Measurements or Static Analysis or Both?

Stefan M. Petters Patryk Zadarnowski Gernot Heiser*
National ICT Australia† and University of New South Wales
Sydney, Australia
firstname.lastname@nicta.com.au

Abstract

To date, measurement-based WCET analysis and static analysis have largely been seen as being at odds with each other. We argue that they should be considered *complementary*, and that the combination of both represents a promising approach. In this paper we discuss in some detail how we aim to improve on our probabilistic measurement-based technique by adding static cache analysis. Specifically we are planning to make use of recent advances in the functional languages research community. The objective of this paper is not to present finished or almost finished work. Instead we hope to trigger discussion and solicit feedback from the community in order to avoid pitfalls experienced by others and help focus our research.

1 Introduction

Embedded systems are becoming more pervasive by the day, and many of these embedded systems are subject to temporal requirements. While many of these systems are not life critical, missing deadlines may well be a costly excersise if experienced as degraded functionality or quality of service by millions of end users.

The analysis of worst-case execution times (WCET) is a fundamental building block of any form of real-time analysis. Most of the work to date has been based either on static analysis or on measurements. The research community has predominantly focussed on static analysis, but measurement-based techniques have gained increased significance in the last ten years.

The two approaches have largely been seen as mutually exclusive, and proponents of either approach tend to be quite critical of the other. Common concerns voiced about measurement-based analysis are that...

1. is unsafe, as there are no guarantees that the worst case has been observed and
2. measurements are too expensive if sufficient coverage is to be achieved.

On the other hand, critics of the static-analysis approaches claim that static analysis ...

1. is unsafe, as modern architectures are highly complex and thus modelling them is an error prone process, not last due to lack of documentation,
2. raises substantial challenges in terms of portability, and
3. does not support the more creative features used to improve performance in today's architectures.

We believe that ultimately a combination of the two paradigms is required to overcome the issues in both. Specifically, we propose to use measurements to obtain realistic, accurate results and static analysis to back the findings of the measurement phase by establishing that major contributors to the variability of the execution time have been adequately covered. Besides variations in program path, which are usually covered in the computation phase of WCET analysis approaches, caches contribute most substantially to variations in the execution times of software. Establishing whether all cache misses predicted by static analysis have been observed in the measurements is of substantial help in ensuring confidence in results obtained by measurements. Focussing on caches allows for easy verification whether the model used is actually correct and provides a high degree of portability of the analysis.

Furthermore the results of the static analysis

*Also with Open Kernel Labs

†National ICT Australia is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Research Centre of Excellence programs.

of caching behaviour can be used to reduce the overestimation produced by analysing the measurements of small untis independently and conservatively covering any possible dependency between the units.

2 Related Work

Cache analysis for WCET analysis was pioneered by Mueller [1] and Lim et al. [2]. The latter represented a holistic WCET/schedulability analysis that was subject to considerable complexity and was eventually abandoned as a line of research. Mueller's work has been refined over the years [3]. The main drawback of the approach is the loss of information inherent to the abstraction process, specifically the loss of information incurred when abstract cache states are merged at points where two paths of a program merge using the predefined *join* function. Further, abstract analysis, the tool of choice for static program analysers, has proven notoriously resilient to all non-trivial attempts of applying the analysis to programs that manipulate dynamic data structures such as linked lists or those in which pointers to functions cannot be resolved statically. While it can be argued that both features are rarely found in real-time programs, they are nevertheless common in certain critical parts of the system such as dynamic schedulers and page tables.

Ferdinand and Wilhelm [4] have extended Mueller's work by introducing the *must and may* analysis, effectively reducing the amount of information lost by the *join* function and prove that the resulting abstract domain is optimal. Nevertheless, even these improvements, the analysis still loses some information at junctions of control-flow paths introduced by any chosen program representation. Further, the *must and may* analysis suffers from the same limitations as earlier approaches when applied to code manipulating dynamic data structures.

Attacking the WCET problem from a different angle, Kirner et al. [5] deployed static analysis to identify a set of input data, which would enforce any possible path combination to be executed, effectively doing a full path enumeration. These sets are then fed into the program and measured on real hardware. In order to manage complexity, the program under test is divided into program segments which are tested and measured independently. The approach did not support caches and thus is not applicable to our work.

Yamamoto et al. [6] approached the problem of ensuring measurement coverage of cache states, by measuring each basic block in isolation in a best case scenario; in other words, all referenced memory locations are preloaded into the caches. A separate cache analysis provides a worst-case cache-miss scenario for the given basic block and enables the addition of the *cost* of these cache misses in the computation stage of the analysis process. The exact cache simulator used is not described in their paper, however, the analysed programs in their evaluation are sufficiently small to allow a brute force computation of the cache states.

3 Potoroo

A brief introduction into the overall framework is necessary to set the proposed approach into context. The Potoroo project aims to analyse the kernel primitives of the L4 microkernel API [7] for their WCET to enable real-time systems to be built on top. So far, we have developed a toolset which allows the measurement-based analysis of the kernel. In terms of the general approach it follows the paradigm used in [8].

The executable code is analysed to extract the control-flow graph (CFG). By using the executable code, all compiler optimisations and preprocessor modifications are considered. The analysis tries to be minimal by mainly focussing on control-flow changing instructions. However, that implies that in particular register-indirect branches are hard to resolve. Instead of a full analysis of the code, we have so far chosen to use a source code parser developed in the Goanna project at NICTA [9] and use debugging information to find corresponding part in the source code.

Traces may be generated either by software instrumentation, HW support, or with cycle-accurate simulators. Software instrumentation is subject to overhead and may quickly become too much of a burden in a running system. Cycle-accurate simulators, however, raise the question of accuracy of the model in the simulator—mostly right is not good enough. HW-supported tracing usually makes use of debugging ports implemented on the processor die, like the ETM macrocell in some ARM processors. While basic blocks exhibit their WCET easily compared to entire programs, there are no guarantees that a given block has been

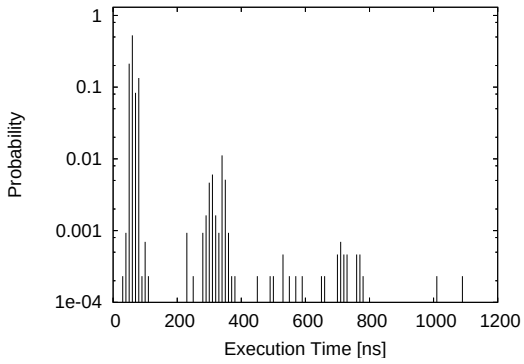

Figure 1: Sample ETP

completely represented in the execution time profile (ETP).

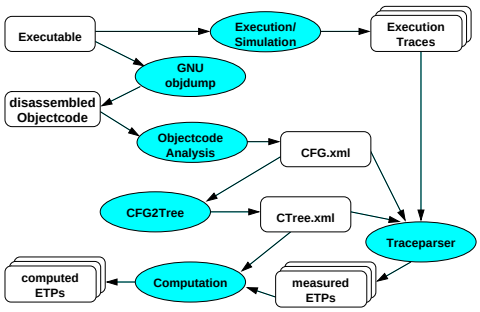

Figure 2: Toolset Overview

The traces are translated into ETPs, a sample of which is depicted in Figure 1, using the control-flow graph previously established. Besides actually producing the traces, this is the most computationally-expensive part of the approach.

The CFG is also translated into a tree. The tree directs the combination of ETPs to form ETPs describing larger code constructs. Using the tree ensures that any possible path combinations is considered.

For this paper the combination of sequential code constructs is of particular relevance. The toolset employs the supremal convolution [10]. The supremal convolution combines two distributions in such a way that any possible dependency between the two distributions is conservatively covered in the result, thus ensuring a safe combination of the two ETPs. However, a mayor drawback of supremal convolutions is that they are very conservative and tend towards a yes/no decision instead of a profile when many ETPs are combined [11].

4 Basic Idea

In the previous section we have identified two fundamental challenges to the approach we are taking in analysing the kernel.

1. Ensuring sufficient test coverage on basic block level.
2. Avoiding the overly-conservative nature of the supremal convolution without jeopardising safety.

Looking at the variability of the execution time in Figure 1 we can see that the ETP is clustered. These clusters can be attributed to cache misses, which are dominating the execution time of a given piece of code. Guaranteeing that the code has actually experienced its worst case of cache misses during the execution would go a long way to guaranteeing sufficient measurement coverage.

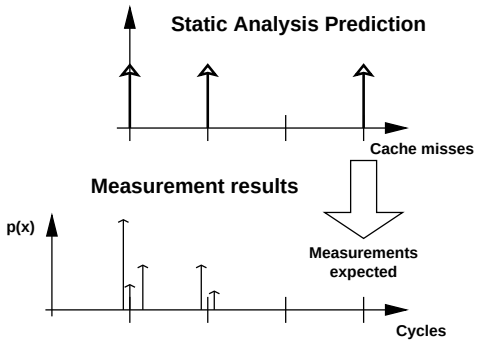

Figure 3: Coverage

In order to tackle this, we aim to establish for each ETP the different cache-miss scenarios expected and compare that to the measured ETP as depicted in Figure 3. While the creation of a complete and accurate model of a system including processor core, caches and peripherial devices is non-trivial and raises the issue of portability, caches themselves are only subject to a few parameters which can be easily established and verified for a given system [12]. In order to be able to make the connection between cache misses predicted and the measured ETP, it is necessary to reason about the cache-miss penalty actually imposed on a given cache miss.

While caches are used to mitigate the effect of long memory access latencies, modern processors try in various ways to mitigate the effect of cache-miss penalties. Critical-word-first loads by caches avoids the overhead of loading data which is not immediatly required, if the request does not hit

the first word in a cache line. Out-of-order execution enables the program to progress on instructions which are not dependent on the memory location being loaded. A side effect of out-of-order execution is that instructions independent of the cache miss are executed. Thus a cache miss at a given point in the program reduces the entropy of states the CPU may be in during the execution of subsequent instructions after the data has been fetched from memory.

Load/store architectures tend to tag registers waiting for outstanding memory requests, to enable continued execution until the register is actually used. This enables a smart compiler to make use of instruction scheduling to preload registers as early as possible to avoid as much of the maximum cache miss penalty as possible. Contrary to out-of-order execution, the pipeline is usually drained of instructions preceding the cache-miss causing instruction. Some architectures allow for only a single outstanding memory transaction as, for example, the ARM9EJ-S processor core. However, other processors allow for several outstanding requests, by implementing fill buffers and pend buffers like, for example, the XScale processor family.

Applying the above discussion to the environment we are performing our analysis in, we make the following observations:

1. The ARM9EJ-S is only subject to a single outstanding memory transaction, forcing a stall on subsequent loads.
2. The ARM-gcc compiler apparently makes little or no use of the reduced penalty of a delayed load, by usually using loaded registers within three instructions.

Any approach performing coverage analysis should inherently have information about dependencies between the cache misses of subsequent basic blocks and possibly beyond that. Exploiting these dependencies as depicted in Figure 4 allows, on the one hand, more realistic bounding of ETPs, and on the other hand, reduce the overall WCET.

5 Static Analysis Approach

Static analysis is well-established as a powerful tool for computing the WCET of a program. In particular, abstract interpretation, the tool of choice for static program analysers, is an attractive technique for WCET analysis, as it provides a method for a formally-provable derivation of concrete program properties such as cache misses or even actual bounds on the execution time. Unfortunately, in the past, all applications of static analysis in the area have been hampered by the limitations, described in section 2, inherent to the abstract interpretation technique.

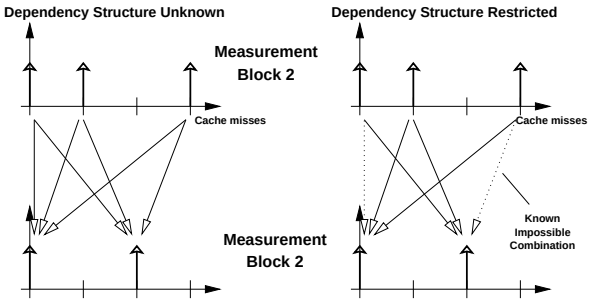

Figure 4: Dependency Analysis

5.1 Motivation

In our work, we observe that the problem of deriving the WCET of a given program using static analysis can be viewed as a search for a proof of the desired program property. In particular, abstract interpretation can be viewed as a way of deriving a constructive proof of the desired property by computing that property directly from the structure of the program. However, if we knew the property in the first place (for example, through empirical measurement of program's behaviour) we could, in principle, construct an indirect proof of the same result. In particular, we can attempt to prove the result by showing that no possible execution scenario can result in an answer different from the assumed one. Conversely, we can disprove our hypothesis by searching for a suitable counter-example during program analysis. In the remainder of this section, we argue that the indirect approach is particularly well suited to the problem of computing the number of cache misses experienced during execution of a program.

5.2 The Basic Approach

We take the set of cache miss counts observed during measurement of the program as a hypothesis, which we subsequently attempt to prove or disprove through static analysis of the program. The problem is simpler than attempting to compute the cache behaviour "from scratch" since the measured answer provides finite bounds on the amount of

computation performed during analysis, independent of the bounds imposed by the particular abstract domain and the associated join function. This gives us more leeway in the design of the abstract domain, and in fact permits us to perform the static analysis of the program with virtually no loss of information at all. In particular, observe that:

1. Since the measured set of cache miss counts is finite, it can always be obtained after a finite number of steps during abstract analysis, provided that we take some simple precautions in the design of our algorithm to avoid divergent chains of computation.
2. If due care is taken during design of the analysis algorithm, the above observation is sufficient to guarantee that the analysis is performed in a "reasonable" amount of time. However, this does not prevent us from taking additional measures to avoid the exponential complexity of complete control path enumeration by combining analysis for sections of the program that are common to two or more potential execution paths. In our approach, we will avoid exponential complexity by binding execution time of our analysis to the number of cache miss counts observed during measurement.

In other words, we can safely "run" the analyser until it has either constrained the set of cache miss counts to a subset of the measured one, or else until it has detected a counter-example to our hypothesis. In the later case, the state of the analyser at the time when the counter-example has been detected provides invaluable clues permitting the user to extend the measurement suite to cover the omitted execution scenarios.

Note that this approach is strictly limited to analysing those programs for which a "perfect" measurement suite can actually be constructed from a finite number of test cases. This excludes, among others, non-terminating programs. Fortunately, this is precisely the class of programs suitable for use in real-time applications and accordingly, covers all programs that we are concerned with.

The remaining subsection outline our implementation of this technique.

5.3 Source Program Preparation

First, we translate the input binary program into a purely-functional representation using the technique pioneered by Chakravarty, *et al.* [13]. We choose a normal form of the continuation-passing style of lambda calculus as our program representation for its similarity to the low-level treatment of control flow on typical processor architectures. In the purely-functional form, all basic blocks are translated into functions with loops represented by recursion. Further, all global variables are replaced by additional function arguments "threaded" throughout the control-flow path of the program. This step is necessary for pragmatic reasons, since the subsequent program transformations would become prohibitively-expensive without the detailed data-flow information explicit in purely-functional programs.

Note that, in this paper, we use the term "function" in the declarative programming sense of the word, rather than to refer to the procedures of the input program. Every function in our analysis corresponds loosely to a basic block of the input program.

5.4 Cache Analysis

Next, we transform the input program into a new *analyser program* that dynamically computes the cache miss counts of the original program. This step is very similar to a conventional abstract analysis, and uses the same form of an abstract domain to represent the cache miss counts. However, since the solution we seek is computed dynamically during execution of the analyser program rather than statically in the course of analysis, we never have to join abstract values in the analyser program as described in section 2. During any given actual execution of the analyser, only one of all possible control flow paths can be followed. In other words, the analyser program provides a compact, finite representation of the large (and potentially infinite) number of all possible control flow paths that would have to be followed to obtain precise cache miss counts for every possible execution scenario, just like the original program provided a compact finite representation of all control flow paths of the original program. Note that the resulting program encodes the *precise* cache miss count for every possible control flow path without any loss of information. Also note that such translation of an input program into an analyser program is performed implicitly by every abstract analysis algorithm, although the resulting program is rarely "materialised" into an actual data structure, and typically remains encoded implicitly in the state of the static analyser.

Further, during conventional abstract analysis, the control structure of the translated program is simplified at the expense of precision to ensure termination of the analyser.

Further, we perform a number of standard optimising transformations of the generated programs, including constant folding, copy propagation and dead code elimination. Since the typical calculations involved in computing cache miss counts are relatively straight-forward in comparison to the work done by the original input program, we expect these simple transformations to result in a dramatic reduction to the size of the analyser program. In particular, large chunks of code that do not affect cache behaviour should disappear from the program, thus substantially reducing the cost of the subsequent stages of the whole process. We also perform an induction variable analysis to reduce many common loop patterns such as those used to obtain a sum of an arithmetic series into a simple scalar expression.

5.5 Coverage Analysis

Finally, we analyse the transformed program to verify that it can only return values from the set of cache miss counts observed during measurement. In other words, we seek to find the maximal set of input arguments for which the analyser program returns an answer within the range specified by the measured set. Our solution first constructs an inverse of each function f in the program (in other words, a function f^{-1} such that $f^{-1}(x) = \mathbf{Y}$ iff, for all y in the set $\mathbf{Y}$, $f(y) = x$.) It is a remarkable fact that such inversion can be performed relatively easily for all functions that may be encountered in an analyser program. This is because we are seeking total inversion only, rather than partial inversion (where some of the input arguments to the original functions remain fixed) which is a harder problem. While the ranges of the inverted functions grow quickly with the number of original function parameters, as will be shown shortly, this is not a problem in our application because the size of those ranges is used to bind the depth of our analysis and facilitate early termination of our algorithm; the true exponential growth is therefore never reached and the overall complexity of the algorithm is a logistic function of the size of the measured set and the number of basic blocks in the input program. The term *logistic function* relates to an intial exponential growth of the function which subsequently slows and finally stops.

The analysis maintains a work list of inverted functions annotated with a set of their input arguments. Initially, the work list contains only those functions that correspond to the leaves of the call graph of the original program, each annotated with the set of cache miss counts obtained through measurement. The algorithm proceeds by extracting each item from the work list in turn, and terminates when the work list becomes empty.

A single work list entry is analysed by applying the given input value set to the corresponding inverted function, thus obtaining the maximal set of corresponding program inputs. We recognise two scenarios:

1. If the resulting set of values is unconstrained (as will often happen by the nature of function inversion), we have determined that the measurements have been exhaustive along the corresponding control-flow path in the original program. Accordingly, the function is removed from the work list.
2. Otherwise, we determine the set of callers of the function under consideration in the original (non-inverted) program, and add the corresponding inverted functions to the work list. If no such functions exist, we have just examined the entry block of the original program, and accordingly report the resulting set of constraints to the user.

All constraints reported to the user as a result of the second point above represent constraints on the input of the original program that must be satisfied in order for the program's cache behaviour to remain within the measured set of cache miss counts. Accordingly, all input values outside of the constraint set represent counter-examples to our hypothesis.

5.6 Algorithmic Complexity

In the worst case, the algorithm may analyse all individual control-flow paths through the program. However, in practice the worst case is incredibly difficult to achieve, as the execution of our algorithm is bounded by the size of the constraint set, which itself grows exponentially during the function inversion process. This means that the actual complexity of the program is a logistic, rather than an exponential function. In fact, we believe that the amortised complexity of our algorithm for all terminating input programs is polynomial (quadratic) in the number of functions (basic blocks) in the pro-

gram. More research is needed to substantiate this result.

6 Conclusions

In this paper we have outlined our approach to supporting probabilistic measurement-based WCET analysis with static analysis. The static analysis is based on a functional representation of the code investigated and an abstract interpretation of representation. The goal is to establish sufficient measurement coverage, and to reduce overestimation of conservative combination ETPs by conservatively covering any possible dependencies between them. Future work will largely center finishing the implementation of the plan presented and subsequent evaluation.

References

[1] F. Mueller, *Static Cache Simulation and its Applications*. PhD thesis, Department of Computer Science, Florida State University, Tallahassee, FL, USA, July 1994.

[2] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, S.-M. Moon, and C.-S. Kim, "An accurate worst-case timing analysis for risc processors," *IEEE Transactions on Software Engineering*, vol. 21, no. 7, pp. 593–604, 1995.

[3] K. Patil, K. Seth, and F. Mueller, "Compositional static instruction cache simulation," in *Proceedings of the Conference on Language, Compiler and Tool Support for Embedded Systems2004*, (Washington, DC, USA), June11–13 2004.

[4] C. Ferdinand and R. Wilhelm, "Efficient and precise cache behavior prediction for real-time systems," *Journal of Real–Time Systems*, vol. 17, pp. 131–181, 1999.

[5] R. Kirner, I. Wenzel, B. Rieder, and P. Puschner, "Using measurements as a complement to static worst-case execution time analysis," in *Intelligent Systems at the Service of Mankind*, vol. 2, UBooks Verlag, Dec. 2005.

[6] K. Yamamoto, Y. Ishikawa, and T. Matsui, "Portable execution time analysis method," in *Proceedings of the 12th International Conference on Embedded and Real-Time Computing and Applications*, (Sydney, Australia), Aug. 2006.

[7] C. van Schaik and G. Heiser, "High-performance microkernels and virtualisation on ARM and segmented architectures," in *Proceedings of the 1st International Workshop on Microkernels for Embedded Systems*, (Sydney, Australia), NICTA, Jan. 2007.

[8] G. Bernat, A. Colin, and S. M. Petters, "pWCET: a tool for probabilistic worst case execution time analysis of real–time systems," technical report YCS353 (2003), University of York, Department of Computer Science, York, YO10 5DD, United Kingdom, Apr. 2003.

[9] A. Fehnker, R. Huuck, P. Jayet, M. Lussenburg, and F. Rauch, "Goanna — A Static Model Checker," in *Proceedings of the 11th International Workshop on Formal Methods for Industrial Critical Systems*, (Bonn, Germany), Aug. 2006.

[10] R. C. Williamson, *Probabilistic Arithmetic*. PhD thesis, Department of Electrical Engineering, University of Queensland, Brisbane, Australia, Aug. 1989.

[11] S. M. Petters, "Execution-time profiles," tech. rep., National ICT Australia, National ICT Australia, Sydney 2052, Australia, Jan. 2007.

[12] T. John and R. Baumgartl, "Exact cache characterization by experimental parameter extraction," in *Proceedings of the 15th International Conference on Real-Time and Network Systems RTNS07*, (Nancy, France), pp. 65–74, Mar. 2007.

[13] M. M. Chakravarty, G. Keller, and P. Zadarnowski, "A functional perspective on ssa optimisation algorithms," in *Electronic Notes in Theoretical Computer Science* (J. Knoop and W. Zimmermann, eds.), vol. 82, Elsevier, 2004.