3, r0) * (str r3, [fp, #-16]) * ··· *
(ldmfd sp, r3,fp,sp,pc)
```

Figure 3: Meta-program derived from the meta-trace in Fig. 2(c)

For the '**while**' loop contained in the source code of Fig. 2(a), the inspection of the iteration strategy (8) detects a feedback arc between the label identifiers 22 and 17, expressed by the total order $22 \preceq 17$. This explicitly states that a recursive pattern was found in the control flow. Accordingly, the '**while**' meta-program uses the recursive operator $(\cdot \oplus \cdot)$ to compose the state-propagation function for the *branch* instruction '`bgt -20`', which is at the *head* of the loop, with a second meta-program containing the body of the loop. This meta-subprogram is the sequential composition $(*)$ of state-propagation functions for the set of instructions inside the loop, starting with '`ldr r3, [fp, #-16]`' and ending with '`cmp r3, #0`'. ▲

Data flow is defined at the lower level of the meta-language by means of state-propagation functions, which are extensions to program states of the data-flow functions of Def. (9). Next, we detail how instances of these state-propagation functions are obtained as abstractions of the relational semantics. As mentioned in Section 3.1, transition relations $\tau \subseteq (\Sigma[\![P]\!] \times \text{Instrs}[\![P]\!] \times \Sigma[\![P]\!])$ are ternary relations which, given a syntactic object $i \in \text{Instrs}[\![P]\!]$, establish an input-output relation between a state and its possible successors in the context of some program P. Assuming the abstract state vector $\Sigma[\![P]\!] = \langle \sigma_1, \sigma_h, \ldots, \sigma_i, \sigma_j, \ldots, \sigma_n \rangle$, where n is a label identifier, we use the relational abstraction defined in [17], where the right-image isomorphism f is applied to every transition relation τ:

$$f_{(i,j)}[\![\tau]\!] \overset{\text{def}}{=} \lambda\sigma_i \bullet (\sigma_j \mid \exists \Sigma[\![P]\!]^i, \Sigma[\![P]\!]^j : \sigma_i = \Sigma^i_P[i] \wedge \sigma_j = \Sigma^j_P[j], \\ \exists \iota \in \text{Instrs}[\![P]\!] : \langle \Sigma^i_P, \iota, \Sigma^j_P \rangle \in \tau) \tag{12}$$

Each function $f_{(i,j)}$ is partially applied to the syntactic object $\iota \in$ Instr, so that, at denotational level, we exclusively reason about functions with the type $(\mathbb{C} \to \mathbb{C})$, only by using the abstract values located at the labels i and j. When computing the least fixpoint solution, the least upper bound of the multiple states arriving at the program label j is computed. The type of a data-flow function f is lifted to the global state functional type $(\Sigma \to \Sigma)$ in order to obtain the space of state-propagation functions $F = \langle f_{(1,h)}, \ldots, f_{(i,j)}, \ldots, f_{(k,n)} \rangle$, where each $f_{(i,j)}$ is defined by (12) and n is the number of syntactic objects.

In practice, using the semantic projection mechanism defined in [17], *the fixpoint algorithm evaluates meta-programs at trace level by expanding the meta-trace according to the iteration strategy, but using the program's structural constructs defined at relational level and the program's functional behavior defined at denotational level.* Assuming that multiple incoming nodes n_k arriving to the node n_i may exist, but that only one outgoing node n_j leaves from n_i, the denotational fixpoint is defined as an abstraction of the relational semantics:

$$F(f_{(i,j)}) \overset{\text{def}}{=} \lambda f_{(i,j)} \bullet \lambda \Sigma^i_P \bullet (\Sigma^j_P[j] := f_{(i,j)}(\sigma_i) \mid \forall k : \sigma_k = \Sigma^k_P[k], \\ \exists i : \sigma_i = \bigsqcup f_{(k,i)}(\sigma_k) : \Sigma^k_P \, \tau \, \Sigma^i_P \wedge \Sigma^i_P \, \tau \, \Sigma^j_P) \tag{13}$$

If fact, the previous definition provides an approximation to the MOP fixed-point solution by providing a compositional and constructive iteration method based in functional application and least upper bound operators. Compared to the natural nondeterministic fixpoint semantics given in [17], two observations should be made: the first is that Def. (13) gives an abstract (approximate) fixpoint semantics of a nondeterministic system of input-output relations, whereas [17, Theorem 33] gives the *collecting semantics* of such a system; the second is that both definitions must be different because the state transformer used in the collection semantics is *additive*, i.e. a complete join morphism, where the abstract state transformer used in Def. (13) is continuous but not additive.

Let $b ::= (\cdot * \cdot) | (\cdot \parallel \cdot) | (\cdot \oplus \cdot) | (\cdot \oslash \cdot)$ be the syntactical meta-variable for the binary combinators in the upper level of the meta-language. Let also the interface adapters 'split' and 'merge' (least upper-bound operator) be represented by input-output relations. Then, the reflexive transitive closure $T^\star$ of the program's initial input-output relation T, where R is a bound relation, is defined in point-free style as a relational generalization of the Kleene fixpoint theorem:

$$T^\star = \bigsqcup_{n \geqslant 0} T^n = \bigsqcup_{n \geqslant 0} (\lambda R \bullet (T \, b \, R))^n (\bot_\Sigma) \tag{14}$$

where $\bot_\Sigma$ is the undefined abstract state. In this way, type-safe fixpoint algorithms can be efficiently obtained *for free* [59] by using program-specific chaotic iteration strategies, specified by the type expressions in the meta-language.

Example 3. Fragment of fixpoint iterations for the loop in Example 1. In order to better illustrate the semantics of our chaotic fixpoint algorithm, first consider the set of relations presented in Fig. 2(b). As previously mentioned, each state transformer $f_{(i,j)}$ is partially applied to the instruction in the correspondent input-output relation, whose delimits are precisely labels i and j. Now consider the iteration strategy given in (8). Since loops are transformed into recursion, the interesting labels are the head of the loop, $\underline{22}$, and entry point of the loop body identified by label 17. The meta-program in Fig. 3 shows that register `R3` is allocated to the variable "x" in program 2(a). As expected for this loop, the least fixpoint solution is $[0, 3]$ and the computed loop bound is 3.

Table 1 presents a summary of the fixpoint iterations for label 17, showing the result of value analysis and the resulting program flow analysis. For label 22, we need to consider that instruction '`b 16`' is inside a relation that points to label 20, which is inside the scope of the recursive combinator $(\cdot \oplus \cdot)$. For this reason, we consider a *previous* fixpoint iteration when presenting the summary of fixpoint iterations for label 22 in Table 2. Note that value analysis requires 4 fixpoint iterations. This fact results from the fact that instruction '`cmp r3, #0`' has to be interpreted in the abstract domain within an "extra" fixpoint iteration in order to compute sufficient pre-conditions to continue loop iteration (detected by instruction '`bgt -20`').

Table 1: Program flow analysis for label 17 (3 fixpoint iterations)

Label	Loop Iteration 1		Loop Iteration 2		Loop Iteration 3	
	Value Analysis	Flow Analysis	Value Analysis	Flow Analysis	Value Analysis	Flow Analysis
17	`R3` = [3,3]	(22,17) = 1	`R3` = [2,3]	(22,17) = 2	`R3` = [1,3]	(22,17) = 3
		(17,18) = 1		(17,18) = 2		(17, 18) = 3

Table 2: Program flow analysis for label 22 (4 fixpoint iterations)

Label	Prev. Fixpoint Iteration		Loop Iteration 3		Loop Iteration 4	
	Value Analysis	Flow Analysis	Value Analysis	Flow Analysis	Value Analysis	Flow Analysis
12	`R3` = [3,3]	(21,22) = 1	`R3` = [1,3]	(21,22) = 3	`R3` = [0,3]	(21,22) = 4
		(22,17) = 1		(22,17) = 3		(22,17) = 4

In fact, the execution counter at label 22 is 4, but the analysis shows that the execution counter at label 17 is 3. Furthermore, the chaotic strategy determines that the execution counter at label 23 is 1. This satisfies the flow-conservation constraints on the execution counts: the sum of execution counts for input arcs must equal the corresponding sum of the output arcs: 4 = 3 + 1. Therefore, the loop bound assigned to the head of the loop is 3. According to Fig 2(b), this is the value assigned to the arc identifier '`d22`' . ▲

4. The WCET Analyzer

Having defined our combinatory and program-specific fixpoint semantics, we now proceed to the definition of a particular abstract interpreter used for WCET estimation. The run-time type Σ is instantiated by the CPU hardware domain $\mathbb{C}$ and the corresponding semantic operator, $F_{\mathbb{C}} \in \mathbb{C} \mapsto \mathbb{C}$, preserves the functional type in Def. (11). Fixpoint algorithms correspond to a particular order of application of the higher-order combinators in Def. (10), which combine instances of $F_{\mathbb{C}}$ according to the topology of the program. The abstract domain $\mathbb{C}$ is a composite domain composed by an abstract register domain, $R^{\sharp}$, an abstract data memory domain $D^{\sharp}$, an abstract instruction memory domain, $M^{\sharp}$, and an abstract pipeline domain, $P^{\sharp}$. The abstract pipeline domain, $P^{\sharp} \triangleq 2^{P}$, is defined as a collection of "hybrid" pipeline states defined by P. As part of our ARM-based processor model, we next describe the 5-stage Harvard pipeline architecture used by the ARM9 processor in shown Fig. 4.

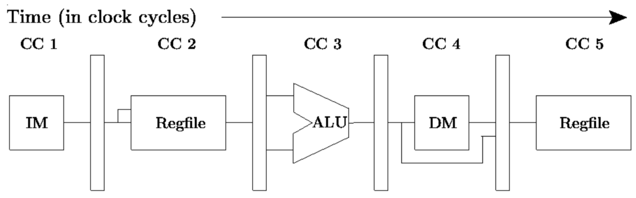


Figure 4: Graphical representation of a Harvard pipeline architecture

The first stage (*Instruction Fetch*) depends exclusively on the instruction memory ("IM") because this stage is when the memory address given by the program counter is "fetched" from the instruction memory. In the second stage (*Instruction Decode*), the values of the operands of the fetched instruction are loaded from the register file ("Regfile"). At this point, store buffers must be created to carry out the processing of the "decoded" instruction. In the third stage (*Execution*), the "ALU" functional unit computes the hardware state after "executing" the instruction using the buffered operands. In the fourth stage (*Memory*), the data memory ("DM") is accessed, if that is required to execute the instruction. Finally, the last stage (*WriteBack*) is where the store buffers are loaded back into the globally shared register file ("Regfile"). The number of elapsed clock cycles in each stage is shown as "CC 1", "CC 2", etc., and indicate the clock cycles required to process a single instruction.

The register domain $R^{\sharp}$ denotes the register file, referred to as Regfile in Fig. 4. We design the "abstract" environment of the register file as a map, $R_{\nu} \triangleq \mathbb{N}_{\mathbb{V}} \mapsto \mathbb{V}(\mathbb{W})$, from register identifiers $\mathbb{N}_{\mathbb{V}}$ to abstract values $\nu \in \mathbb{V}(\mathbb{W})$, where the domain $\mathbb{W}$ defines a 32-bit value and $\mathbb{V}(\mathbb{W})$ defines intervals of 32-bit values. However, in order to perform data-flow analysis as simulation in the abstract domain, the registers holding control information, such as the *stack pointer* or the *program counter*, must not be abstracted into the interval domain. Hence, $R^{\sharp}$ is defined for the entire set of register names $\mathbb{N}$ in the following way.

Let $\mathbb{N}_{\mathbb{W}} = \mathbb{N} \setminus \mathbb{N}_{\mathbb{V}}$ be the set of registers storing "concrete" control information and let $R_w \triangleq \mathbb{N}_{\mathbb{W}} \mapsto \mathbb{W}$. Then, the abstract domain $R^\sharp$ is a coalesced domain of both abstract and "concrete" values and is formally defined by the disjoint union of the corresponding maps, $R^\sharp_\nu$ and R_w: $R^\sharp \triangleq R^\sharp_\nu \biguplus R_w$.

Automatic determination of loop bounds is achieved by instrumenting the abstract domain $\mathbb{V}(\mathbb{W})$ with the concrete domain of naturals $\mathbb{N}$. An example of such instrumented value analysis is given in [57, Section 6.4]. To this end, we invoke the relationship between the constructive method of the Kleene first-recursion theorem [40], used to compute least fixpoints, with Church numerals [61, Chapter 11]. Intuitively, given a continuous function φ, loop bounds are natural numbers that are mapped to the n-fold composition of φ. In this context, a loop bound is a natural number called the *fixed-point value* of φ.

When compared to a real ARM9 processor, the design of abstract memory domains is relatively simpler because the data cache is not included in the analysis. Our approach to cache analysis is pragmatic in the sense that our objective is not to propose new abstract domains for cache analysis, but reuse existent designs [31]. Moreover, we assume that even though RR or FIFO replacement policies are typically used by ARM9 processors, the used cache replacement policy is Least Recently Used (LRU) [56]. Technically, there are also relevant simplifications, since only a *must analysis* [56] needs to be implemented.

Our memory model consists of two different main memory domains: the *data memory*, $D^\sharp$, and the *instruction memory*, $M^\sharp$. A cache is included only inside the instruction memory hierarchy, whereas all data memory accesses are directly made to the main memory. Accesses to the instruction cache are requested by the pipeline during the "Fetch" stage. If the requested program counter memory address is contained in the cache, corresponding to a *cache hit*, then the opcode is returned with low latency. Conversely, upon a *cache miss*, the opcode needs to be transferred from the main memory to the cache, replacing existent cached data. For LRU, a must analysis is sufficient to obtain precise results about cache hits. Hence, we are able to deterministically infer the penalty associated to cache misses. This is an essential feature of our design because we are able to integrate the analytical model of $\mathcal{LR}$ servers to predict access time to a shared main memory. Section 6 explains how the $\mathcal{LR}$ is soundly integrated into our data-flow analysis by abstract interpretation using a proper Galois connection.

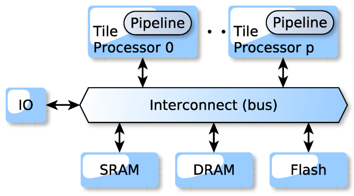
By design, the value analysis depends on the domains $R^\sharp$ and $D^\sharp$ and the cache analysis depends on the domain $M^\sharp$. Therefore, fixpoint iterations of the pipeline analysis are defined in terms of a sequence of "hybrid" pipeline states, P, each containing three corresponding store buffers, $R'^\sharp$, $D'^\sharp$ and $M'^\sharp$. Pipeline analysis by abstract interpretation is not a typical analysis because it requires both state traversal, as determined by a control-flow graph, and the computation of least upper bounds on abstract domains [73]. In this sense, the pipeline analysis can rather be seen as a "history-sensitive" analysis that requires the *collection* of all states encountered during fixpoint computation.

Since there is no abstraction known to the WCET community for concrete timing properties [62], the abstract pipeline domain must be defined as a set of "hybrid" pipeline states. Therefore, and according to the pipeline model of

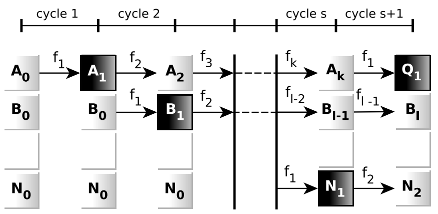
Fig. 4, the domain P includes the concrete timing properties, measured in *cycles per instruction* (CPI), in its definition. In this way, the fixpoint algorithm is able to compute, for each instruction, an invariant on the hardware states that can occur whenever execution reaches that instruction. After fixpoint stabilization, these execution times are the values of the cost variables of the linear program.

4.1. Pipeline Analysis

We consider a tiled multicore architecture with several ARM9-based cores, shared memories and IO, as shown in Fig. 5(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 k pipeline stages and by simultaneously processing N instructions. The considered pipeline is an "in-order" pipeline with five pipeline stages (PS): *fetch* (FI), *decode* (DI), *execute* (EX), *memory access* (MEM), and *write back* (WB). Figure 5(b) illustrates a functional view on pipelining.



(a) Generic multicore architecture



(b) Functional overview of pipeline steps

Figure 5: 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 5 CPU cycles in the presence of pipeline *bubbles* caused by stalls due to pipeline hazards [62]. Examples of pipeline hazards in our analysis are described in [57, Example 10]. In these cases, we witness either a cache miss or a backward data dependency between the instructions or the impossibility to predict the program counter of the next instruction to fetch. For example, instruction B in Fig. 5(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 pipeline analysis by abstract interpretation presented in [62] introduces the notion of *resource association* as a pair $(k, \{r_{j_1}, \ldots, r_{j_n}\})$, where $k \in PS$ 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 resource carries its own state. The particularity of our approach is that the state of the dynamically allocated sequences is updated after each pipeline stage. Hence, concrete timing information is combined with the abstract state of resources in a single hybrid pipeline state P. When compared to [62], our approach detects infeasible paths more precisely, because the path-sensitivity of the value analysis reduces the number of possible pipeline states.

The extra set of store buffers $R'^{\sharp}$, $D'^{\sharp}$ and $M'^{\sharp}$ are domains containing the resource states that are to be allocated across the internal stages of the pipelining of every instruction. This means that before and after analyzing an instruction, it is required to compute the least upper bound between the top-level domains $R^{\sharp}$, $D^{\sharp}$ and $M^{\sharp}$, contained in the CPU domain $\mathbb{C}$, and the store buffers $R'^{\sharp}$, $D'^{\sharp}$ and $M'^{\sharp}$. Formally, the hybrid pipeline state is defined as:

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

where *Time* is the global number of CPU cycles, *Pc* is the *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.

$$\mathit{Coord} \stackrel{\mathsf{def}}{=} [\mathit{TimedTask}]_N$$

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

$$\begin{aligned} \mathit{TimedTask} &\in (\mathit{Cycles} \times \mathit{Stage} \times \mathit{Task}) \\ \mathit{Task} &\in (\mathit{Instr} \times \mathit{Pc} \times \mathit{Demand} \times R'^{\sharp} \times D'^{\sharp} \times M'^{\sharp}) \end{aligned}$$

The semantic transformers required by our *functional approach* to pipeline analysis are described by the following. 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 Fig. 5(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 $P^{\sharp}$, which uses F_P to transform each element in the input set. The semantic transformers F_P and $F_P^{\sharp}$ are concisely defined as:

$$\begin{aligned} F_P &\in \mathit{Instr} \mapsto P \mapsto P \\ F_P(i)(p) &\stackrel{\mathsf{def}}{=} \mathit{toContext}(i) \circ [F_T \circ \mathit{fromContext}(p)]_N \\ F_P^{\sharp} &\in \mathit{Instr} \mapsto P^{\sharp} \mapsto P^{\sharp} \\ F_P^{\sharp}(i)(p^{\sharp}) &\stackrel{\mathsf{def}}{=} \{F_P^{5+}(i)(p) \mid p \in p^{\sharp}\} \end{aligned}$$

where F_P^{5+} corresponds to the recursive functional application of F_P at least five times. Note that F_P^{5+} does not correspond to the transitive closure of F_P by the fact that *local* execution times 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.

Nonetheless, even in fully timing compositional architectures [24], such as ARM9, the nondeterminism 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 nondeterminism 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:

$$F_P^{s^i_{k+1}}(i)(p) \overset{\mathsf{def}}{=} F_P(i)(F_P^{s^i_k}(i)(p))$$
$$F_P^{5+} \overset{\mathsf{def}}{=} F_P^{s^i_{WB}}$$

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.

Consider, for example, that the current stage is *FI* and there is free space inside the pipeline to fetch a new instruction from memory. Depending on the context of the actual pipeline state, structural hazards, such as a cache miss may delay memory access and, therefore, create a pipeline bubble. In the case of a cache hit, the *TimedTask* enters the *DI* stage. In both cases, the timing property *Cycles* is calculated according to the timing model of the processor [58]. This concrete timing model is parametrizable and is not specific to ARM9.

Example 4. Examples of sets of pipeline states in presence of hazards. To better describe all aspects involved in our timing static analysis, examples of pipeline hazards are given for the assembler program described in Fig. 2(b). Three types of pipeline hazard can be the cause of pipeline *bubbles*: 1) structural hazards that are caused by resource conflicts arising when the pipeline does not have enough resources to execute all the possible combinations of instructions without stalling; 2) the data hazards that are a natural cause of the predetermined order of instructions and, consequently, of the logical dependency between the operands of these instructions; and 3) the control hazards that arise when the destination address of branch instructions is not resolved early enough to decide which instruction should enter the pipeline next.

An example of structural hazards is when the next fetched instruction is not found inside the instruction cache. For instance, when the fixpoint algorithm reaches the program label 20, after computing the pipeline states for instruction '`b 16`', the next instruction entering the pipeline is '`ldr r3, [fp, #-16]`' (see Fig. 6(a)). At this program point, the fetched instruction cannot be found in the abstract cache state. Hence, a *cache miss* penalty relative for a main memory access request is computed according to the processor's timing model, leaving instruction '`ldr r3, [fp, #-16]`' in the *fetch* (*FI*) stage and causing the pipeline to stall. The corresponding timing penalty is 3 CPU cycles and it is computed using the $\mathcal{LR}$-server timing model, presented later in Section 6.1.

cycles	N	Next	State
0	0	*FI*	*Ready*: `nop`
3	1	*FI*	*Stalled*: `ldr r3, [fp,#-16]`
0	2	*FI*	*Ready*: `nop`

cycles	N	Next	State
3	0	*FI*	*Stalled*: `cmp r3, #0`
4	1	*DI*	*Feched*: `ldr r3, [fp,#-16]`
0	2	*FI*	*Ready*: `nop`

(a) Structural hazard

cycles	N	Next	State
1	0	*DI*	*Fetched*: `sub r3, r3, #1]`
2	1	*EX*	*Decoded*: `ldr r3, [fp,#-16]`
0	2	*FI*	*Ready*: `nop`

cycles	N	Next	State
2	0	*DI*	*Stalled*: `sub r3, r3, #1`
3	1	*MEM*	*Executed*: `ldr r3, [fp,#-16]`
1	2	*DI*	*Fetched*: `str r3, [fp,#-16]`

(b) Data hazard

Cycles	N	Next	PC	State
0	0	*FI*	0	*Ready*: `nop`
9	1	*WB*	68	*Stalled*: `bgt -20`
0	2	*FI*	0	*Ready*: `nop`

Cycles	N	Next	PC	State
0	0	*FI*	0	*Ready*: `nop`
9	1	*WB*	92	*Stalled*: `bgt -20`
0	2	*FI*	0	*Ready*: `nop`

(c) Control hazard

Figure 6: Examples of pipeline states sets in presence of hazards

Data hazards happen whenever the private data of an instruction is currently being processed (blocked) inside the pipeline by other instruction. In these cases, the blocked instruction can proceed only after the instruction holding the data has completed the *write back* phase (*WB*). One example of data stalling is given in Fig. 6(b) for the instruction '`sub r3, r3, #1]`', at program label 18 during the second fixpoint iteration for the loop. Although there are no cache misses to be accounted for, the instruction '`ldr r3, [fp, #-16]`' is in its *decode* stage (*DI*), meaning that accesses to register '`r3`' are blocked to other instructions. Therefore, the instruction '`sub r3, r3, #1]`' causes a *bubble* in the pipeline and the number of CPU cycles is incremented until the value for '`r3`' has been sent to the "context" of the pipeline.

Control hazards happen in circumstances like the one described in Fig. 6(c). We known for a fact that the instruction '`bgt -20`' may change control flow if the branch condition is verified. For this reason, and in the absence of a branch prediction mechanism, the pipeline is flushed so that the next program counter to fetch is available, right after the instruction '`bgt -20`' writes back. Fig. 6(c) illustrates the context and flow sensitivity of the pipeline analysis by showing two of the pipelines states computed for the branch instruction (an extra column shows the computed program counter). On the left side, we show the pipeline state after reaching the least fixpoint condition for the loop. The program counter 68 is computed 4 times and, under the assumption of compositionally, the abstract pipeline state (a set of concrete pipeline states) allows us to compute the local execution time (in *CPI*) at program point 17 as the maximum value in $\{11, 9, 9, 9\}$. On the right side, we show that the same instruction computes the value 92 for the program counter. Therefore, the abstract pipeline state at program label 23 is a singleton set, whose timing property is 9 *CPI*. ▲

5. Semantics-based WCET Verification

As the previous section demonstrates, the WCET analysis depends on hardware components, such as cache memories and pipelines, and the cost and complexity of the analysis is expected to increase as the internal design of such components becomes more sophisticated. The same applies when reasoning about WCET verification. Therefore, the methods used to compute the WCET must have good certifying properties, so that they can be applied to efficiently verify the correctness of the WCET upon reception. Indeed, the theory of fixpoints used in ACC [9] effectively performs the verification of the existence of least fixpoints. Hence, existing solutions based of fixpoint theory can be reused *as is* to compute/verify the invariants on execution times at every program point.

However, due to the limited computational resources of embedded systems, the design of a verification mechanism to check WCET estimates cannot integrate the simplex solver as part of the checking algorithm. This article proposes an extension of the existing ACC framework with an additional efficient mechanism to check the solutions of the linear program. However, the main goal of the verification framework remains unchanged: the software update is guaranteed to be safe if the received certificate, packed along with the "untrusted" program, can be proven correct with respect to some safety policy. The particular case of WCET verification implies that, in order to prove the optimality of the linear solutions, certificates must be proven to be least fixpoints, as computed by the pipeline analysis on the supplier side.

This section is organized as follows. We start with the description of a transformation algebra on the intermediate graph language, whose main objective is to *reduce the size* and *minimize the checking time* of ACC certificates. Then, we present and highlight the main contributions of the LP checker design in Section 5.2, in particular, the possibility to apply the certifying properties of duality theory in linear programming in order to *minimize the checking time of the optimality of linear solutions.* In Section 5.3, we present the experimental results for the certificate size reduction and checking time for a subset of the Mälardalen WCET benchmark programs [33]. Although the design of our approach differs in terms of style with [7], the experimental results of both approaches are compared and examined.

5.1. Transformation Algebra on Dependency Graphs

Algebraic transformations are performed on an induction definition that we named *intermediate graph language.* Terms of this language encode dependency graphs, whose building blocks are relations, and expresses control-flow patterns in accordance with the higher-order combinators of the two-level denotational meta-language introduced in Section 3. Two transformations are defined: the first is named *sequential* and the second is named *recursive.* The former significantly reduces the size of certificates and the latter is used to extract highly efficient fixpoint algorithms in terms of verification time.

The *sequential* transformation benefits from the compositional design of the fixpoint semantics to compute the global effect of a sequence of instructions.

However, the length increase on the syntactical term must preserve loop bounds to ensure the tightness and soundness of the WCET. A sequence of adjacent relations is reduced into a single relation by using the algebraic rules of the intermediate graph language. The main advantage is the reduction of the number of connected subgraphs and, consequently, the reduction of the number of program points considered during static analysis. However, the sequential transformation must be proven sound in relation to the results of the program flow analysis. Hence, a given set of adjacent relations is transformed into a single relation if and only if the loop bounds at the entry/exit labels of the latter are preserved in relation to the former set.

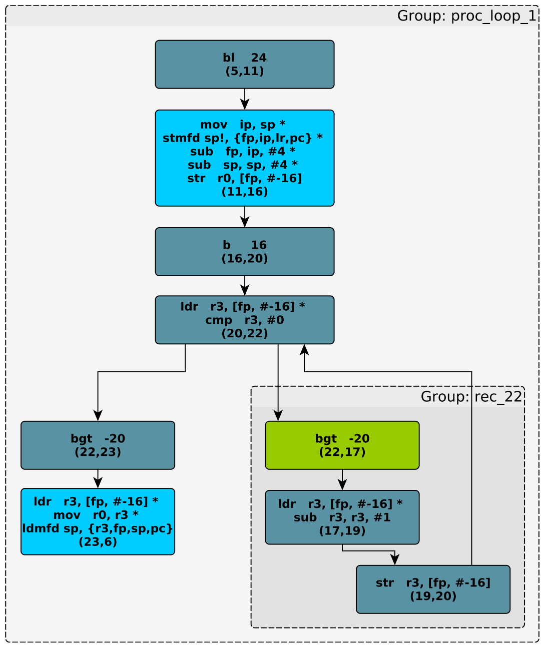
As mentioned in Section 3, the sequential composition $(\cdot * \cdot)$ of two relations T and R is defined by $a(T * R)c$ iff there exists b such that aTb and bRc. The usefulness of the point-free notation is that the input value a is associated to the right with the output value c, allowing the sequential composition to be redefined simply as $(T * R)$ with type $(a \rightarrow c)$. In terms of fixpoint computation, the interpretation of the relational composition $(\cdot * \cdot)$ in λ-calculus is the functional composition operator $(\cdot \circ \cdot)$. Thus, the interpretation of a relation with multiple syntactical elements performs multiple functional applications in order to obtain a value with the same type, but within the same fixpoint iteration, i.e. considering only one pair of input/output labels.

The *recursive* transformation removes the loops from the dependency graphs so that the generated fixpoint checking algorithm uses a single state traversal across purely sequential programs. The soundness of this transformation is based on the two following facts. Firstly, according to the weak topological order of an inter-procedural loop, e.g. the one given in Ex. (1), the meta-program has part of the loop body expression as the prefix of the combinator $(\cdot \oplus \cdot)$, as described in Ex. (2). Nevertheless, as provided by the path-sensitive analysis, the loop condition is analyzed even when the precondition for entering the loop is not satisfied, forwarding the analysis on the fall-through path.

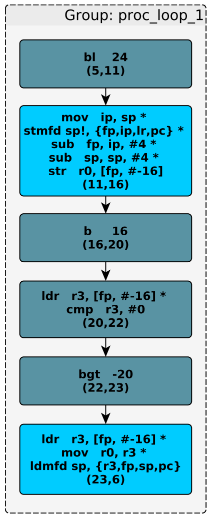
Secondly, by stating that the recursive chaotic strategy proceeds by stabilizing every subcomponent before the outer-component is stabilized [15], the stabilization of any loop can be detected by checking the invariant at the *head* of the loop. Hence, the inductive definition of dependency graphs can easily be used to remove all recursive subgraphs. Therefore, all meta-programs that are automatically generated by an interpretation of their dependency graphs on the consumer side are pure sequential fixpoint algorithms.

Example 5. Transformation of Dependency Graphs
This example describes how terms of our intermediate graph language can be transformed into reduced dependency graphs that, by design, have adequate characteristics for efficient fixpoint checking. The *sequential* transformation in Fig. 7(a) is relative to the procedure '`loop`' and is performed on top of the set of the input-output relations presented in Fig. 2(b). The dependency graph obtained with *recursive* transformation is given in Fig. 7(b).

The instructions '`bl 24`' and '`b 16`' are excluded from the process because they are "branch" instructions. The same applies for the path-sensitive instruc-



(a) Reduction of program points inside '`loop`'



(b) Purely sequential graph of '`loop`'

Figure 7: Examples of transformations of intermediate graph representations

tion '`bgt -20`'. Finally, the instruction '`str r3, [fp, #-16]`' cannot be reduced because according to the program-flow information computed for the original dependency graph, multiple nodes with the "sink" label '20' have different loop iteration counts. All the remaining instructions are sequentially reduced by connecting the corresponding relations and preserving the program-flow information. The main advantage is the reduction of program points included in the certification and the fact that only pipeline states that have reached the *WB* stage need to be included in the certificate. ▲

5.2. WCET Verification

In this section, we propose the inclusion of the WCET checking phase inside the ACC framework using the *duality theory* applied to linear programming. Additionally, Section 5.2.3 presents a proof sketch that the coefficient matrix of the LP problem is *totally unimodular* [37]. In this way, the LP checker is able to execute in linear time, by checking LP solutions using simple linear-algebra computations, without the need to re-run the simplex method.

Given the assembler program P, the structure of ACC certificates correspond to the map $\Sigma[\![P]\!]$ of Def. (3), consisting of the abstract contexts computed during *program-flow analysis*, plus the solutions computed by the primal/dual simplex method on the supplier side. On the consumer side, along the lines of ACC [9], the verification of abstract contexts of Def. (2), Invs$[\![P]\!]$, is performed by a single-pass fixpoint iteration over the assembly program P, while the LP checking of the Primal, Dual and WCET solutions is based on the duality theory [51]. The idea behind duality theory applied to linear programming is that for any solution of the simplex algorithm there is another associated solution called the *dual*. The relationships between the dual problem and the original problem (called the *primal*) is useful to determine if the received LP solutions on the consumer side are in fact the *optimal* ones, that is, the solutions that maximize the objective function (WCET) on the supplier side.

5.2.1. The ILP Verification Problem

The optimization problem is defined as the maximization of the objective function WCET, subject to a set of linear constraints. The variables of the problem are the node iteration variables, x_k, which are defined in terms of the arc iteration variables, d^{IN}_{ki} and d^{OUT}_{kj}. Arc iteration variables correspond to the incoming (i) and outgoing (j) arcs to/from a particular program label identifier k contained in the weak topological order $\langle \mathrm{in}_P, \preceq \rangle$, where in_P denotes the set of labels of a program P. These linear constraints are called *flow-conservation* constraints. Additionally, a set of *capacity constraints* establish the upper bounds, b_{ki} and b_{kj}, for the arc iterations.

$$x_k = \sum_{i=1}^{n} d^{\mathrm{IN}}_{ki} = \sum_{j=1}^{m} d^{\mathrm{OUT}}_{kj} \tag{15}$$

$$d^{\mathrm{IN}}_{ki} \leqslant b_{ki} \ \text{ and } \ d^{\mathrm{OUT}}_{kj} \leqslant b_{kj} \tag{16}$$

The objective function is a linear function corresponding to the number of node iterations on each label identifier k, weighted by the set c_k, which specifies the corresponding execution costs, measured in *cycles per instruction* (CPI).

$$\mathrm{WCET} = \sum_{k \in in_P} c_k \cdot x_k$$

When compared to the AI+ILP approach to WCET analysis [72], the structure of our optimization problem is particular in the sense it is guaranteed that integer values are always assigned to each variable. Indeed, *the proof of total unimodularity not only allows the use of the simplex method based on floating point arithmetic, but also guarantees the integral (algebraic integer) semantics on the linear program, hence preserving precision.* This is the main advantage of our approach to WCET verification because it allows us to omit integrality constraints, thereby enabling the use of the certifying duality properties, which only apply to linear programming in general and not to ILP.

Next, we describe how the set of linear constraints can be formally obtained using the theory of abstract interpretation. To this end, we state the fact that program states differ on their labels [18] in order to index a program state in $\Sigma[\![P]\!]$ by some corresponding label in in_P, according to Def. (3). Let $\mathrm{d}[\![P]\!]$ denote a set of identifiers for input-output relations in P. Examples of relation identifiers are "d1", "d5", etc. in Fig. 2(b). Then, the flow-conservation constraints of Def. (15) are a set of equations of type $\wp(\mathrm{in}_P \mapsto \wp(\mathrm{d}[\![P]\!]))$, where

in_P is the set of labels in P. Therefore, we apply the right-image isomorphism $\langle \alpha, \gamma \rangle$ [17, Section 8] to the relational semantics, R, in order to obtain a set of flow conservation constraints, C, in a correct way:

$$\begin{aligned} \alpha(R) =& \{x_k = \textstyle\sum_{i=1}^{n} d_{ki}^{\mathrm{IN}} \mid \exists x_k \in \mathrm{in}_P : d_k^{\mathrm{IN}} = \{d_i \mid \exists x_l \in \mathrm{in}_P : \langle x_k, d_i, x_l \rangle \in R\}\} \cup \\ & \{x_k = \textstyle\sum_{j=1}^{m} d_{kj}^{\mathrm{OUT}} \mid \exists x_k \in \mathrm{in}_P : d_k^{\mathrm{OUT}} = \{d_j \mid \exists x_l \in \mathrm{in}_P : \langle x_l, d_j, x_k \rangle \in R\}\} \end{aligned} \quad (17)$$

$$\begin{aligned} \gamma(C) = \{\langle x_k, d_i, x_l \rangle \mid \quad & \exists s_1 \in C, \exists d_j \in rhs(s_1) : x_k \in lhs(s_1) \ \wedge \\ & \exists s_2 \in C, \exists d_j \in rhs(s_2) : x_l \in lhs(s_2) \ \wedge \ d_i \equiv d_j\} \end{aligned} \quad (18)$$

5.2.2. Verification Mechanism

Both the objective function and the set of linear constraints can be represented in matrix form. For this purpose, we need to abstract from the node (x) and arc (d) iteration variables previously defined and consider a single set of variables ($\mathbf{x}$), indexed by non-negative values. In particular, the cost values associated with arc variables are zero in the objective function and the arc iteration bounds ($\mathbf{b}$) are zero for all linear constraints including a node variable.

The equation system of the *primal* problem is defined in terms of the matrix $\mathbf{A}$, with the coefficients of constraints (15) and (16), the column vector $\mathbf{x}$ of variables and the column vector $\mathbf{b}$ of capacity constraints. Then, given the row vector $\mathbf{c}$ of cost coefficients, the objective of the primal problem is to maximize the WCET $= \mathbf{cx}$, subject to $\mathbf{Ax} \leqslant \mathbf{b}$. Conversely, the *dual* problem is also defined in terms of the vectors $\mathbf{c}$ and $\mathbf{b}$ plus the matrix $\mathbf{A}$, but the set of dual variables are organized in a complementary row vector $\mathbf{y}$. Then, the objective of the dual problem is to minimize WCET $^{\mathrm{DUAL}} = \mathbf{yb}$, subject to $\mathbf{yA} \geqslant \mathbf{c}$.

Using the simplex method, it is possible to compute a feasible solution $\mathbf{x}$ for the primal problem and a paired feasible solution $\mathbf{y}$ for the dual problem. The *strong duality property* of the relationship between this pair of solutions for the purpose of LP checking is: the vector $\mathbf{x}$ is the optimal solution for the primal problem if and only if:

$$\mathrm{WCET} = \mathbf{cx} = \mathbf{yb} = \mathrm{WCET}^{\mathrm{DUAL}}$$

In the ACC setting, this property allows us to use simple linear-algebra algorithms to verify the LP solutions that were computed using the simplex method. The verification mechanism is composed by three steps:

1. Use the static analyzer to verify the local execution times present in the fixpoint. If valid, execution times are organized in the cost row vector **c'**. Then, take the received primal solutions **x'** and solve the equation WCET' = **c'x'** to check if it is equal to the received WCET.

2. Use the static analyzer to verify the loop bounds abstract context. If valid, loop bounds are organized in the row capacities vector **b'**. Then, take the received dual solutions **y'** and verify the strong duality property by testing the equality of the equation **c'x'** = **y'b'**.

3. Extract the coefficient matrix **A'** from the received code and check if the received primal and dual solutions satisfy the equations **A'x'** $\leqslant$ **b'** and **y'A'** $\geqslant$ **c'**. In conjunction with the two previous steps, this allows us to conclude that **x'** and **y'** are the optimal solutions of the primal and dual problem and, therefore, conclude that the LP verification is successful.

Example 6. Numerical example of a LP program.
Next, we give a numeric example of the LP problem associated with the source program in Example 1. A subset of the relational semantics of the corresponding assembler program is shown in Fig. 2(b). For each transition relation, Fig. 2(b) includes the name of the arc, indexed to the variable name d, that would correspond to the graph view of the relational semantics. For example, the feedback arc between the nodes "**n17**" and "**head_22**" is called "**d22**".

Vars (**x**)	Primal (**x***)	Costs in CPU cycles (**c**)
$\cdots$	–	–
x_{16}	1.0	8
x_{17}	3.0	11
x_{18}	3.0	4
x_{19}	3.0	8
x_{20}	4.0	10
x_{21}	4.0	9
x_{22}	4.0	9
x_{23}	1.0	9
$\cdots$	–	–
d_{21}	4.0	0
d_{22}	3.0	0
d_{23}	1.0	0
$\cdots$	–	–

(a) Costs and primal values

	Coefficients of variables (matrix **A**)		Constants (**b**)	Dual (**y***)
Flow Conservation	$\cdots$	$=$	–	–
	$x_{17} - x_{18}$	$=$	0	10.0
	$x_{17} - x_{22}$	$=$	0	-21.0
	$x_{20} - x_{21}$	$=$	0	39.0
	$x_{20} - x_{19} - x_{16}$	$=$	0	-49.0
	$x_{22} - x_{17} - x_{23}$	$=$	0	21.0
	$x_{22} - x_{21}$	$=$	0	-30.0
Capacities	$\cdots$	$\leqslant$	–	–
	d_{22}	$\leqslant$	3	0.0
	d_{21}	$\leqslant$	4	0.0
	d_{20}	$\leqslant$	4	0.0
	d_{19}	$\leqslant$	3	0.0
	d_{18}	$\leqslant$	3	0.0
	d_{17}	$\leqslant$	3	51.0
	$\cdots$	$\leqslant$	–	–

(b) Linear equation system and dual values

Figure 8: Numeric example of the LP problem in matrix form

Table 8(a) shows the primal values and execution costs associated with the LP variables (columns in the matrix **A**). For sake of readability, the column **x** displays the node variables (x) plus the un-renamed arc variables (d). As already mentioned, the execution cost associated with arc variables in vector **c** is equal to zero. The column **x*** contains the optimal (primal) solutions for the variable names x_k, where $k \in N$, and for the arc variable names d_{ki}^{IN} and d_{kj}^{OUT}, where $i, j \in E$. Table 8(b) shows the linear equation system from which the coefficient matrix **A** is inferred. The vector **b** contains the arc iteration upper bounds that are obtained directly from the *program flow* certificate and the optimal (dual) solutions are given by the vector **y***.

Although no integral constraints are necessary, the observation of both primal (**x***) and dual (**y***) solutions of the LP program shows that they are convertible to integers with no loss of precision. Next, we formally demonstrate that the relaxation of the ILP problem is safe under the conjecture of total unimodularity of the matrix **A**. ▲

5.2.3. Total Unimodularity by Construction

This section presents a proof sketch that the coefficient matrix is totally unimodular based on the fact that the *determinant* $\mid \mathbf{A} \mid$ of a square matrix $\mathbf{A}$ can be expanded from the "pivot" determinant of a 2×2 matrix to any matrix size by using *minors* and *cofactors* [37]. We conclude that the variables x_k are irrelevant by rewriting the WCET objective function as $\sum_{k \in in_P} c_k \cdot \left(\sum_{i=1}^{n} d_{ki}^{\mathrm{IN}}\right)$, which fits inside the category of *minimum cost network flow problems.*

Proposition 1. The coefficient matrix $\mathbf{A}$ is called *totally unimodular* iff each of its sub-determinants equals 0, 1, or -1. In particular, each entry of $\mathbf{A}$ is either 0, 1, or -1. Then, the solutions $\mathbf{x}$ in $\mathbf{Ax} \leqslant \mathbf{b}$ are integral.

Proof sketch. A proof by construction can be deduced from the abstraction function (17). In essence, it specifies that if we take the label of each program state $\Sigma[\![P]\!]$ and build a relational semantics R containing only program labels (x_k) and arc identifiers (d_{ki}^{IN} and d_{kj}^{OUT}), then the set of input-output relations in R, ordered the by the input label k, can be abstracted using the function α into a set of linear equations, C, in the form of Def. (15).

By the fact that input-output relations with the same label at the input and output positions are not allowed, we conclude that, for each label k, there are exactly two linear equations for each program label x_k, according to the *flow-conservation* constraints. The first is given by $\sum_{i=1}^{n} d_{ki}^{\mathrm{IN}}$, where n is the size of the set of incoming arcs to x_k. The second is given by $\sum_{j=1}^{m} d_{kj}^{\mathrm{OUT}}$, where m is the size of the set of outgoing arcs from x_k. Therefore, given one program label x_k, the arcs d_{ki}^{IN} and d_{kj}^{OUT} are necessarily different from each other.

Moreover, since each linear constraint is stored in a different row in matrix $\mathbf{A}$, and since each constraint can be transformed into an equation that is equal to 0, the arc variables all have coefficients equal to -1. Conversely, the program label variables all have coefficients equal to 1. These properties allow us to exclude the possible cases where a sub-determinant could be different from 0, 1, or -1. First, we identify the cases where the determinant of a matrix does not satisfy Prop. (1). Any matrix $\mathbf{A}$ of the form:

$$\mathbf{A} = \begin{bmatrix} \vdots & \vdots & \vdots & \vdots & \vdots \\ \cdots & (+1) & \cdots & (+1) & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \cdots & (+1) & \cdots & (-1) & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

is not totally unimodular because it has a square submatrix of determinant -2. We proceed by *reduction ad absurdum* to identify the base cases where the determinant of a square matrix 2×2 is -2 and then preclude it by using the properties of abstraction function $\alpha(R)$. In these cases, only one coefficient is -1 and the others are 1: a) $\left[\begin{smallmatrix} +1 & +1 \\ +1 & -1 \end{smallmatrix}\right]$, b) $\left[\begin{smallmatrix} +1 & +1 \\ -1 & +1 \end{smallmatrix}\right]$, c) $\left[\begin{smallmatrix} +1 & -1 \\ +1 & +1 \end{smallmatrix}\right]$ and d) $\left[\begin{smallmatrix} -1 & +1 \\ +1 & +1 \end{smallmatrix}\right]$. Since each of the variables x_k, d_{ki}^{IN} and d_{kj}^{OUT} correspond to a separate column of $\mathbf{A}$ (they constitute the column vector $\mathbf{x}$), we can deduce the following.

On one hand, the sub-matrices containing only arc variables, d_{ki}^{IN} or d_{kj}^{OUT}, all have coefficients 0 or -1. Similarly, sub-matrices containing only node variables, x_k, all have coefficients 0 or 1. Therefore, such sub-matrices satisfy the premises of the "pivot" determinant operation. On the other hand, in the cases where a sub-matrix contains node variables on the left column and arc variables on the right column, we cannot construct any of the counter-example 2×2 square matrices because:

- The coefficients on the left are either both 1 or 1 and 0 (or vice-versa) because there are at most two equations for each node variable x_k. Therefore, cases b) and d) above are precluded.
- The remaining cases, a) and c), are also precluded because the abstraction function $\alpha(R)$ guarantees that the variables d_{ki}^{IN} and d_{kj}^{OUT} do not overlap inside a 2×2 sub-matrix for a given x_k. In other words, the same arc cannot enter and leave one node at the same time.

The generalization of the definition of determinant for all sizes can be given by the following. For each element of a $n \times n$ matrix, there is a *minor* value that is a $(n-1) \times (n-1)$ determinant. Then, each of those elements is expanded until the "pivot" 2×2 determinants are reached. Then, a *cofactor* value is calculated for every element as either the *minor* or its opposite in sign. Finally, we require the definition of a *sign chart* with size $n \times n$, where every entry is either $+$ or $-$. The first element (row 1, column 1) is always $+$ and it alternates from there.

The method to calculate the determinant is the following. Pick any row or column and multiply every element in that row or column by its *cofactor* and add the results. To prove that the variables x_k are irrelevant, let us pick a column for some $k \in in_P$. For any sub-matrix with size bigger than 2, there are at moust two values different from zero. Hence, according to the definition of $\alpha(R)$, these values are in adjacent positions. Moreover, the *sign chart* assigns alternate signs to adjacent positions. Therefore, after eliminating the counter-examples of the "pivot" determinants, we conclude that the two lines for which x_k is 1 will cancel each other out, because the corresponding *cofactors* have different signs. Hence, we prove our assumptions that WCET $= \sum_{k \in in_P} c_k \cdot \left(\sum_{i=1}^{n} d_{ki}^{\mathrm{IN}} \right)$ and that the coefficient matrix is totally unimodular, because the linear problem is equivalent to a minimum cost network flow problem. $\square$

5.3. Analysis of Experimental Results

This section presents the experimental results of the checking mechanism for a subset of the Mälardalen WCET benchmark [33]. For the purpose of certificate reduction, the *sequential* transformation is particularly relevant. The reduction of the certificate sizes, including the example in Fig. 2(a), is shown in Table 3. The number of program points resulting from the reduction is also observed. However, the reduction of the certificate size is not lineally proportional to the reduction of program points because the number of "hybrid" pipeline states in each program point also depends on which program paths are actually feasible. For example, the benchmark `cover` has a large number of '`case`' patterns, each followed by a '`break`' statement. Hence, although the reduction of program points is not the highest, the certificate size reduction is about 3.317 times the size of the original certificate. Similarly, there is only pass per loop in the `ud` benchmark, which makes the gain in certificate size reduction very significant.

Table 3: Variation of the certificate size

Filename	Source (KBytes)	Assembler (KBytes)	Original Cert. (KBytes)	No. of P. Points	Transf. Cert. (KBytes)	No. of P. Points	Reduction Factor
loop	0.12	9.2	14	27	12.0	10	1.167
bs	4.2	10.5	149	70	61	19	2.243
bsort	2.5	11.2	97	127	71	30	1.366
cnt	2.9	11.7	95	148	71	39	1.338
cover	6.3	14.9	136	56	41	23	3.317
crc	5.5	12.0	419	262	262	43	1.599
expint	3.9	12.3	163	185	113	42	1.442
fdct	8.7	13.5	2137	709	1708	15	1.251
fibcall	3.7	10.1	91	48	32	12	2.844
matmult	3.9	11.7	207	165	134	52	1.545
minmax	0.89	10.8	117	139	67	41	1.746
prime	0.88	12.1	159	127	102	41	1.559
ud	4.9	12.9	840	85	95	10	8.842

However, different results are obtained for structured programs with a high number of loop iterations, for example `fdct`. Despite having the highest percentage in terms of reduction of program points, the fact that only pipeline states that have reached the *WB* stage are stored has a small impact on certificate reduction. The main reason is that full loop unrolling propagates intermediary state at the head of the loops across fixpoint iterations and "branch" instructions cannot be sequentially reduced. Moreover, precision in detecting infeasible paths compromises efficiency when the "history" of the pipeline analysis is relevant for all the instructions inside loops. For the example of `fdct`, the gain in certificate reduction is only 1.251 times the size of the original certificate and, when compared to the size of the assembler file, the certificate size can be more than one hundred times bigger.

Compared to the results presented in [7], the size of our certificates is not in the same order of magnitude as the size of assembler files. In fact, the experiments in [7] use the highly optimized *sharing+freeness* abstract domain [49], while our abstract pipeline domain is a "hybrid" domain, where the value and cache abstract domains are non-relational and need to be shared, and replicated, with a set of concrete timing properties. Additionally, full loop unrolling increases stored data due to the impossibility of using convergence accelerators [18]. However, *relatively to the original certificate, our certificate reduction mechanism based on dependency-graph transformations falls behind the approach of [7], whose reduction factor is about 3.35 on average, whereas our approach achieves a reduction factor of 2.33 on average.* However, as described in Section 5.2, the gain in verification time is significantly greater compared to [7].

Experimental results concerning the overall checking time are given in Table 4 (obtained off-device using an Intel®Core2 Duo Processor at 2.8 GHz). The first term of the sum is relative to the fixpoint algorithm and the second is relative to the LP simplex method. The checking time of the solutions of the

LP problem is close to zero in all cases because the verification mechanism uses simple matrix operations to check that the received solution at consumer sites is indeed the optimal one. *The experiments show that the LP verification time is almost constant for every benchmark. This demonstrates the high efficiency of the certifying mechanism of dual theory.*

Table 4: Verification Time Experimental Results

Benchmark	Generation Time (sec)	Verification Time (sec)	Ratio (%)
loop (3)	0.417 + 0.006	0.311 + 0.000	25.4
loop (9)	2.238 + 0.008	0.458 + 0.000	79.5
bs	2.209 + 0.412	0.260 + 0.001	88.23
bsort	0.751 + 0.506	0.219 + 0.002	70.8
cnt	0.673 + 0.161	0.343 + 0.003	49.0
cover	2.222 + 0.445	0.225 + 0.001	89.9
crc	14.28 + 0.066	2.031 + 0.011	85.8
expint	2.235 + 0.447	0.226 + 0.001	89.9
fdct	1.852 + 0.055	0.520 + 0.004	71.9
fibcall	260.0 + 1.839	22.66 + 0.003	91.3
matmult	2.969 + 0.042	0.398 + 0.005	86.6
minmax	1.263 + 0.029	0.306 + 0.002	75.8
prime	2.017 + 0.030	0.324 + 0.008	83.9
ud	42.51 + 0.288	0.596 + 0.000	98.60

As illustrated in Fig. 7(b), all nodes of the dependency graph after the *recursive* transformation have depth "0". In this way, and along the lines of [7], the fixpoint algorithm is able to check the validity of certificates within a single chaotic iteration. Although this algorithm is generated from a single-path dependency graph, the input certificate contains all the history of computation up to the least fixpoint solution. Therefore, the fixpoint condition is checked at the *head* labels of loops, which are kept in the transformed graph. Nevertheless, the execution times at the program points excluded from the single-path graph must be used to check the LP objective function in order to validate the WCET.

Curiously, experiments also show that the verification time of certificates is strongly reduced for the recursive parts of programs, but not for the purely sequential parts. Consider Ex. (1), where the function '`loop (x)`' executes '`x`' times. The computation time increases with the number of iterations, e.g. when the arguments are "3" and "9". However, the same is not observed if the fixpoint checking algorithm has no recursive combinators. The reason is the following: knowing that before running the fixpoint algorithm the value at every program is the absence of information, $\perp_{\mathbb{C}}$, the particularity of purely sequential programs is the fact that data-flow functions are computed exactly once. In terms of implementation, this affects results because the comparison operator between abstract values takes longer to compute than the equality test with $\perp_{\mathbb{C}}$.

As mentioned before, dependency graphs containing a substantial number of recursive sub-graphs allow the verification time to be greatly reduced. This is the case for all the benchmarks in Table 4 and it is most evident for those whose loops bounds are higher. A closer inspection of the function calls '`loop (3)`' and '`loop (9)`' shows that by increasing the loop bounds by a factor of 3 makes the fixpoint checker run 3 times faster. For the benchmark that has the highest loop bounds, '`fdct`' benchmark, the gain in efficiency is above 70%. In the case of the benchmark '`ud`', which has a single loop inside the '`main`' function, the gain is indeed extremely high (98%). In summary, *we are able to gain* 78% *in verification time on average, which demonstrates that our WCET checker is overall highly efficient.*

6. WCET on Multicores

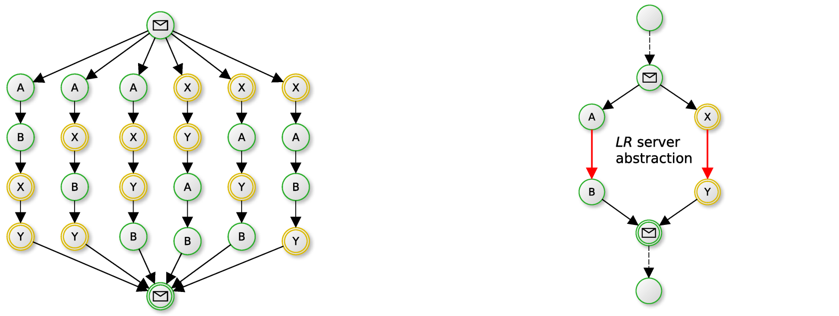
Nowadays, WCET estimation on embedded systems with multicore chips is one of the most challenging topics in timing analysis because of the intrinsic computational complexity of the analysis of multiple and concurrently executing processing units sharing resources. When compared to single-core architectures, the complexity of the timing analysis in multicores depends not only on hardware features, but also on the predictability of their timing behavior when some resources are shared, e.g. instruction and/or data memories. This means that, besides the control flows of a program, also the "architectural flows" or "interleavings", i.e. the number of ways in which a shared resource can be accessed, must be accounted for. This fact causes the complexity of the analysis to explode. In this section, we explain this problem in more detail and we present an approach for mitigating its nature while ensuring the scalability of the analysis.

To exemplify the complexity of 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. Generalizing for a program with n processes, each one executing a sequence of m instructions, the number of possible interleavings is $(n.m)!/(m!)^n$. This exponential growth of interleavings precludes any timing analysis to run in feasible time.

Consider as an example a tiled multicore with several ARM-based processor cores, sharing memories and IO, as shown previously in Fig. 5(a). Each processor core has an instruction pipeline and an instruction cache memory and we assume that applications do not share data. Compared to the single-core setting, the extra source of potential unpredictability is due to shared resources, an SRAM serving as the main instruction memory in this case. Shared accesses increase the execution time of programs and must hence be accurately and efficiently modeled, precluding the use of a constant penalty for every cache miss, as it was previously described for single-core timing analysis.

As described in Section 4.1, the pipeline analysis computes upper bounds on execution times, expressing the elapsed *cycles per instruction* (CPI) that are associated with a particular stage of an instruction inside the pipeline. The absence of an abstraction for the *concrete* CPI values in the abstract interpretation literature [62] implies that the *abstract* pipeline domain must be defined as a *set* of all pipeline states statically allowed within the program. For single-core architectures, this does not constitute a computational problem, because this set consists of a manageable number of pipeline states.

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 Fig. 9(a) and the original control flow is given in Fig. 9(b). Next, we present a sound and computationally efficient abstraction used for timing analysis of architectural flows. Assuming independence in the value domain, i.e. no data is shared between processes, the timing analysis of architectural flows depends only on the scheduling made by the arbiter of the shared resource. For arbiters that *statically partitions the resource to provide complete isolation between applications* 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 time slots to each processor. The benefit of this approach is that applications can be analyzed independently, even in a multicore environment, as Fig. 9(b) illustrates.



(a) Non-compositional timing analysis considering the architectural flows (b) Compositional timing analysis considering only the control flows

Figure 9: 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

However, a severe limitation is that the isolation approach is restricted to more or less a single resource arbiter that cannot be found in all systems. If the TDM arbiter is replaced by a more common work-conserving *round-robin* arbiter (RR), the system no longer isolates applications, since the scheduling of requests depends on the presence or absence of requests from other processor cores. In this case, the analysis of every scheduled sequence in Fig. 9(a) must be performed.

6.1. Latency-Rate Servers

Our proposed approach to a scalable WCET analysis of multicores with shared resources is to apply the latency-rate ($\mathcal{LR}$) server model [67], which defines an abstraction of shared resources in terms of upper bounds on access times. This implies that the interference between processor cores visible in Fig. 9(a) is removed (abstracted), enabling the analysis to only consider control flows. Moreover, it provides a compositional timing analysis that allows us to reuse the higher-order combinators presented in Section 3.4, which are used to estimate the WCET on single-cores. Additionally, as we will show in Section 6.2, the use of the $\mathcal{LR}$-server model can be formalized in the context of the abstract interpretation framework by a Galois connection, proving the soundness of its use in our approach to timing analysis.

We proceed by explaining the $\mathcal{LR}$-server model in more detail. In essence, $\mathcal{LR}$ servers guarantee a processor core a minimum allocated bandwidth, ρ, after a maximum service latency (*interference*), Θ. This is illustrated in Fig. 10, where a processor core requests service from a shared resource over time (red line) and the resource providing service (blue line). The $\mathcal{LR}$-service guarantee, the dashed line indicated as service bound in Fig. 10, provides a lower bound on the amount of data that can be transferred to a processor core during any interval of time independently of the behavior of other processor cores. This makes a $\mathcal{LR}$ server suitable for performance analysis of streaming applications concerned with the time to serve *sequences of requests* rather than single requests.

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 arbiters belonging to the class of $\mathcal{LR}$ servers are TDM, several varieties of the Round-Robin and Fair-Queuing algorithms [67], as well as priority-based arbiters like Credit-Controlled Static-Priority [3] and Priority-Based Scheduling [66]. Of course, the abstraction also relies on that the access time of a single resource access can be bounded, just like in the single-core case. This is straight-forward for simple resources, such as a SRAMs, but requires a more complex analysis for unpredictable resources like many commercial-of-the-shelf DRAM controllers [39]. For this reason, both software [75] and hardware solutions [2] have been presented to make DRAM behave in a more predictable manner and produce tighter bounds.

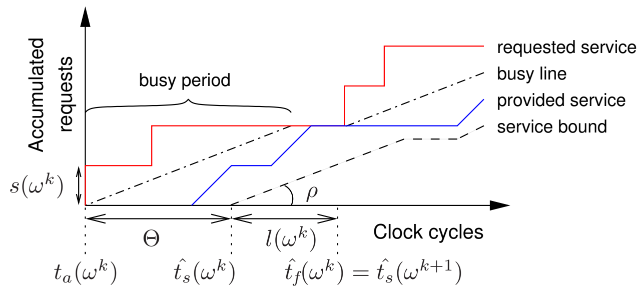


Figure 10: A $\mathcal{LR}$ server and its associated concepts.

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. As illustrated in Fig. 10, a 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*. The figure also shows how the service bound is shifted when the processor core is not in a busy period.

The intuitive benefit of the notion of busy periods is that if multiple requests are known to arrive close together, it is possible to show that they cannot all experience worst-case interference from other processor cores, resulting in lower WCET estimates. When integrated into a WCET analyzer, the information about busy periods is derived based on the concrete arrival times of request provided by the analyzer. Note that the result of the WCET analysis is always conservative no matter if the processor core is frequently busy or not. Accurately capturing busy periods is only a matter of tightness.

We proceed by showing how starting times and finishing times of requests in the shared resource are bounded using the $\mathcal{LR}$-server service bound illustrated in Fig. 10. For simplicity, we assume that a request corresponds to an instruction fetch that takes a single clock cycle to serve. From [71], the worst-case scheduling time, $\hat{t_s}$ of the k^{th} request from a processor core is expressed according to Def. (19), where $t_a(\omega^k)$ is the concrete arrival time of the request and $\hat{t_f}(\omega^{k-1})$ is the worst-case finishing time of the previous request from the same processor core. The worst-case finishing time, expressed in Def. (20), 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 defined as $l(\omega^k) = s(\omega^k)/\rho$.

$$\hat{t_s}(\omega^k) = \max(t_a(\omega^k) + \Theta, \hat{t_f}(\omega^{k-1})) \tag{19}$$

$$\hat{t_f}(\omega^k) = \hat{t_s}(\omega^k) + s(\omega^k)/\rho \tag{20}$$

Next, we give the declarative definitions for the temporal behavior of a general $\mathcal{LR}$ server. We then proceed by briefly explaining how the service latency and rate parameters are derived for the abstraction of a TDM arbiter, which later enables us to experimentally compare the accuracy and analysis time of the $\mathcal{LR}$ abstraction to a specialized analysis of TDM in Section 6.3. Let the datatype **ExTime** denote a timing property in terms CPI, *cycles*, and also the concrete timing properties specific to the $\mathcal{LR}$ server, $t_a(\omega^k)$ and $\hat{t_f}(\omega^{k-1})$.

$$\textbf{data ExTime} = \textbf{ExTime}\ \{\mathit{cycles} :: \mathit{Int}, t_a(w^k) :: \mathit{Int}, \hat{t}_f(w^{k-1}) :: \mathit{Int}\}$$

The worst-case finishing time, $\hat{t}_f(\omega^{k-1})$, is used by the $\mathcal{LR}$-server to model the interference experienced by a single processor core. According to Def. (19), the upper bound on the finishing time is dominated by the service latency of the arbiter, or by the finishing time of the previous request from the processor core, whichever is greater. Afterwards, according to Def. (20), the function *missed* defines the timing behavior of a cache miss:

```
$\mathit{missed}\ w@\textbf{ExTime}\ \{\mathit{cycles} = c, t_a(w^k), \hat{t}_f(w^{k-1})\}$
  $= \textbf{let}\ \mathit{newBusy} = \hat{t}_f(w^{k-1}) \leqslant t_a(w^k) + \Theta$
        $d = \textbf{if}\ \mathit{newBusy}\ \textbf{then}\ \Theta + (1/\rho)\ \textbf{else}\ 1/\rho$
    $\textbf{in}\ w\ \{\mathit{cycles} = c + d, \hat{t}_f(w^{k-1}) = d + (\textbf{if}\ \mathit{newBusy}\ \textbf{then}\ t_a(w^k)\ \textbf{else}\ \hat{t}_f(w^{k-1}))\}$
```

Having shown the declarative definitions of a general $\mathcal{LR}$ server, we proceed by showing how to derive the service latency and rate parameters of a processor core c for the $\mathcal{LR}$ abstraction of a TDM arbiter. The considered TDM arbiter is assumed to have a TDM table size (frame size) of f slots of which ϕ_c are allocated to core c. The rate allocated to core, ρ_c, is determined purely by the number of allocated slots in the schedule and is computed according to

Equation (21). However, the service latency depends on how the allocated slots are distributed in the schedule. A common way to distribute the slots is to use a *continuous allocation*, where slots allocated to a processor core appear consecutively in the schedule. Using this distribution, the service latency of a core, Θ_c, can simply be computed according to Equation (22). The service latency assumes that the busy period starts just after the last slot allocated to the processor core to maximize the number of interfering slots. A service latency analysis for arbitrary TDM slot distributions is presented in [4].

$$\rho_c = \phi_c / f \tag{21}$$

$$\Theta_c = f - \phi_c = f \cdot (1 - \rho_c) \tag{22}$$

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

When compared to the pipeline analysis presented in [62], the main advantage of our approach is that we eliminate the non-determinism introduced by a separate cache analysis. In this way, *concrete arrival times of shared requests can be determined according to a program-specific iteration strategy.* The novelty of lifting the analytical model of $\mathcal{LR}$ servers into the semantic model of abstract interpretation is the possibility to prove the soundness of the predicted timing penalties when accessing shared resources by means of a Galois connection. The single-core pipeline analysis can be instrumented with an additional timing property, which is denoted by the variable d in the function *missed* given in Section 6.1, hence demonstrating the feasiblity and scalability of our approach.

The abstract semantic meaning of access times to shared resources is an upper bound for possible concrete, i.e. actual, access times. Due to the limited bandwidth of the shared bus, pipeline bubbles caused by structural hazards can be introduced when additional CPU cycles are required during a shared access, e.g. upon a cache miss. In such scenarios, the benefit of using the $\mathcal{LR}$ abstraction is the possibility to predict upper bounds for the delay required to complete an access to a shared resource. Sound approximations to these additional delays are defined as elements of the totally ordered set *UCycles*:

$$\mathit{TimedTask} \in (\mathit{Cycles} \times \mathit{UCycles} \times \mathit{Stage} \times \mathit{Task})$$

As mentioned in Section 4.1, the pipeline abstract domain is defined as a set of hybrid pipeline states, each including a "concrete" timing property *Cycles*. The new definition of *TimedTask* indicates that pipeline states include additional abstract timing properties. The purpose of modeling the $\mathcal{LR}$ abstraction by means of a Galois connection is to reduce the number of joins required by the interleaving semantics and, therefore, reduce the number of pipeline states. 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 in *UCycles*, which are computed using the $\mathcal{LR}$ model on control flows only.

From the observation of Figure 9(a), it is clear that the number of join operations is proportional to the number of architectural flows. However, Figure 9(b) shows that when applying the $\mathcal{LR}$ abstraction to compute sound upper bounds on the finishing times of shared requests, the number of joins is determined solely by the control flows of each process independently. Therefore, the analysis time is strongly reduced in the latter case because the static analyzer is able to predict a unique and sound timing property in each fixpoint iteration.

Let *UCycles* be an upper semilattice equipped with a total order $\leqslant$ on naturals $\mathbb{N}$, denoting abstract timing properties, and let $Cycles^\sharp$ be the powerset of the set *UCycles*. A Galois connection $Cycles^\natural(\subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} Delay^\sharp(\subseteq)$ is defined in terms of a *$\mathcal{LR}$-representation function* $\beta : Instr \mapsto UCycles$, which computes the timing penalty of shared accesses, in terms of CPI, when a particular instruction enters the pipeline. Given a subset $X \subseteq Cycles^\natural$ and an abstract property $p^\sharp \in Cycles^\sharp$, the abstraction and concretization maps are defined by:

$$\alpha(X) = \bigcup\{\beta(\omega_c^k) \mid \forall \omega_c^k : t_f(\omega_c^k) \in X\} \tag{23}$$

$$\gamma(p^\sharp) = \{t_f(\omega_c^k) \in Cycles \mid \forall \omega_c^k : \beta(\omega_c^k) \subseteq p^\sharp\} \tag{24}$$

where ω_c^k is the k^{th} instruction to fetch from the shared memory after a cache miss in processor core c. The actual (concrete) finishing time is denoted by $t_f(\omega_c^k)$ and the “best” abstraction $p^\sharp$ is denoted by a set containing a single element that is the finishing time given by Def. (20), where ω_c^k is an instruction. The notion of “best” abstraction (more precise) is given by the comparison of the elements of two singletons under the total order ($\leqslant$). Hence, the $\mathcal{LR}$-server model can be formally defined in terms of the representation function β as:

$$\beta(\omega_c^k) = \{\max(t_a(\omega_c^k) + \Theta_c, \hat{t}_f(\omega_c^{k-1})) + s(\omega_c^k)/\rho_c\} = \{\hat{t}_f(\omega_c^k)\} \tag{25}$$

Proof sketch. Assuming that the *initial* arrival times of shared accesses are independent, the predictability of the $\mathcal{LR}$ server formally abstracts from architectural flows by eliminating the variation of interference between processor cores. Since the finishing time of a shared access is soundly predicted, we have, by compositionality, that the maximum local timing property given by Def. (23), after joining ($\bigcup$) all abstract states across the architectural flows in Fig. 9(a), is exactly equal to the maximum local timing property when only the control flows are considered: $\alpha(X) = \{\hat{t}_f(\omega_c^k) \mid t_f(\omega_c^k) \in X\}$.

Given that α and γ are monotone functions on powersets of $\mathbb{N}$, the proof that the representation function β can be used to define the Galois connection $\langle\alpha, \gamma\rangle$ is obtained by satisfying the following closure properties [20]:

$$\gamma \circ \alpha \supseteq \lambda X.\{p \mid p \in X\} \tag{26}$$

$$\alpha \circ \gamma \subseteq \lambda p^\sharp.p^\sharp \tag{27}$$

The closure operators (26) and (27) prove that soundness is not lost by going back and forth between $Cycles^\natural$ and $Cycles^\sharp$, although loss of precision may occur. According to (19), the predicted upper bounds on finishing times (20) are affected by the arrival time plus by the worst-case latency Θ, or by the finishing time of the previous request from the processor core, whichever is greater. Hence, $\alpha \circ \gamma \subseteq \{\lambda\omega_c^k.\beta(\omega_c^k)\}$ always holds.

Conversely, by inspecting Fig. 10, we have that concrete finishing times are given by the “provided service” curve and that the corresponding upper bounds are given by the “service bound” curve. Put simply, after the elimination of variation in interference between processor cores, the starting times can be either $(t_a(\omega^k) + \Theta)$ or $(\hat{t_f}(\omega^{k-1}))$. Let $\hat{p}$ be the maximal timing property, i.e. the most precise upper bound, obtained by Def. (20). Hence, by the ordering $(\leqslant)$ on the singleton set, we have that $p \leqslant \hat{p}$ always holds, proving that $\gamma \circ \alpha$ is an upper closure. Therefore, $\hat{t}_f(\omega_c^k)$ is a safe approximation to the actual (concrete) finishing time $t_f(\omega_c^k)$, that is, $\{t_f(\omega_c^k)\} \subseteq \gamma(\alpha(\{t_f(\omega_c^k\}))$ holds. $\square$

6.3. Experimental Evaluation

This section experimentally evaluates our approach to WCET estimation in multicores using the $\mathcal{LR}$ abstraction. Example 7 starts by evaluating the results of WCET estimation for a multi-process toy example with a finite number of architectural flows using the specialized TDM analysis that relies on its static temporal partitioning of the resource. We then compare these results with those obtained when applying the corresponding $\mathcal{LR}$ abstraction for TDM arbitration. After, using the same configuration for the TDM arbiter and for the corresponding $\mathcal{LR}$-server, we present and discuss the WCET results for a range of the Mälardalen benchmarks programs.

The objectives of these experiments are: 1) provide an example that supports the proof given in Section 6.2 and shows that the proposed compositional $\mathcal{LR}$ analysis is fast and scalable compared to analysis of all possible architectural flows; 2) demonstrate that the $\mathcal{LR}$ abstraction is more flexible than relying on the temporal isolation provided by TDM arbitration by supporting any arbiter in the class of $\mathcal{LR}$ servers just by changing two parameter values; and 3) show that the $\mathcal{LR}$ abstraction is efficient in terms of incurring limited overhead.

The implementation of the specialized TDM analysis used for comparison is straight-forward since it statically partitions the resource to achieve isolation. This means that the alignment between arriving requests and the TDM table is statically determined by dividing $t_a(\omega^k)$ by f and that the waiting time (if any) can be easily computed by knowing the number of consecutively allocated slots, ϕ_c, and their location in the TDM table. The request is considered to be finished one slot after being scheduled, which limits this analysis to a single outstanding request.

The experiments consider the simplified multicore system with two cores in Fig. 11, where instructions are located in a partitioned SRAM memory shared by a TDM arbiter. The TDM arbiter has a table size of 2 and the cores are allocated one slot each. This arbiter and configuration results in a $\mathcal{LR}$ server parametrized with $\Theta = 1$ and $\rho = 0.5$ according to Equations (21) and (22), respectively. Note that these are the same parameters as if a common Round Robin arbiter was used. For the considered SRAM memory, a TDM slot corresponds to a single clock cycle. The applications used in the experiment are the well-known Mälardalen WCET benchmark programs [33]. By compositionality of the $\mathcal{LR}$ abstraction and assuming that each processor core has a sufficiently large private data memory, each program is analyzed independently from the program running on the other core.

Example 7. Example of timing analysis of architectural flows
In order to support the soundness proof of the $\mathcal{LR}$-abstraction, this example shows the results of timing analysis for a very simple program containing one single branch instruction. We consider that both the *true* and *false* program paths of the conditional branch consist of straight-line assembler code, to which we refer to as 'application A" and "application X", respectively. The purpose of the experiment is to analyze these alternative paths in a multicore setting, where the the branch instruction simulates the procedure *fork*. Accordingly, the WCET is computed for all interleavings between instructions of "application A" and "application X" [58].

From the observation of Table 5, we conclude that the analysis of interleavings in undecidable. As expected, the number of interleavings grows exponentially with the number of instructions, and the timing analysis of the two toy applications only with a few instructions each take almost ten minutes to finish. Failure to analyze such toy examples in reasonable time clearly shows that although the accuracy of the approach is excellent, the *analysis of architectural flows does not scale to realistic applications.*

Alternatively, 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 in order to compute safe WCET estimates. Also, since the timing analysis using the $\mathcal{LR}$-abstraction only considers control flows, the analysis time when using the compositional $\mathcal{LR}$-server model is approximately equal to that of TDM. Concerning precision, the results show that composable TDM bounds the WCET results computed by the timing analysis of architectural flows. This fact is a consequence of the context-sensitivity of the pipeline analysis that can significantly change the number of the compute intermediate hardware states, thereby affecting precision. As expected, we also observe that the WCET computed using the $\mathcal{LR}$-server model are an upper bound of the WCET computed when using TDM.

Table 5: Results for architectural flows, composable TDM and compositional $\mathcal{LR}$-server

No. instrs. in app. A'	No. instrs. in app. X	No. of interleavings	Results (CPI/sec.)	Arch. Flows (TDM)	Composable TDM	Compositional $\mathcal{LR}$-server
4	5	126	WCET	179	185	197
			Analysis Time	57.0	0.17	0.19
5	5	252	WCET	188	188	205
			Analysis Time	140.3	0.18	0.2
6	5	462	WCET	195	195	216
			Analysis Time	588.7	0.43	0.19

The results demonstrate that timing analysis for WCET estimation is scalable when considering the abstraction provided by the $\mathcal{LR}$-server model. As previously explained in Example 4, the reason for this is the fact that abstract pipeline states for straight-line programs are singleton sets. Nevertheless, the following evaluation of WCET analysis for a range of benchmark programs allows us to conclude that timing analysis using the $\mathcal{LR}$-server model is scalable in general and to demonstrate that *sound WCET estimates can be efficiently* *computed in a multicore setting by applying the $\mathcal{LR}$ abstraction to architectural flows in order to efficiently analyze multi-process applications independently.* ▲

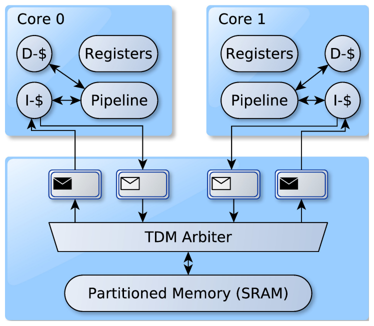


Figure 11: Simplified multicore architecture

Next, we proceed by quantifying the overhead of the $\mathcal{LR}$ abstraction for the considered system by comparing the results to the specialized TDM analysis. The results of this experiment are shown in Table 6. Note that the results of this experiment 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. the 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. Hence, the delay overhead of the $\mathcal{LR}$ abstraction is maximized compared to the specialized analysis in this experiment.

The results in Table 6 show the overhead of the $\mathcal{LR}$ abstraction ranges between 9.2% and 12.1% for the considered arbiter, configuration, and applications. This is partly because the use of a small TDM table size reduces the penalty of starting a new busy period for every cache miss through the low $\Theta = 1$ value, but also because the case of an SRAM shared by a TDM arbiter is quite predictable and captured well by the abstraction. A more complex case with DRAM and Credit-Controlled Static-Priority arbitration is shown in [64] along with an optimization to reduce the overhead of the abstraction for resources with non-preemptive accesses. For that more dynamic and unpredictable case, the $\mathcal{LR}$ abstraction incurs an overhead of 40% once the proposed optimizations have been applied. In terms of analysis time, both the specialized TDM analysis and the $\mathcal{LR}$-server are approximately the same, both taking between a second to a minute for the applications in the benchmark suite. *This shows that our approach is scalable to the abstraction of interleaving semantics at the expense of loss of precision that depends on the predictability of the resource and its arbitration.*

More precise WCET estimates can be obtained with the $\mathcal{LR}$ abstraction 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 and hence number of times a processor core suffers maximum interference from other cores, Θ, but it would also increase the complexity of the WCET analyzer. However, the main benefit of the $\mathcal{LR}$ abstraction is the possibility to perform compositional timing analysis using *any* arbiter belonging to the class, as opposed to only TDM, by only replacing Equations (21) and (22) with those of the chosen arbiter. For example, this enables the use of priority-based arbitration mechanisms, such as Credit-Controlled Static-Priority, that reduce the memory delay of latency-sensitive applications.

Table 6: 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. ($\approx$)
bs	152	1162	111	1036	10.8	2.3
bsort	156	1459	152	1311	10.1	0.9
cnt	145	1291	175	1157	10.4	0.8
cover	111	787	105	699	11.2	3.9
crc	459	3153	304	2820	10.4	15.0
expint	251	2121	233	1906	10.1	1.9
fdct	1011	10890	720	9886	9.2	20.1
fibcall	111	985	59	877	10.7	2.3
matmult	287	2558	188	2322	9.2	5.2
minmax	221	983	263	864	12.1	2.6
prime	232	1055	196	937	11.1	5.2
ud	418	3943	97	3464	12.1	40.0

7. Conclusions

The main objective of this article is to provide a formal approach to worst-case execution time (WCET) safety verification in the context of the Abstract Carrying Code (ACC) framework. The static analysis by abstract interpretation is supported in the automatic generation of combinatory fixpoint algorithms, by means of an intermediate graph language, that are bound to a specific program and to an iteration strategy. The definition of a relational/algebraic intermediate graph language allows topological transformations on dependency graphs with the objective of making the generation of fixpoint checkers highly efficient in terms of verification time and reduce the certificate size. Experimental results show a gain in verification time of 78% on average and an overall certificate size reduction of 2.33 times the size of the original certificate.

As opposed to state-of-the-art WCET analysis, we relax the integer linear optimization problem (ILP) to rational arithmetic linear programming (LP) by proving that the integer semantics of our ILP formalization is preserved and no loss of precision is introduced. In this way, the complexity of the optimization problem on the supplier side is reduced from NP-hard to P and the checking of linear solutions on the consumer side is performed using the certifying properties of duality theory in linear time. In combination with the computational efficiency of specialized chaotic fixpoint algorithms, the verification of the WCET by means of simple algebra computations is significantly more efficient than the computation of the WCET on supplier sites. This can enable the design of more robust and safer embedded systems, mainly those that are distributed across networks and have real-time safety requirements.

Additionally, we integrate the analytical model of *latency-rate* ($\mathcal{LR}$) servers into the semantic framework of abstract interpretation in order to design a WCET analyzer capable to surpass the intrinsic computational complexity of timing analysis of multiple processing cores sharing common resources. Although the considered multicore architecture is rather simplified, the results show that our solution for WCET analysis on multicores can be easily parameterized with an abstraction of the timing behavior of any arbiter for shared resources belonging to the class of $\mathcal{LR}$-servers and that the resulting analysis and verification times preserve the feasibility and scalability of the single-core timing analysis. Although the $\mathcal{LR}$ abstraction is proved sound by a Galois connection, the prediction of accesses to shared resources introduces an over-approximation that is about 10.6% on average in our experiments.

Acknowledgments

Part of this work by the first author was done at the Laboratoire d'Informatique Fondamentale d'Orléans (LIFO), University of Orleans, France. This work was partially supported by the Ministry of Education of the Czech Republic under the project "Support for improving R&D teams and the development of intersectoral mobility at CTU in Prague" number CZ.1.07/2.3.00/30.0034.

8. References

[1] AbsInt, 2014. Angewandte informatik. `http://www.absint.com/pag/`.

[2] Akesson, B., Goossens, K., 2011. Architectures and modeling of predictable memory controllers for improved system integration. In: Design, Automation & Test in Europe Conference & Exhibition. IEEE, pp. 1–6.

[3] Akesson, B., Steffens, L., Strooisma, E., Goossens, K., Aug 2008. Real-time scheduling using credit-controlled static-priority arbitration. In: Embedded and Real-Time Computing Systems and Applications, 2008. RTCSA '08. 14th IEEE International Conference on. pp. 3–14.

[4] Akesson, B., Minaeva, A., Sucha, P., Nelson, A., Hanzalek, Z., 2015. An Efficient Configuration Methodology for Time-Division Multiplexed Single Resources. In: Proc. of Real-Time and Embedded Technology and Applications Symposium. RTAS'15. pp. 161–171.

[5] Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D., 2007. Cost analysis of java bytecode. In: De Nicola, R. (Ed.), Programming Languages and Systems. Vol. 4421 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 157–172.

[6] Albert, E., Arenas, P., Puebla, G., 2006. An incremental approach to abstraction-carrying code. In: Hermann, M., Voronkov, A. (Eds.), Logic for Programming, Artificial Intelligence, and Reasoning. Vol. 4246 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 377–391.

[7] Albert, E., Arenas, P., Puebla, G., Hermenegildo, M. V., 2012. Certificate size reduction in abstraction-carrying code. TPLP 12 (3), 283–318.

[8] Albert, E., Puebla, G., Hermenegildo, M., 2005. An abstract interpretation-based approach to mobile code safety. Electronic Notes Theoretical Computer Science. 132 (1), 113–129.

[9] Albert, E., et al., 2004. Abstraction-carrying code. In: Logic for Programming, Artificial Intelligence, and Reasoning. LPAR'04. pp. 380–397.

[10] Allen, F. E., Jul. 1970. Control flow analysis. SIGPLAN Not. 5 (7), 1–19.

[11] Ballabriga, C., Cassé, H., Rochange, C., Sainrat, P., 2010. Otawa: An open toolbox for adaptive WCET analysis. In: Proc. of the 8th nternational Conference on Software Technologies for Embedded and Ubiquitous Systems. SEUS'10. Springer-Verlag, Berlin, Heidelberg, pp. 35–46.

[12] Barthe, G., Beringer, L., Crégut, P., Grégoire, B., Hofmann, M., Müller, P., Poll, E., Puebla, G., Stark, I., Vétillard, E., 2007. Mobius: mobility, ubiquity, security objectives and progress report. In: Proc. of the 2nd international conference on Trustworthy global computing. TGC'06. Springer-Verlag, Berlin, Heidelberg, pp. 10–29.

[13] Besson, F., Cachera, D., Jensen, T., Pichardie, D., 2009. Certified static analysis by abstract interpretation. In: Foundations of Security Analysis and Design V. Tutorial Lectures. Springer-Verlag, Berlin, Heidelberg, pp. 223–257.

[14] Besson, F., Jensen, T., Pichardie, D., Turpin, T., 2010. Certified result checking for polyhedral analysis of bytecode programs. In: Proc. of the 5th International Conference on Trustworthly Global Computing. TGC'10. Springer-Verlag, Berlin, Heidelberg, pp. 253–267.

[15] Bourdoncle, F., 1993. Efficient chaotic iteration strategies with widenings. In: Proc. of the International Conference on Formal Methods in Programming and their Applications. Springer-Verlag, pp. 128–141.

[16] Cousot, P., 1999. The calculational design of a generic abstract interpreter. In: Broy, M., Steinbrüggen, R. (Eds.), Calculational System Design. NATO ASI Series F. IOS Press, Amsterdam.

[17] Cousot, P., Apr. 2002. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. Theoretical Computer Science 277 (1-2), 47–103.

[18] Cousot, P., Cousot, R., 1976. Static determination of dynamic properties of programs. In: Proc. of the Second International Symposium on Programming. Dunod, Paris, France, pp. 106–130.

[19] Cousot, P., Cousot, R., 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. POPL '77. ACM, New York, NY, USA, pp. 238–252.

[20] Cousot, P., Cousot, R., 1992. Abstract interpretation and application to logic programs. Journal of Logic Programming 13 (2–3), 103–179.

[21] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Min, A., Monniaux, D., Rival, X., 2005. The ASTRÉE analyzer. In: Programming Languages and Systems, Proc. of the 14th European Symposium on Programming, volume 3444 of Lecture Notes in Computer Science. Springer, pp. 21–30.

[22] Cousot, P., Halbwachs, N., 1978. Automatic discovery of linear restraints among variables of a program. In: Proc. of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. POPL '78. ACM, New York, NY, USA, pp. 84–96.

[23] Cruz, R., 1991. A calculus for network delay. I. Network elements in isolation. IEEE Transactions on Information Theory 37 (1), 114–131.

[24] Cullmann, C., Ferdinand, C., Gebhard, G., Grund, D., Maiza, C., Reineke, J., Triquet, B., Wegener, S., Wilhelm, R., September 2010. Predictability considerations in the design of multi-core embedded systems. Ingénieurs de l'Automobile 807, 36–42.

[25] Cullmann, C., Martin, F., 2007. Data-Flow Based Detection of Loop Bounds. In: Rochange, C. (Ed.), 7th International Workshop on Worst-Case Execution Time Analysis (WCET'07). Vol. 6 of OpenAccess Series in Informatics (OASIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany.

[26] de Michiel, M., Bonenfant, A., Casse, H., Sainrat, P., Aug 2008. Static loop bound analysis of C programs based on flow analysis and abstract interpretation. In: Embedded and Real-Time Computing Systems and Applications, 2008. RTCSA '08. 14th IEEE International Conference on. pp. 161–166.

[27] Coq development team. The coq proof assistant. `coq.inria.fr`, 1989-2014.

[28] Ermedahl, A., Gustafsson, J., 1997. Deriving annotations for tight calculation of execution time. In: Proc. of the Third International Euro-Par Conference on Parallel Processing. Euro-Par '97. Springer-Verlag, London, UK, pp. 1298–1307.

[29] Ermedahl, A., Gustafsson, J., Lisper, B., July 2011. Deriving WCET bounds by abstract execution. In: Healy, C. (Ed.), Proc. 11th International Workshop on Worst-Case Execution Time (WCET) Analysis. Austrian Computer Society (OCG).

[30] Ferdinand, C., Heckmann, R., 2004. ait: Worst-case execution time prediction by static program analysis. In: Jacquart, R. (Ed.), Building the Information Society. Vol. 156 of IFIP International Federation for Information Processing. Springer Boston, pp. 377–383.

[31] Ferdinand, C., Wilhelm, R., 1998. On predicting data cache behavior for real-time systems. In: Proc. of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems. LCTES '98. Springer-Verlag, London, UK, UK, pp. 16–30.

[32] Gibbons, J., 1999. A pointless derivation of radixsort. Journal of Functional Programming 9 (3), 339–346.

[33] Gustafsson, J., Betts, A., Ermedahl, A., Lisper, B., 2010. The Mälardalen WCET benchmarks: Past, present and future. In: Lisper, B. (Ed.), WCET. Vol. 15 of OASICS. pp. 136–146.

[34] Gustafsson, J., Ermedahl, A., Sandberg, C., Lisper, B., 2006. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In: Proc. of the 27th IEEE International Real-Time Systems Symposium. RTSS '06. IEEE Computer Society, Washington, DC, USA, pp. 57–66.

[35] Hansson, A., et al., 2009. Enabling application-level performance guarantees in network-based systems on chip by applying dataflow analysis. IET Computers & Digital Techniques, 398–412.

[36] Hillier, F. S., Lieberman, G. J., 1986. Introduction to operations research, 4th ed. Holden-Day, Inc., San Francisco, CA, USA.

[37] Hoffman, A., Kruskal, J., 2010. Integral boundary points of convex polyhedra. In: Jünger, M., Liebling, T. M., Naddef, D., Nemhauser, G. L., Pulleyblank, W. R., Reinelt, G., Rinaldi, G., Wolsey, L. A. (Eds.), 50 Years of Integer Programming 1958-2008. Springer Berlin Heidelberg, pp. 49–76.

[38] Kildall, G., 1973. A unified approach to global program optimization. In: Proc. of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages. POPL'73. ACM, New York, NY, USA, pp. 194–206.

[39] Kim, H., de Niz, D., Andersson, B., Klein, M., Mutlu, O., Rajkumar, R. R., 2014. Bounding memory interference delay in cots-based multi-core systems. In: Proc. of the Twentieth IEEE Real-Time Technology and Applications Symposium. RTAS '14.

[40] Kleene, S. C., 1952. Introduction to metamathematics. Van Nostrand.

[41] Lisper, B., 2003. Fully automatic, parametric worst-case execution time analysis. In: Proceedings of the 3rd International Workshop on Worst-Case Execution Time Analysis, WCET 2003 - a Satellite Event to ECRTS 2003, Polytechnic Institute of Porto, Portugal, 2003, pp. 99–102.

[42] Lisper, B., 2014. Sweet - A tool for WCET flow analysis (extended abstract). In: Margaria, T., Steffen, B. (Eds.), Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications. Vol. 8803 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 482–485.

[43] Lundqvist, T., Stenström, P., 1998. Integrating path and timing analysis using instruction-level simulation techniques. In: Proc. of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems. LCTES '98. Springer-Verlag, London, UK, UK, pp. 1–15.

[44] Manna, Z., 2003. Mathematical Theory of Computation. Dover Publications, Incorporated.

[45] Maroneze, A., Blazy, S., Pichardie, D., Puaut, I., 2014. A Formally Verified WCET Estimation Tool. In: Falk, H. (Ed.), 14th International Workshop on Worst-Case Execution Time Analysis. Vol. 39. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, pp. 11–20.

[46] Martin, F. M., Alt, M., Wilhelm, R., Ferdinand, C., 1998. Analysis of loops. In: Proc. of the 7th International Conference on Compiler Construction, volume 1383 of Lecture Notes in Computer Science. Springer-Verlag, pp. 80–94.

[47] McConnell, R. M., Mehlhorn, K., Näher, S., Schweitzer, P., 2011. Certifying algorithms. Computer Science Review 5 (2), 119–161.

[48] Morrisett, G., Walker, D., Crary, K., Glew, N., May 1999. From system F to typed assembly language. ACM Trans. Program. Lang. Syst. 21, 527–568.

[49] Muthukumar, K., Hermenegildo, M. V., 1991. Combined determination of sharing and freeness of program variables through abstract interpretation. In: Furukawa, K. (Ed.), ICLP. MIT Press, pp. 49–63.

[50] Necula, G. C., 1997. Proof-carrying code. In: Proc. of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL '97. ACM, New York, NY, USA, pp. 106–119.

[51] Nemhauser, G. L., Wolsey, L. A., 1988. Integer and Combinatorial Optimization. Wiley-Interscience, New York, NY, USA.

[52] Nielson, F., Nielson, H. R., 1985. Code generation from two-level denotational meta-languages. In: On Programs as data objects. Springer-Verlag New York, Inc., New York, NY, USA, pp. 192–205.

[53] Nielson, F., Nielson, H. R., Hankin, C., 1999. Principles of Program Analysis. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[54] Nielson, H. R., Nielson, F., 1986. Pragmatic aspects of two-level denotational meta-languages. In: Proc. of the European Symposium on Programming. ESOP '86. Springer-Verlag, London, UK, pp. 133–143.

[55] Puschner, P. P., Schedl, A. V., Jul. 1997. Computing maximum task execution times - a graph-based approach. Real-Time Systems 13 (1), 67–91.

[56] Reineke, J., Grund, D., Berg, C., Wilhelm, R., Aug. 2007. Timing predictability of cache replacement policies. Real-Time Systems 37 (2), 99–122.

[57] Rodrigues, V., 2013. Semantics-based program verification: an abstract interpretation approach. Ph.D. thesis, Department of Computer Science, Faculty of Sciences, University of Porto.

[58] Rodrigues, V., Akesson, B., de Sousa, S. M., Florido, M., 2013. A declarative compositional timing analysis for multicores using the latency-rate abstraction. In: Fifteenth International Symposium on Practical Aspects of Declarative Languages (PADL'13). Springer, pp. 43–59.

[59] Rodrigues, V., Florido, M., de Sousa, S. a. M., 2011. A functional approach to worst-case execution time analysis. In: Proc. of the 20th international conference on Functional and constraint logic programming. WFLP'11. Springer-Verlag, Berlin, Heidelberg, pp. 86–103.

[60] Rodrigues, V., Pedroso, J. a. P., Florido, M., de Sousa, S. a. M., 2012. Certifying execution time. In: Proc. of the Second international conference on Foundational and Practical Aspects of Resource Analysis. FOPARA'11. Springer-Verlag, Berlin, Heidelberg, pp. 108–125.

[61] Rogers, Jr., H., 1987. Theory of Recursive Functions and Effective Computability. MIT Press, Cambridge, MA, USA.

[62] Schneider, J., Ferdinand, C., May 1999. Pipeline behavior prediction for superscalar processors by abstract interpretation. SIGPLAN Not. 34, 35–44.

[63] Seshia, S. A., Kotker, J., 2011. Game time: A toolkit for timing analysis of software. In: Proc. of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Part of the Joint European Conferences on Theory and Practice of Software. TACAS'11/ETAPS'11. Springer-Verlag, Berlin, Heidelberg, pp. 388–392.

[64] Shah, H., Knoll, A., Akesson, B., 2013. Bounding sdram interference: detailed analysis vs. latency-rate analysis. In: Proc. of the Conference on Design, Automation and Test in Europe. DATE '13. EDA Consortium, San Jose, CA, USA, pp. 308–313.

[65] Sriram, S., Bhattacharyya, S., 2000. Embedded multiprocessors: Scheduling and synchronization. CRC.

[66] Steine, M., Bekooij, M., Wiggers, M., Aug 2009. A priority-based budget scheduler with conservative dataflow model. In: Digital System Design, Architectures, Methods and Tools, 2009. DSD '09. 12th Euromicro Conference on. pp. 37–44.

[67] Stiliadis, D., Varma, A., 1998. Latency-rate servers: a general model for analysis of traffic scheduling algorithms. IEEE/ACM Transactions on Networking 6 (5), 611–624.

[68] Stoy, J. E., 1977. Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. MIT Press, Cambridge, MA, USA.

[69] Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V., 1999. Soot - a java bytecode optimization framework. In: Proc. of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research. CASCON '99. IBM Press, pp. 13–.

[70] Vink, J., van Berkel, K., van der Wolf, P., 2008. Performance analysis of SoC architectures based on latency-rate servers. In Proc. Design, Automation and Test in Europe Conference and Exhibition, 200–205.

[71] Wiggers, M. H., Bekooij, M. J. G., Smit, G. J. M., 2007. Modelling run-time arbitration by latency-rate servers in dataflow graphs. In: Proc. of the 10th international workshop on Software & compilers for embedded systems. SCOPES '07. ACM, New York, NY, USA, pp. 11–22.

[72] Wilhelm, R., 2003. Why AI + ILP is good for WCET, but MC is not, nor ILP alone. In: Steffen, B., Levi, G. (Eds.), Verification, Model Checking, and Abstract Interpretation. Vol. 2937 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 309–322.

[73] Wilhelm, R., Wachter, B., 2008. Abstract interpretation with applications to timing validation. In: Proc. of the 20th international conference on Computer Aided Verification. CAV'08. Springer-Verlag, Berlin, Heidelberg, pp. 22–36.

[74] Wilhelm, R., et al., May 2008. The worst-case execution-time problem - overview of methods and survey of tools. ACM Trans. Embed. Comput. Syst. 7 (3), 36:1–36:53.

[75] Wu, Z. P., Krish, Y., Pellizzoni, R., 2013. Worst case analysis of dram latency in multi-requestor systems. In: Real-Time Systems Symposium. RTSS'13. 2013 IEEE 34th. pp. 372–383.