A Memory Interference Analysis using a Formal Timing Analyzer (WIP)*

Mihail Asavoae
Oumaima Matoussi
Asmae Bouachtala
Hai-Dang Vu
Mathieu Jan
Université Paris-Saclay, CEA, List
Palaiseau, France

Abstract

Safety-critical applications require well-defined and documented timing behavior. These requirements shape the design and implementation of a timing analyzer based on a formal Instruction-Set Architecture (ISA) semantics and formal micro-architecture models. In this paper we present the key elements of such a timing analyzer and how to systematically combine the formal components to address timing properties such as evaluating memory interferences. We also report preliminary experiments of memory interference analysis of multi-threaded applications in a multicore context.

CCS Concepts: • **Computer systems organization** → *Real-time system architecture.*

Keywords: formal methods, timing analysis, multicores

ACM Reference Format:
Mihail Asavoae, Oumaima Matoussi, Asmae Bouachtala, Hai-Dang Vu, and Mathieu Jan. 2022. A Memory Interference Analysis using a Formal Timing Analyzer (WIP). In *Proceedings of the 23rd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '22), June 14, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3519941.3535077

1 Introduction

The design, implementation and verification of embedded real-time applications are subject to careful considerations of both functional and non-functional nature. Addressing the latter requires accurate knowledge of the underlying execution platform since a non-functional evaluation aims to determine bounds on computational resources. Of particular interest is the execution time which, for certain types of embedded applications (i.e. deemed safety-critical), is approached with a worst-case mindset. The worst-case execution time (WCET) analysis computes safe upper bounds on the execution paths of an input application on a given execution platform.

There are two complementary approaches to derive WCET bounds, using static analysis [1] and using end-to-end measurements [5]. Both approaches produce (desirably safe and accurate) timing bounds. A static timing analysis computes a bound under all possible program executions, using an over-approximation of the program behavior (i.e. the set of executions). A measurement-based timing analysis computes a bound under a set of actual program executions, using an under-approximation of the program behavior. Thus, it should be beneficial to combine the program coverage brought by a static timing analysis with the accuracy of a measurement-based one, into a timing analyzer which could mitigate these trade-offs. One such timing analyzer could be designed over a formal instruction set architecture (ISA) semantics and formal micro-architecture models.

The main motivation behind these design decisions (e.g. having a formal ISA as a good starting point) comes from the aforementioned observations, that of reasoning about program executions. By definition, a formal language semantics encodes all the possible executions, of any program written in the particular language. In case of a timing analyzer, the program at hand is at binary-level and the formal semantics of interest is that of the ISA. The fact that the infrastructure of a timing analyzer is formal implies the availability of interpreters and state space exploration tools. A formal ISA semantics could strengthen, through concrete interpretations, the over-approximation bounds which are computed by a static timing analysis. Also, a formal ISA semantics could be used to extrapolate, through symbolic state space exploration, the under-approximation bounds which are computed by a measurement-based timing analysis. A

*This research was partially supported by the ECSEL-JU under the program ECSEL-Innovation Actions-2018 (ECSEL-IA) for research project CPS4EU (ID-826276) in the area Cyber-Physical Systems.

LCTES '22, June 14, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9266-2/22/06…$15.00
https://doi.org/10.1145/3519941.3535077

similar argumentation stands for formal micro-architecture models: the available interpreters and state-space explorers are de-facto cycle-accurate micro-architecture tool support. In this paper, we present preliminary results on how to engineer such a timing analysis tool.

2 Formal Infrastructure

2.1 ISA Semantics Model - $\mathcal{S}_{ISA}$

Our formal ISA defines, for each instruction, two functions: `decode` and `execute`. The decoding semantics of an instruction i is a mapping between the instruction representation (i.e. mnemonic, operands) to its bit encoding: $\texttt{decode} : i \leftrightarrow bits(size)$, where $size = \{32, 64, 96\}$ bits. The state configuration $\langle PC, R, M\rangle_{ISA}$ consists of program counter (PC), register file (R) and memory (M) with $R = Map[rn \mapsto bits(64)]$, rn is the register name and $M = Map[(addr, w) \mapsto valu]$, with $addr$ is the address of size w bits and $valu$ is the respective memory value.

The proposed ISA model does not only serve as formal specification of the ISA but it also enables an instruction-accurate execution of any given well-defined program. A program is a sequence of instructions $prog = \{(addr, i)\}$; each instruction i is paired with its memory address $addr$. The execution of a program consists in executing the semantics of i. Concretely, the top-level execution loop of a program is a straightforward *fetch-execute* loop that fetches the instructions pre-loaded in the program memory, according to the address given by the PC, and invokes from the formal ISA model the execution semantics corresponding to the fetched instructions. We denote the concrete formal ISA semantics by $\mathcal{S}_{ISA}$.

2.2 Formal Micro-architecture Model - $\mathcal{A}_{arch}$

We introduce next the formal semantics of kvx, a VLIW architecture based on a 7-stage pipeline [4], as shown in Figure 2. The pipeline frontend presents a prefetch buffer (PFB) and an instruction cache (IC) while the data memory accesses are through a write-buffer (WB) and a data cache (DC). The execution model of the kvx pipeline is encoded at bundle-level (i.e. up to 8-slot) and the execution model through the memory components is represented at address-level. We present next the semantics for the kvx pipeline, the PFB and the IC. We opt to detail the PFB semantics since we further evaluate its impact on the proposed interference analysis and to detail the abstract IC semantics since it provides some of the variabilities to be explored in an interference analysis.

Pipeline Semantics - $\mathcal{S}_{pipe}$. The pipeline model advances a set of independent instructions forming the so-called bundle simultaneously (i.e. as one large instruction) across its stages. This means that each pipeline stage is able to process up to 8 syllables (a syllable is a 32-bit word), which is the maximum bundle size, in parallel. A pipeline stage communicates with its upstream and downstream adjacent stages by sending a stalling/un-stalling request and passing data (e.g. a decoded bundle), respectively. In addition to their neighbor stages, the IF, ID and RR stages interact with the PFB through an interface by sending a bundle request with the address of the requested bundle, flushing the PFB in case of a branch, receiving a bundle if it is available and a stalling request if it is not. As for the operation specific to each stage, it is provided by the ISA semantics model. For instance, the ID stage uses the `decode` semantics and an EX stage leverages the `execute` semantics of an instruction. We denote the concrete semantics of the kvx pipeline by $\mathcal{S}_{pipe}$.

PFB Semantics - $\mathcal{S}_{PFB}$. The PFB is responsible to feed instruction bundles to the pipeline. In short, the PFB of kvx keeps 4 FIFOs of 3x32bit size, receives 4x32bit packet from the IC each clock cycle and is flushed whenever a branch is taken. Moreover, the timing penalty of a PFB flush is 1 cycle unless the branching is to a bundle spanning two IC lines. In this case, the penalty is two clock cycles. The timing behavior of the PFB (i.e. more complex than exemplified above) is defined by the interplay between the pipeline demands from ID and even RR stages and the IC content.

The state configuration $\langle C, P, D, cy\rangle_{PFB}$ has the following elements: C encodes the interface between IC and PFB, P is the PFB content, D encodes the interface between PFB and the pipeline and cy represents the execution cycle. More precisely, $P = Map[f_i \mapsto c_i]$, is the PFB with FIFOs $f_i, i = 1..4$ and their respective content c_i. Also, C has fields hm to denote the hit/miss type of IC access, req to represent a requested block (e.g. a branch target), fb to encode the bundle status and $blks$ to capture the memory blocks sent to PFB ($C = Fields\{hm, req, fb, blks\}$). The fb is necessary since the PFB is agnostic to bundle formation from the IC (i.e. the 4x32bit fetched block could represent a partial bundle, a complete bundle or spanning several bundles). As such, fb denotes in which of the three bundle category a fetched block from IC belongs. Finally, $D = Fields\{id, rr, st, blks\}$ has fields id and rr for taken branches from ID and respectively RR stages to signal the PFB flush. Also, st signals pipeline stalling and $blks$ represents the blocks pushed into the pipeline.

A state of the PFB is (c, p, d), $c \in C, p \in P$ and $d \in D$, the set of states is S and the semantics of the PFB is defined in terms of a state transformer $T : S \rightarrow S$. We consider the following notations: *empty*, to overload an empty semantic entity (i.e. an empty map for the PFB or a zero memory address), $|s|$ to represent the size of s, _ to denote a placeholder, '.', a field access and '...' to stand for the unchanged/unused elements in $Fields$. Each clock cycle, a single state transformer $T_{next}[addr]$ is applied, yielding to the following definition over the pipeline progression.

Definition 1. *The concrete semantics of PFB, $\mathcal{S}_{PFB}$ is defined over the pipeline requests $P = List[addrs]$ as $\mathcal{S}_{PFB}(addr :: addrs) = T_{next}[addrs] \circ T_{next}[addr]$.*

IC Semantics - $\mathcal{A}_{IC}$. The instruction cache is connected to the PFB model to provide the requested blocks, either directly, i.e. cache hit, or through RAM, i.e. cache miss. Whereas

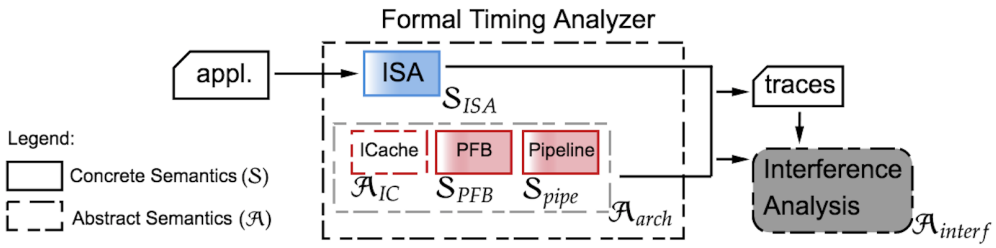

Figure 1. Hierarchy of abstract and concrete semantics.

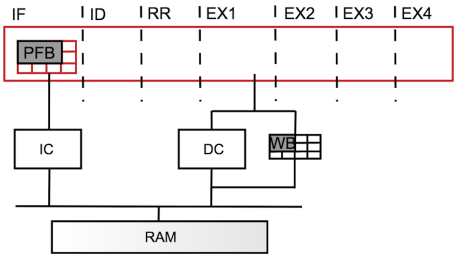

Figure 2. The kvx architecture: pipeline and memory model

a formal concrete cache semantics could be defined, in this paper we consider an abstract semantics $\mathcal{A}_{IC}$, defined by two classical cache abstractions in the context of the WCET analysis, i.e. for may- and must-, to predict cache misses and respectively cache hits. As such, $\mathcal{A}_{IC}[addr]$ establishes, for a memory block *addr*, if it is always a cache hit, always a cache miss or unclassified. In this latter case, a variable latency (for cache hit/miss) is assigned to *addr*. Intuitively, $\mathcal{A}_{IC}$ augments program executions with variable latencies due to IC accesses. These variable latencies are then propagated through the $\mathcal{S}_{PFB}$ and $\mathcal{S}_{pipe}$ to data memory accesses.

Definition 2. *The abstract semantics of IC, $\mathcal{A}_{IC}$ is defined over the pipeline requests $P = List[addrs]$ as $\mathcal{A}_{IC}(addr :: addrs) = asgn(addr) :: \mathcal{A}_{IC}(addrs)$, where $asgn(addr) = (addr, c)$, with $c = 1$ (hit), $c = mp$ (miss penalty) or $c = \{1, mp\}$ (for an unclassified access).*

3 Memory Interference Analysis

In the following, we propose a method to quantify memory interferences when a task system, developed as a multi-threaded application, is executed on a manycore architecture like MPPA. This algorithm considers the following elements: the mapping of tasks to cores (i.e. through threads) is a priori determined and a task is characterized by the longest execution path L (which is to be executed by our formal infrastructure). Precisely, L is determined by the worst input data of a task, provided, for example, by benchmarks.

Our application model is defined by n threads, where n is the number of cores, with each thread including one or several tasks, executed under periodic or aperiodic assumptions. Formally, the projection function φ is defined over timed traces, as executed on a the particular core, with two cases: $\varphi(L)$ whenever a task t is in execution and $\varphi(o)$ whenever the core is idle. $\varphi(L)$ is the bitvector of size equal to the execution of longest path L of task t where 1 stands for a data memory access and 0 for anything else. The position of 1s in $\varphi(L)$ represents the timestamp (i.e. cycle) of the memory access. $\varphi(o)$ is a bitvector of 0s with the size equal to the idle time (i.e. in cycles) of the core. Henceforth, a thread th is the concatenation of bitvectors corresponding to the computation of mapped tasks and idle times on the particular core, up to the hyperperiod of the task system.

Then, quantifying the memory interference would be based on the exhaustive, parallel exploration of threads $th_i, i = 1..n$, for n cores in order to determine the maximal number of concurrent data memory accesses (i.e. several 1s at the same timestamp). While the construction of a thread is through bitvector concatenation of $\varphi(L)s$ and $\varphi(o)s$, the Ls for the respective tasks are produced when running formal ISA semantics, $\mathcal{S}_{ISA}$ on an abstract architecture $\mathcal{A}_{arch}$. In short, the pipeline model $\mathcal{S}_{pipe}$ demands the first bundle (as sequences of *addr*), which is constructed in the IF stage and further decoded, instruction by instruction in the ID stage, by using the `decode` function of $\mathcal{S}_{ISA}$. These demands are served by the frontend memory model, namely PFB, IC and RAM. The $\mathcal{S}_{PFB}$, through its pipeline interface D (i.e. field *blks*) receives these *addr* and applies one or several transitions of T_{next} to either return the requested *addr* or advance the request to the IC interface C. Here, the field *hm* value is decided by the $\mathcal{A}_{IC}$, where standard may- and must-analyses would classify the correponding *addr*. Once the *addr* is returned to the pipeline, the `execute` function of $\mathcal{S}_{ISA}$ is applied. As such, the algorithm to evaluate the memory interferences is based, for each *addr*, on an abstraction $\mathcal{A}_{interf}$, defined as: $\mathcal{A}_{interf}(addr) = (\mathcal{A}_{arch} \circ \mathcal{S}_{ISA})(addr)$ with $\mathcal{A}_{arch} = \mathcal{S}_{pipe} \circ \mathcal{S}_{PFB} \circ \mathcal{A}_{IC}$.

We conduct preliminary experiments using two configurations for $\mathcal{A}_{arch}$ (with and without PFB) on the TACLe

Table 1. Runtime slowdown of the $\mathcal{S}_{PFB}$.

		without PFB		with PFB		
Benchmark	**#Bndl**	**Exec.**	**#Cyc.**	**Exec.**	**#Cyc.**	**Sl.**
adpcm_dec	14543	4.15	26952	5.48	25335	1.32
ammunition	22318	6.35	39354	7.16	35781	1.12
mpeg2	20069	5.80	35901	7.06	32661	1.21
statemate	15595	4.58	29284	5.89	27474	1.28
susan	27992	7.41	48018	8.77	42032	1.18

Table 2. Results of the Interference Analysis.

Tasks		without PFB			with PFB		
Thread	**Thread_type**	**#Max_th_size.**	**Exec.**	**Inter (%)**	**#Max_th_size**	**Exec.**	**Inter (%)**
mpeg2 susan	periodic	1446460	0.086	13	1334360	0.079	12
ammunition susan	aperiodic	1729042	0.090	10	1535938	0.089	11
statemate petri	periodic	1204644	0.082	11	1150401	0.078	10
audiobeam ndes	aperiodic	1729042	0.090	10	1132790	0.089	11
adpcm_dec adpcm_enc	periodic	1109973	0.081	13	1065734	0.077	12

benchmarks [6]. The $\mathcal{A}_{IC}$ could be provided by static analysis tools like Otawa [1]. In the experimentation we aim to evaluate mostly the execution of the formal infrastructure on both software and hardware models (e.g. PFB). For example, in Table 1, we present the runtime impact of the $\mathcal{S}_{PFB}$. The benchmarks have a number of bundles *#Bndl* with reported runtime *Exec* (in seconds) and resulting cycles *#Cyc* (of our cycle-accurate model) for both configurations. Finally, *Sl* reports the slowdown caused by $\mathcal{S}_{PFB}$. For example, the execution time of `adpcm_dec` using the PFB is 1.32 × slower than without the PFB. We could also observe that the execution using a PFB is faster.

Preliminary results of the interference analysis are in Table 2. In particular, we aim to determine runtime performance, i.e. columns *Exec*, for single interleavings of the considered threads (i.e. task system representation). The reported runtime performance is with respect to our timing analysis infrastructure, namely the cycle-accurate micro-architecture modeling and not to the MPPA platform. Columns *Thread* and *Thread_type* show the core mapping (e.g. first line - `mpeg2` is executed on a core and `susan` on another core) respectively the type of thread execution. For example, thread `mpeg2` is a concatenation of six $\varphi(L_{mpeg2})$ and five $\varphi(o)$ (of size 3000 cycles), while thread `susan` is a concatenation of three $\varphi(L_{susan})$s and two $\varphi(o)$ (of size 2000 cycles). Also, the aperiodic thread type is modeled by considering different bitvector size for the $\varphi(o)$s (e.g. values between 1000 cycles and 5000 cycles, for `audiobeam` and `ndes`). Column *max_th_size* is the size of the longest thread, in cycles and finally, *Interf* is the percentage of concurrent data memory accesses (thus potential interferences). The impact of the PFB on the results is not uniform, for example the presence of the PFB could lead to more (e.g. lines 3 or 5) or less potential interferences (e.g. lines 2 or 4). More specifically, for the thread with `ammunition` and `susan`, the shorter execution trace exhibits more memory interferences than longer one since our analysis counts the number of concurrent accesses. Under this consideration, the memory interference analysis, which considers at this point only the longest execution path (i.e. interleaving), is not conservative; an efficient exploration of all possible interleavings is left for future work.

4 Related Work

There is considerable work tackling the formalization of ISA specifications but each proposed formal ISA is usually created for a specific purpose, e.g. verifying the correctness of an embedded OS [11], extending a compiler [10], generating tests and verifying functional properties [3], etc. Therefore, only the ISA subset that caters for each specific purpose is modeled, leading to partial formal ISA models. In the context of extending the CompCert backend to support VLIW cores, the authors in [10] formalize the assembly semantics of `kvx` to verify the preservation of the original sequential semantics wrt. the proposed parallel semantics.

To demonstrate the usability of our formal ISA model, we targeted the problem of memory interference characterization. This problem has been investigated in previous work in the context of WCET analysis [8] and using tools such as [1] and [7], whose starting point is a high level representation of the software (usually a control flow graph - CFG). Some existing approaches like in [9] rely on analytical interference models under a timing compositionality assumption to account for contention delay. Other approaches [2, 5] analyze the changes in hardware performance counters, which are not always accurate, to instrument their interference models. These approaches yield pessimistic interference bounds because of their high-level and sometimes overly-simplified SW and/or HW models. We, however, aim for an accurate application representation using a formal and executable ISA, complemented with formal hardware models.

5 Future Work

The development of our formal timing analyzer is ongoing (i.e. the $\mathcal{S}_{ISA}$ covers most of the ALU, memory and branch instructions) Also, the $\mathcal{S}_{pipe}$ is complete but with ongoing integration with $\mathcal{S}_{PFB}$.The memory intereference analysis requires an efficient exploration of all possible interleavings for which an implementation with hash consing is in progress. This analysis also requires a validation infrastructure wrt. the cycle-accurate behavior of the actual platform.

References

[1] Clement Ballabriga, Cassé Hugues, and Rochange Christine. 2010. OTAWA: An Open Toolbox for Adaptive WCET Analysis. In *SEUS*.

[2] C. Courtaud, J. Sopena, G. Muller, and Pérez D. Gracia. 2019. Improving Prediction Accuracy of Memory Interferences for Multicore Platforms. In *RTSS*.

[3] Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S. Adve, and Grigore Rosu. 2019. A complete formal semantics of x86-64 user-level instruction set architecture. In *PLDI*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 1133–1148.

[4] Benoît Dupont de Dinechin. 2021. Engineering a Manycore Processor for Edge Computing. In *MECO*. IEEE, 1.

[5] E. Díaz, E. Mezzetti, L. Kosmidis, J. Abella, and Francisco J. Cazorla. 2018. Modelling Multicore Contention on the AURIXTM TC27x. In *DAC*.

[6] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sorensen, Peter Wägemann, and Simon Wegener. 2016. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In *WCET*, Martin Schoeberl (Ed.). 2:1–2:10.

[7] C. Ferdinand and R Heckmann. 2004. aiT: Worst-Case Execution Time Prediction by Static Program Analysis. *Building the Information Society. IFIP* (2004).

[8] E. Jakob, E. Andreas, and S. Mikael. 2003. Worst-case execution-time analysis for embedded real-time systems. In *STTT*.

[9] Altmeyer S., Davis Robert I., and Indrusiak L. 2015. A generic and compositional framework for multicore response time analysis. In *RTNS*.

[10] Cyril Six, Sylvain Boulmé, and David Monniaux. 2020. Certified and efficient instruction scheduling: application to interlocked VLIW processors. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 129:1–129:29.

[11] Jiawei Wang, Ming Fu, Lei Qiao, and Xinyu Feng. 2020. Formalizing SPARCv8 instruction set architecture in Coq. *Sci. Comput. Program.* 187 (2020), 102371.