METAMOC: MODULAR EXECUTION TIME ANALYSIS USING MODEL CHECKING

Andreas E. Dalsgaard[1], Mads Chr. Olesen[1], Martin Toft[1], René R. Hansen[1], Kim G. Larsen[1]

Abstract
Safe and tight worst-case execution times (WCETs) are important when scheduling hard real-time systems. This paper presents METAMOC, a path-based, modular method, based on model checking and static analysis, that determines safe and tight WCETs for programs running on platforms featuring caching and pipelining. The method works by constructing a UPPAAL model of the program being analysed and annotating the model with information from an inter-procedural value analysis. The program model is then combined with a model of the hardware platform, and model checked for the WCET. Through support for the platforms ARM7, ARM9 and ATMEL AVR 8-bit the modularity and retargetability of the method is demonstrated, as only the pipeline needs to be remodelled. Modelling the hardware is performed in a state-of-the-art graphical modeling environment. Experiments on the Mälardalen WCET benchmark programs show that taking caching into account yields much tighter WCETs, and that METAMOC is a fast and versatile approach for WCET analysis.

1. Introduction

Embedded software is virtually ubiquitous these days. It is used to control the proper functioning of technical devices we routinely use and rely on in our daily life. Often embedded software is applied in safety-critical systems — e.g. the braking system of a car or the steering gear of an airplane. Many of these safety-critical systems are also time-critical, meaning that the calculations performed by the tasks of an embedded system need not only be correct but must be carried out in a timely fashion. Worst-case execution time (WCET) analysis is concerned with providing guarantees for proper timing behaviour of system tasks by computing bounds for their execution time on given processors.

In order to allow for reliable and efficient scheduling of tasks, the scheduling algorithms need safe and tight WCETs. Two different classes of methods are predominant (see also [11]): *measurement-based methods*, where statistical information on WCETs is obtained by executing tasks on the given processor or simulator for a sample collection of input, and *static methods*, where static analysis (typically abstract interpretation and integer linear programming [10]) of the task, taking the specific hardware platform into account, allow the derivation of *safe* upper bounds on the execution time. The method presented in this paper, Modular Execution Time Analysis using Model Checking (METAMOC)[2], is a static method providing safe and tight WCET bounds, but utilising real-time model checking to establish WCETs. Figure 1 provides an overview of the prototype implementation of METAMOC.

Modern processors utilise techniques such as caching and pipelining, which increase the average number of operations that can be executed per time unit. Since these techniques are also found in

[1]Department of Computer Science, Aalborg University, Denmark
[2]`http://metamoc.martintoft.dk`

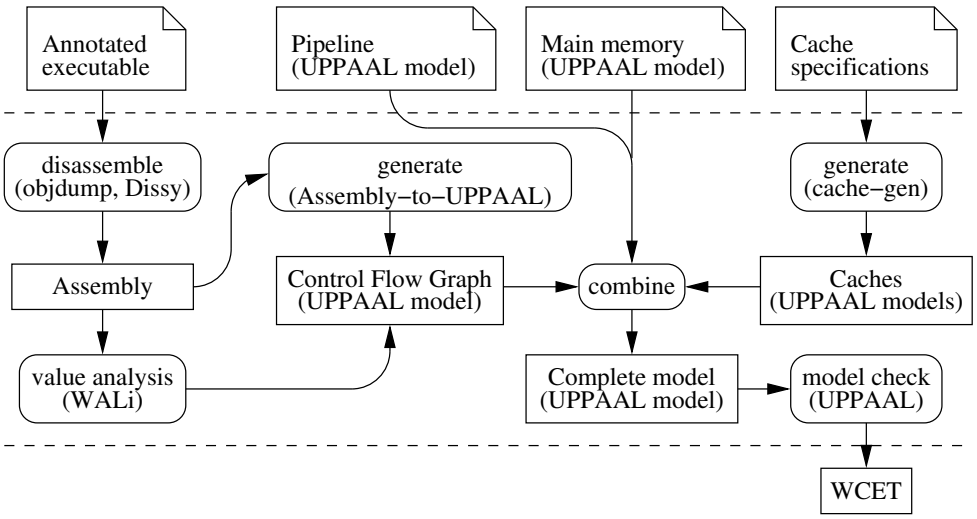

Figure 1. Overview of the prototype implementation of METAMOC. The top row shows required inputs. The executable (annotated with loop bounds) is the only user input, whereas the other inputs are platform specific models developed by specialists or hardware vendors. The output is a WCET estimate for running the executable on the hardware platform. Rounded and rectangular boxes represent actions and objects, respectively.

many processors intended for embedded devices, such as members of the widely deployed ARM7 and ARM9 families, a modern WCET analysis method must take them into account to be useful. The use of model checking in METAMOC provides a very modular approach for dealing with these techniques: the model to be analysed comprises an abstract model of the program, and similarly for the component models for the hardware platform, which include caches, pipelines and memories. Thus, WCET analysis of a platform with a new pipeline component, say, only requires a model for the new component.

The paper is organised as follows. Section 2 provides a brief introduction to the model checker UPPAAL [2] and its extensions to timed automata (TA). In Section 3 we describe the models used in METAMOC for hardware components and programs, and in which ways they interact. The modularity of the method is demonstrated through support for the platforms ARM7, ARM9 and ATMEL AVR 8-bit. Section 4 details a number of experiments, which evaluate the applicability and performance of METAMOC. The experiments are conducted using a suite of WCET benchmark programs from Mälardalen Real-Time Research Centre[3]. In Section 5 we give an overview of related work. Section 6 concludes the paper and presents possible directions for future work.

2. The UPPAAL Model Checker

UPPAAL [2] is a model checker for real-time systems which, besides the verification engine, features a state-of-the-art graphical user interface for modelling, simulation and verification. This section gives a brief introduction to UPPAAL models which is used as model formalism in METAMOC.

Systems in UPPAAL are modelled using an extension of timed automata (TA), which can be thought of as a finite automaton with a number of free-running clocks. Properties to be verified for the systems are formulated in a logic inspired by timed computation tree logic (TCTL). Besides standard TCTL UPPAAL also provides a special `sup` property, for finding the supremum of a clock. For example,

[3] `http://www.mrtc.mdh.se/projects/wcet/home.html`

the property "`sup: cyclecounter`" causes UPPAAL to determine an upper bound for the clock `cyclecounter`. For an introduction to TA model checking and TCTL see [1]. Rather than limiting a system to a single TA, UPPAAL uses a network of TA (NTA). Moreover, the TA are extended with a number of features to ease modelling. *Binary synchronisation channels* enable a TA having an edge labelled `name!` to synchronise with another TA having an edge labelled `name?`, i.e. they follow the edges together in one transition. If several pairs are possible, a pair is chosen non-deterministically. *Urgent channels* dictate that synchronisations must be carried out immediately when they are possible, i.e. a time delay must not occur. Another case where a time delay must not occur is when one or more of the TA are in a location marked as *committed*. *Priorities* can be assigned to TA, such that a transition in a TA is enabled only if no transitions in any higher priority TA are enabled.

3. Modelling Hardware Components and Programs

It is evident from Figure 1 that METAMOC is centered around a number of models. In this section we explain the ideas behind the models and how they fit together. Starting in the upper left corner of Figure 1, the method takes as input an executable annotated with loop bounds. The executable is disassembled using the tools objdump and Dissy[4], and the resulting assembly code is given as input to a generator and a value analysis. The generator creates a control flow graph (CFG) from the assembly code, in the form of a UPPAAL model, which is annotated with results from the value analysis. Besides the executable, the method takes as input a pipeline model, a main memory model and some cache specifications. The specifications are given as input to another generator, which creates cache models. Finally, the four models are combined and model checked, resulting in a WCET estimate for running the executable on the hardware platform. The CFG generator, the value analysis interface, the cache generator and the combine tool have been written for prototype implementation by the authors of this paper and are released as open source.

We use a prototype implementation of METAMOC for the ARM920T processor[5] as a continuing example in this section. The ARM920T processor is a member of the ARM9 family, which features an ARM9TDMI processor core[6], separate instruction and data caches, a memory management unit (MMU), and a bus interface for connecting main memory. We have modelled the core, the caches and a simple main memory. The core implements the ARM architecture v4T and contains a five stage pipeline with the stages fetch, decode, execute, memory and writeback. The communication between components in the modelled ARM920T is illustrated in Figure 2. In order to demonstrate the modularity of the method, we have utilised the ARM920T implementation to rapidly create implementations for the processors ARM7 and ATMEL AVR 8-bit. This process is detailed in Section 3.4.

In the following, a program is understood as a low-level machine executable representation, which has been disassembled to human readable assembly. The WCET of a program depends heavily on the hardware platform it is executed on, which explains why it is necessary to do the analysis at the lowest level; it is only at this level that enough information is present to determine the exact behaviour of the hardware platform.

[4] `http://www.gnu.org/software/binutils/` and `http://code.google.com/p/dissy/`
[5] `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0151c/ARM920T_TRM1_S.pdf`
[6] `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0180a/DDI0180.pdf`

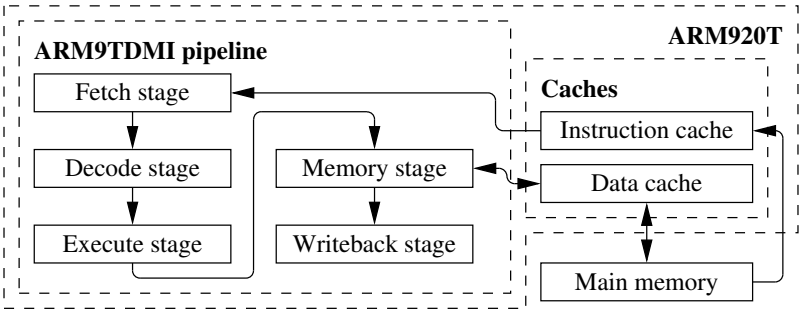

Figure 2. Communication between components in the ARM920T and the main memory.

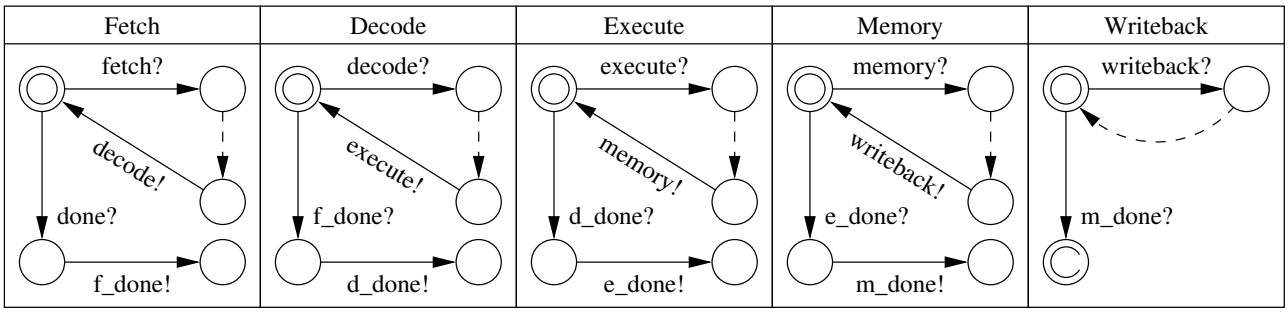

Figure 3. Sketch of the UPPAAL model for the pipeline in the ARM9TDMI processor core.

3.1. Modelling Pipelines

A pipeline is the part of a processor responsible for the execution of instructions. A pipeline works by dividing the execution of an instruction into a number of parallel stages, in order to increase the average pace of execution. The five stages found in the ARM9TDMI processor core are illustrated in Figure 2. The fetch stage fetches instructions from main memory through the instruction cache. The decode stage determine the instruction type and the involved registers, and prepares the needed values for the execute stage. The execute stage performs the actual arithmetic or logical computation. The memory stage performs access to main memory, through the data cache. Finally, the writeback stage writes computed values back into the registers. Each instruction flows through all stages, staying at least one cycle in each stage. The actual UPPAAL model for the decode stage is shown in Figure 4.

The parallel nature of a pipeline matches the parallel nature of a UPPAAL model. Figure 3 shows a sketch of the UPPAAL model for the pipeline. The model contains an automaton for each stage in the pipeline. Progress in the model is forced by declaring all the synchronisation channels as urgent, and time is bounded using a committed location in the writeback automaton. The non-determinism arising from the automata combinations is limited using priorities. The simulation ensures that a safe overapproximation of the execution time is found: e.g. since branch instructions are first evaluated in the execute stage in actual hardware, prior to entering the pipeline in METAMOC, special handling is required. While the hardware flushes the fetch and decode stages in case of a non-sequential branch, the fetch automaton in METAMOC performs two additional fetches in order to affect the instruction cache correctly.

Another example is pipeline stalls, which is handled in the decode automaton. The automaton initially delays for one cycle. Then, if the current instruction depends on data being loaded by the memory stage or data being shifted or sign extended by the writeback stage, it stalls until the data is ready.

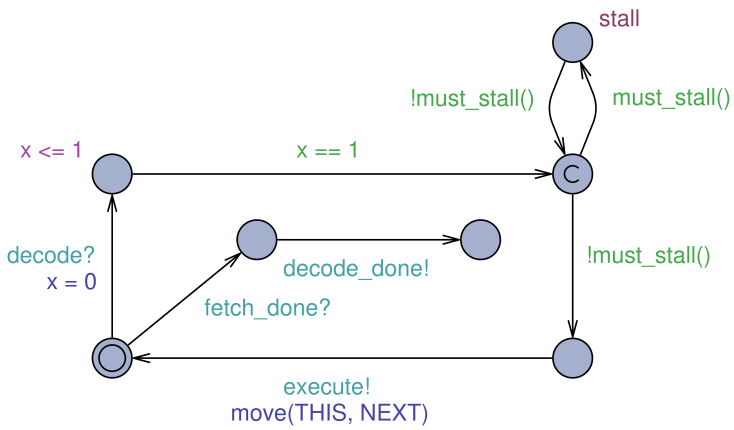

Figure 4. UPPAAL model for the decode stage in the ARM9TDMI pipeline, with x being a local clock.

Consider the instructions:

```
LDR R0, [R1]     /* Load R0 with the word pointed to by R1 */
ADD R2, R0, R1   /* Store the sum of R0 and R1 in R2 */
```

The sum cannot be computed before the value of `R0` is available, and the second instruction must therefore stay in the decode stage until the load has finished in the memory stage. The possibilities for pipeline stalls are documented by four examples in the reference manual for the core. The pipeline model in METAMOC handles all four examples cycle-accurately.

To further validate our pipeline model, we have used it to calculate the number of cycles for executing some small, single-path programs from the Mälardalen WCET benchmarks, and compared these cycle counts to results from the ARMulator emulator[7], assuming only cache hits. The cycle counts are comparable (with our estimates erring on the safe side), e.g.: `fibcall` gives 407 vs. 415. It should be noted that ARMulator does not give any definite guarantees regarding cycle-accuracy[8], which means the cycle counts can only be used for approximate comparisons.

An important property of the ARM920T processor is that it is free of "timing anomalies", as its pipeline is in-order [3]. If a processor has timing anomalies, it means that the local worst-case might not lead to the global worst-case. For instance, a cache hit rather than a cache miss might lead to a longer overall execution time. The absence of timing anomalies makes it convenient to find overapproximations, as the local worst-case can be used. Alternatively, if presented with a processor with timing anomalies, additional non-determinism in the model might be used to try all local possibilities.

3.2. Modelling Caches

Another feature for improving the average execution pace is caching. The basis of caching is the principles of locality. Caches improve the pace greatly, since main memory access might take 33 cycles while a cache access typically only requires a single cycle. A cache is divided into sets, where each block from main memory can reside in precisely one of these sets. Each set is divided into lines, also called "ways". A memory block can be stored in any of the lines, in the set it can be cached in. When memory access occurs, eviction of a line in a cache set might be required, and a replacement

[7] `http://infocenter.arm.com/help/topic/com.arm.doc.dui0058d/DUI0058.pdf`
[8] `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka4106.html`

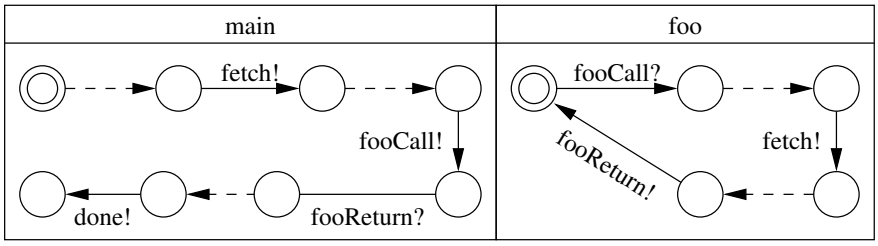

Figure 5. Sketches of the UPPAAL models for the functions `main` and `foo`.

policy is used to determine which line to evict.

The ARM920T processor has separate instruction and data caches. Both are 16 KB, 64 way associative, have eight words (i.e. 32 bytes) per line, support the write-through and write-back write policies, and support the pseudo-random and round-robin replacement policies. We only consider the round-robin policy, as it is the most predictable. The set for a byte at address x is determined by $(x \ \& \ ((ns-1) \ll log_2(ls))) \gg log_2(ls)$, where ns is the number of sets, ls is the line size in bytes, `&`, $\ll$ and $\gg$ are bitwise and and shift operators. This expression, slightly modified, is part of the cache models.

In order to add caching to the pipeline model, each cache is modelled as a UPPAAL model, simulating a cache hit by delaying for one cycle and a cache miss by synchronising with the UPPAAL model for main memory, which delays the appropriate number of cycles. The cache model has to keep track of which memory blocks are currently in the cache. It does so by storing an array of 512 addresses. Cache hits are determined using this array, and the cache replacement policy is implemented as functions.

3.3. Modelling Programs

The program is modelled as a data-insensitive CFG of the program, that communicates with the first stage of the pipeline. Figure 5 shows a simplified example of a program with two functions: `main` and `foo`. All programs have a `main` function, which is where the execution starts. Function calls are simulated by transferring control to the function automaton and transferring control back to the call-site when the function returns. This is illustrated in Figure 5 by synchronisation over the channels `fooCall` and `fooReturn`. Loops are handled using loop counter variables that ensure a loop back-edge can only be taken the specified number of times.

In order to reduce the amount of non-determinism in the program model it is determinised using a simple rule: executing more code increases the execution time. Concretely, this transforms loops to be taken the maximum number of times, and not allow forward branches to be taken if execution will eventually always flow to the destination.

The program CFG is annotated with the memory addresses accessed, determined statically using a value analysis. We have implemented a precise inter-procedural constant-propagation value analysis using weighted push-down systems (WPDSs) [7] and loop unrolling. For brevity reasons we will omit the details on the value analysis.

3.4. Support for ARM7 and ATMEL AVR 8-bit

Inspired by the WCET Tool Challenge 2008[9] we have implemented METAMOC for the ARM7TDMI processor core[10]. The core has the three pipeline stages fetch, decode and execute. The execute stage covers the actions performed by the execute, memory and writeback stages in the ARM9TDMI. Since the ARM9TDMI model could be reused extensively, and since both cores implement the v4T architecture, we were able to create the ARM7TDMI model in less than a man-week.

To show that other popular embedded processors can be supported as well, we have implemented support for the ATMEL AVR 8-bit instruction set. It took about one man-week to implement the support, and only required adding a new pipeline, creating support in Dissy for the AVR architecture and sligthly generalising the CFG generator.

4. Experiments

To evaluate the applicability and performance of our method, we evaluate it on a number of WCET benchmark programs from the Mälardalen Real-Time Research Centre. We compile the programs using a cross-compiling GNU C Compiler (GCC)[11]. The model generation is done on a 2 GHz Intel Core 2 Duo processor with 4 GB of RAM, and the model checking is done on a Dell PowerEdge 2950 with two 2.5 GHz Intel Quad Core Xeon processors and 32 GB of RAM.

We have manually annotated all loops in the programs with loop bounds. In addition we have promoted a few local variables to the global scope to sidestep GCC's translation of large local arrays into data segments with specialised initialiser code. We have discarded programs that either use floating point operations, do dynamic jumps (writes to the program counter), or do not compile. GCC inserts software floating point routines, which we could analyse given an estimation of the routines' loop bounds — these are hard to estimate though, without thorough manual analysis. This resulted in 21 programs[12] for the ARM architecture and 19 programs for the AVR architecture[13].

METAMOC has many parameters that can be adjusted for different trade-offs between precision, memory and analysis time: the compiler optimisation level, the amount of heuristic determinisation and manual annotation of the models, the level of hardware detail modelled and model checker options (specifically state space reduction techniques). To demonstrate the modularity of the method we have tested three different ARM9 configurations in order of increasing precision, while also increasing the analysis time: with no caches (always assuming that main memory is accessed), with only an instruction cache, and with both an instruction cache and a data cache. Our value analysis is only used when the data cache is enabled. The improvements gained by using more precise models can be seen in Figure 6a, while the increase in analysis time can be seen in Figure 6b. We have omitted the benchmarks for the ARM7 architecture as the results are very similar to the ARM9 results.

Our applicability results are presented in Table 1, together with the analysis times in Figure 6b. For

[9] `http://www.mrtc.mdh.se/projects/WCC08/`

[10] `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0210c/DDI0210B.pdf`

[11] For ARM: GCC 4.1.2, with the options `-O2 -g -fno-builtin -fomit-frame-pointer`. For AVR: GCC 4.3.3, with the options `-O2 -g -fno-builtin -fno-inline -fomit-frame-pointer -mmcu=avr5`

[12] adpcm, bs, bsort100, cnt, compress, crc, edn, expint, fac, fdct, fibcall, fir, insertsort, janne_complex, jfdctint, matmult, ndes, ns, nsichneu, prime, ud.

[13] The same as for the ARM, except bsort100 and nsichneu, which failed compiling due to being too large for the AVR.

ARM9, 21 benchmarks	
Analysable without caches	21
Analysable with instruction cache	20
Unanalysable, state space explosion	1
Analysable with data and instruction cache	20
Unanalysable, state space explosion	1
Manual modification of e.g. data cache size	4

ATMEL AVR 8-bit, 19 benchmarks	
Analysable	16
Unanalysable, state space explosion	3

Table 1. How many programs were analysable, and reasons for failure.

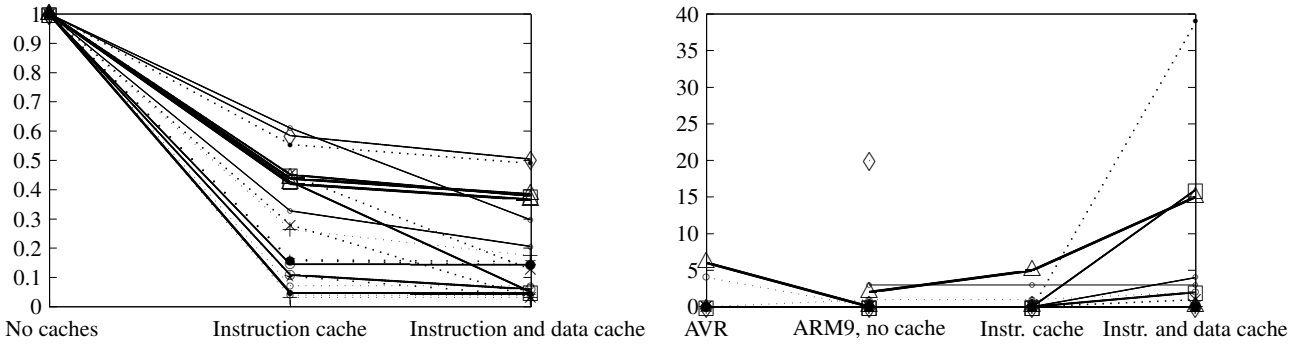

Figure 6. (a) Improvement in WCET estimate by precisely modelling the different caches on the ARM9. The average improvement in WCET estimate is 71.6% by modelling the instruction cache, and 81.3% by modelling both caches. (b) Analysis times for the different configurations in minutes. The average succesful analysis times are, respectively: 52.46 secs., 83.61 secs., 44.8 secs. and 265.75 secs.

the ARM9 we are able to provide WCETs for all 21 benchmarks. The adpcm program results in state space explosion when enabling any caches. When both caches are enabled, we manually have to modify the models for four of the benchmarks: compress has a small syntactical error due to deep loop nesting; and for bsort100, matmult and ndes the number of data cache lines modelled concretely must be reduced from 512 to 128, 64 and 32 (which amounts to editing a constant in the model editor, due to the modular design). Without this manual modification, UPPAAL runs into its 4 GB memory limit and quits.

More AVR benchmarks suffer from state space explosion than ARM benchmarks, primarily due to the ARM architecture having support for conditional execution of all instructions, thus reducing the number of distinct paths through the program.

The analysis times are all within 40 mins., with the average across all configurations and benchmarks being 111.65 secs. Details of the benchmarking are available at the METAMOC website, including the actual WCET estimates and UPPAAL models generated.

5. Related Work

Using model checking for determining worst-case execution times (WCETs) is a debated approach. In [10] it is claimed that model checking is not suitable for WCET analyses, however, in [6] it is shown that model checking can actually improve WCET estimates under the influence of caching on a simple processor. In this paper we show that model checking can be used for WCET analysis on a real-world, modern processor — and with good results and performance. A further advantage of our approach is the great modularity with which the model is constructed; it encourages reuse and easy

retargeting to new processors.

A WCET analysis is often separated in the following four analyses: value, cache, pipeline and path analysis. In [5] it is claimed that it is impossible to make the four analyses as separate, modular analyses and at the same time get sufficient precision. The method of this paper is separated in modular elements with clean interfaces.

Cache analyses can generally be sorted into abstract and concrete cache analyses. The common model for abstract cache analyses is presented in [4] and has the advantage of being space efficient, with a trade-off in the loss of precision.

The pipeline analysis typically uses an abstract model of the pipeline to take its impact on the execution into account [4]. The pipeline analysis should be able to handle unknown memory values. They might lead to non-determinism, as it might be impossible to deduce a reasonable overapproximation. For this reason, abstract pipeline states are traditionally represented as a set of concrete pipeline states [8]. Recent work has looked into using binary decision diagrams (BDDs) to represent abstract pipeline states [12]. The work presented in this paper is conceptually similar but the standard reduction techniques of the model checker is used.

For the path analysis, implicit path enumeration technique (IPET) and integer linear programming (ILP) have been combined in several tools [11, p. 42]. In [9], a path-based method is presented and has been implemented as an alternative to IPET in the SWEET tool. The method is more effective than previous path based methods. Furthermore, path based methods explore a path explicitly which, in contrast to IPET, could make debugging and infeasible path pruning easier. The path analysis presented in this paper is a simple exploration of the CFG of the program, with pruning of paths which cannot lead to the worst-case behaviour, but no pruning of infeasible paths.

6. Conclusion and Future Work

The optimisation features of modern processors, such as caching and pipelining, makes it difficult to determine safe and tight WCETs. Our method, METAMOC, is a very modular and easily retargetable approach for determining WCETs for programs running on hardware platforms featuring e.g. caching and pipelining. In order to evaluate the method, a prototype implementation has been made for the ARM9 architecture, a typical processor for embedded systems. To show the modularity of the approach the initial prototype has been extended with support for the ARM7 and ATMEL AVR architectures.

The prototype has been benchmarked to test its performance and general applicability. The experiments additionally show that much tighter WCET estimates are found when taking instruction caching is into account: up to 96% tighter estimates, and 71.6% on average. Also considering the data cache increases the average to 81.3%. When taking both caches into account, the average analysis time is just under five minutes. For the ARM9 and ARM7 architecture WCET estimates are given for all benchmarks, but requiring manual tweaking in four cases. For the ATMEL AVR three programs are unanalysable due to the model checker running out of memory.

Future work includes improving the model checker technology. We speculate that our models will parallelise very efficiently, as paths seem to be quite independent (especially when including caches). Distributing the model checking across more hosts will allow us to use much more memory, thereby

allowing the analysis of much larger programs. Exploiting the structure of our models, to summarise the effects of long deterministic chains into single steps should also help. Seeing that abstract caches seem to give a good trade-off between precision and performance, adding support for abstract caches would be interesting. Finally, instead of being data-insensitive, we would like to incorporate some form of flow facts into the program model. We already support this in some form, by allowing the user to manually annotate the program model, but it would be more beneficial if some flow facts were deduced automatically.

References

[1] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, first edition, 2008.

[2] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A Tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *SFM-RT*, number 3185 in LNCS, pages 200–236. Springer, 2004.

[3] Jakob Engblom and Bengt Jonsson. Processor Pipelines and Their Properties for Static WCET Analysis. In *Embedded Software*, volume 2491 of *LNCS*, pages 334–348. Springer, 2002.

[4] Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Applying Compiler Techniques to Cache Behavior Prediction. In *LCTRTS '97*, ACM SIGPLAN, pages 37–46, 1997.

[5] Reinhold Hechmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The Influence of Processor Architecture on the Design and the Results of WCET Tools. *Proceedings of the IEEE*, 91(7):1038–1054, 2003.

[6] Alexander Metzner. Why Model Checking Can Improve WCET Analysis. In Rajeev Alur and Doron Peled, editors, *CAV*, volume 3114 of *LNCS*, pages 334–347. Springer, 2004.

[7] Thomas Reps, Akash Lal, and Nick Kidd. Program Analysis using Weighted Pushdown Systems. In *FSTTCS 2007*, volume 4855 of *LNCS*, pages 23–51. Springer, 2007.

[8] Jörn Schneider and Christian Ferdinand. Pipeline Behavior Prediction for Superscalar Processors by Abstract Interpretation. In *LCTES '99*, pages 35–44, New York, NY, USA, 1999. ACM.

[9] Friedhelm Stappert, Andreas Ermedahl, and Jakob Engblom. Efficient Longest Executable Path Search for Programs with Complex Flows and Pipeline Effects. In *CASES '01*, pages 132–140, New York, NY, USA, 2001. ACM.

[10] Reinhard Wilhelm. Why AI + ILP is good for WCET, but MC is not, nor ILP alone. In Bernhard Steffen and Giorgio Levi, editors, *VMCAI*, volume 2937 of *LNCS*, pages 309–322. Springer, 2004.

[11] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The Worst-Case Execution Time Problem — Overview of Methods and Survey of Tools. *Transactions on Embedded Computing Systems*, 7(3):1–53, 2008.

[12] Stephan Wilhelm. Efficient Analysis of Pipeline Models for WCET Computation. In *Proceedings of the 5th International Workshop on Worst-Case Execution Time Analysis*, 2005.