Best Practice for Caching of Single-Path Code*

Martin Schoeberl[1], Bekim Cilku[2], Daniel Prokesch[2], and Peter Puschner[2]

1 **Department of Applied Mathematics and Computer Science**
Technical University of Denmark
masca@imm.dtu.dk

2 **Institute of Computer Engineering**
Vienna University of Technology, Austria
{bekim,daniel,peter}@vmars.tuwien.ac.at

Abstract

Single-path code has some unique properties that make it interesting to explore different caching and prefetching alternatives for the stream of instructions. In this paper, we explore different cache organizations and how they perform with single-path code.

1998 ACM Subject Classification C.3 Real-Time and Embedded Systems

Keywords and phrases single-path code, method cache, prefetching

Digital Object Identifier 10.4230/OASIcs.WCET.2017.2

1 Introduction

Worst-case execution time (WCET) analysis is a non-trivial analysis problem. It becomes especially difficult with more complex processor architectures. A strategy to simplify WCET analysis is to write programs that have a constant execution time, i.e., the best-case and worst-case execution time are equal. In that case, we do not need to analyze the program, but can simply measure the execution time. Single-path code gives constant execution time [15].

Single-path code is code that is structured so that there are no data dependent control flows. On an `if/else` condition both conditions are executed. However, to retain the program's semantics and data flow, all instructions are executed with a predicate. The compiler sets these predicates according to the original conditions of the branching code. When executing single-path code, instructions whose predicate evaluates to `false` do not update the processor state, i.e., they act as `nop` instructions. Loops always execute the maximum number of iterations (the so-called loop bound), which is a known number in a real-time context. Like the `if/else` case, the original loop condition is used to evaluate to a predicate and all instructions within loops are predicated.

Single-path code can be manually coded or a compiler can translate *normal* code to single-path code. The translation of an `if/else` condition is also a common technique in compilers applied on small code fragments to avoid expensive branches. This is called if conversion [1].

* The work presented in this paper was partially funded by the Danish Council for Independent Research | Technology and Production Sciences under the project PREDICT (`http://predict.compute.dtu.dk/`), contract no. 4184-00127A. This paper was partially funded by the EU COST Action IC1202: Timing Analysis on Code Level (TACLe) and the European Union's 7th Framework Programme under grant agreement no. 288008: Time-predictable Multi-Core Architecture for Embedded Systems (T-CREST).

The time-predictable execution of single-path code demands two features from a processor: (1) the processor needs to support predicates or a conditional move and (2) a predicated instruction shall have the same execution time irrespective of whether the predicate evaluates to `true` or `false`. Therefore, we explore single-path code on a processor that fulfills both conditions. Patmos is a processor that is designed especially for real-time systems [21]. Patmos contains features that shall make WCET analysis simpler, but also supports execution of single-path code with a predicated instruction set and constant execution time of instructions.

Patmos contains also a special instruction cache that caches full functions [4]. For historical reasons this cache is named method cache (it appeared first in a Java processor [18]). Cache misses can only occur at function calls or returns. Caching full functions has one drawback: code that is not executed is still loaded into the cache. However, as programs organized as single-path code execute all their instructions, this main drawback disappears. Therefore, our hypothesis is that the method cache is a good cache organization for single-path code.

This paper explores the method cache in the context of single-path code. We compare and evaluate the method cache against a standard instruction cache using the TACLeBench benchmarks [6]. Furthermore, we explore performance benefits of an extension of a standard instruction cache with a prefetcher that has been especially designed for single-path code.

The paper is organized in 6 sections: The following section presents related work. Section 3 provides background on single-path code generation and the time-predictable Patmos processor. Section 4 describes different options of caching for single-path code. Section 5 evaluates the different caching options on the Patmos processor and compares them. Section 6 concludes the paper.

2 Related Work

For real-time systems, caches are one of the main sources of temporal uncertainty. State-of-the-art cache analysis tools are using abstract interpretation for classifying cache accesses [12]. However, even if these approaches derive safe bounds, the precision of the results derived from the abstract models strongly vary depending on the cache architecture and replacement policy [9]. For example, an abstract model for the LRU replacement policy achieves better predictability than a model for FIFO or PLRU [17].

Another mechanism that aims at making caches more predictable is cache locking [14]. This technique loads memory contents into the cache and locks it to ensure that it will remain unchanged afterwards. The benefit of cache locking is that all accesses to the locked cache lines will always result into cache hits. The cache content can be locked entirely [7] or partially, it can be locked for the whole system lifetime (static locking) or it can be changed at runtime (dynamic locking) [5]. Although cache locking increases predictability, it reduces performance by restricting the temporal locality of the cache to a set of locked cache lines.

In contrast to conventional code, single-path conversion overcomes predictability issues by generating code that has only a single trace of execution. Thus, keeping traces of possible cache states is no more needed. Furthermore, the use of single-path code eliminates the necessity for cache locking.

3 Background

This paper builds on prior research work on single-path code and research on the time-predictable computer architecture developed for the T-CREST platform [19].

3.1 Single-path Code Generation

Puschner and Burns propose single-path code to simplify WCET analysis by avoiding data-dependent control flow decisions [15]. The defining property of single-path code is that any execution follows a single instruction trace, independent from input data. This is achieved by conversion of control dependence to data dependence, with the use of predicated instructions. In code that is WCET analyzable, loops must be bounded. The compiler transforms input-data dependent loops such that they iterate for a fixed number of times, which is the local loop bound [13].

The resulting single-path code may have a longer execution path, due to the serialization of otherwise alternative paths in the original program. Also, in some scenarios it is undesirable to always consume the computational resources required for the worst-case, e.g., in mixed-critical systems where slack time is used for non-critical tasks. Nonetheless, single-path code generation provides a constructive approach to time-predictable real-time code. On a "well-behaved" hardware platform, the execution time for single-path tasks is constant. In this ideal case, WCET analysis simplifies to measurement.

One requirement is that the instruction timing is independent of the instruction operands. Memory accesses introduce another source of variability in execution time. Though, the single-path property makes the code easier to analyze with regards to instruction memory. Abstract interpretation based analysis becomes superfluous, there is no need for approximation. The known singleton instruction stream can be directly applied to a hardware model of the instruction cache (as in simulation). This knowledge is exploited to implement perfectly accurate prefetching schemes for instructions [2].

Data accesses are also subject to execution-time variability. Enforcing local availability of the required data during the task execution may alleviate the problem, e.g., by data cache locking or usage of a scratchpad memory. However, we restrict ourselves to the instruction cache in this paper.

3.2 Patmos and the T-CREST Platform

We explore instruction caching options on the Patmos processor [21], which itself is part of the T-CREST multicore platform [19]. The T-CREST platform aims to build a processor, network-on-chip, and compiler toolchain [16] to simplify WCET analysis. We optimized all components to be time-predictable, even when average-case performance is reduced. AbsInt aiT [8] static WCET analyzer supports the Patmos processor. T-CREST also includes the research WCET analyzer platin [11].

Patmos is a RISC architecture supporting dual-issue instructions. To the best of our knowledge, Patmos is timing anomaly free. There is no timing dependency between any two instructions. Even all cache misses (instruction or data) happen in the same pipeline stage (the memory stage). Therefore, only a single cache miss can happen any clock cycle. Patmos uses special forms of instruction and data cache that shall simplify cache analysis. For instructions, Patmos has a method cache [4], which caches whole functions. Besides these special caches Patmos also supports a standard instruction cache, a standard data cache, and instruction and data scratchpad memories.

One issue with a method cache is that complete functions are loaded into the method cache, even when only part of it is executed. We attack this issue by splitting larger functions into smaller subfunctions [10]. However, with single-path code there is no code that is not executed. The processor executes all instructions of a called function. Therefore, a method cache may well fit for caching single-path code, especially when callee functions are not

evicted by called functions.

We extended a standard instruction cache by a prefetching unit [3] to improve single-path execution time. The prefetcher uses the static "knowledge" of single-path code to anticipate the upcoming instructions and bring them into the cache before they are needed. Such a property allows the prefetcher to perform in time-predictable fashion without polluting the cache with unused instructions.

4 Caching of Single-Path Code

Single-path code is instruction-cache friendly as all instructions that are loaded into the cache are executed, except at the end of a function.

4.1 Standard Instruction Cache

A standard instruction cache is organized in cache blocks and can be configured as direct mapped cache or set associative cache. Some single-path code can benefit from a direct mapped instruction cache, especially when the cache is small [3]. We will evaluate direct mapped and set-associative instruction caches for single-path code.

4.2 Method Cache

The method cache is an instruction cache designed to simplify WCET analysis. The method cache caches full functions/methods. Therefore, a cache miss can only happen on a call or a return. All other instructions are guaranteed hits and cache analysis can ignore those. Method cache analysis only needs to consider functions and not individual instructions.

One disadvantage of the method cache is that instructions in a function that are not executed are still loaded on a cache miss. However, with single-path code all instructions of a function are always executed. Therefore, the method cache should perform well with single-path code.

4.3 Time-predictable Prefetcher with a Standard Cache

Prefetching hides large memory access latencies by loading instructions into the cache before they are needed. However, to take advantage of this improvement, the prefetcher needs to guess the right prefetch target and issue the request at the right time. Any wrong speculation on these two parameters can degrade the system performance.

The time-predictable prefetcher exploits properties of single-path code to anticipate the future instruction cache accesses with full accuracy and bring those instructions into the cache at the right moment [3]. For higher efficiency, the prefetcher implements an aggressive algorithm that prefetches every cache line of the code. Its direction is controlled through a *Reference Prediction Table* (RPT), which is an optimized projection of single-path code that captures the control-flow behavior of the code.

The entries of the RPT control the behavior of the prefetcher. They contain addresses at which the prefetcher should switch between sequential and non-sequential prefetching. Figure 1 shows the generation of the RPT. It begins with obtaining the execution trace of the single-path code. We use the Patmos simulator to export the program counter values during a program run. We extract the start addresses of the functions from the symbol table of the executable. The trace analyzer uses the trace and the start addresses to produce a dynamic control-flow graph of the single-path function, where nodes are addresses of single

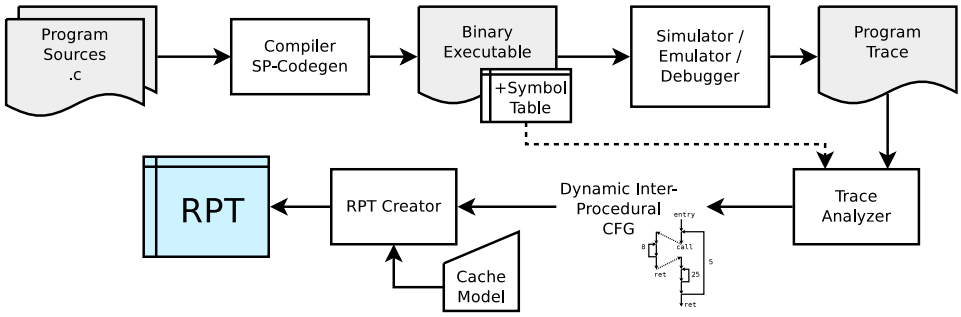

Figure 1 Generation of the Reference Prediction Table (RPT).

instructions. The trace analyzer identifies call sites, loops, loop nests, and loop iteration counts. The RPT creator then creates entries containing an address that shall trigger a change in the behavior of the prefetcher, a destination where to continue prefetching, and additional information depending on the entry type.

Generation of the RPT is based on a form of dynamic program analysis, without instrumentation of the original program. However, the extracted instruction trace is invariable regardless of the program inputs because of the single-path code. Also, in contrast to trace-based and profile-guided optimization, the analysis results are not directed back to the compiler. Instead, the execution is optimized by means of hardware in form of the specialized prefetcher.

5 Evaluation

We evaluate the program performance of single-path code with different caching methods. For the comparison, we use the Patmos processor. We configure Patmos for the Altera DE2-115 FPGA board, which means that the main memory is a 16-bit SRAM. This memory results in 21 clock cycles for a burst of 4 32-bit words to fill or spill a 16-byte cache line. All standard caches have the line size of the burst length, 16 bytes. We configure the instruction or method cache to be 8 KB large and the method cache to cache up to 16 functions. The data cache is 4 KB and the stack cache 2 KB. We use hardware simulation to get cycle accurate measurements.

For the evaluation, we use the TACLeBench benchmark collection [6] in version 1.9. We have added an attribute to the benchmarks' main function to avoid that it is inlined by the compiler. Otherwise we did not touch the source of TACLeBench. This main function is also the root function for the single-path code generation. We measure the execution time of the whole program, including initialization and result comparison code, in clock cycles.

We used a subset of the benchmarks. The variation of the execution time of the benchmarks is high, i.e., between hundreds and a billion clock cycles. For practical reasons, we did not use the long running benchmarks, as cycle accurate hardware simulation is time consuming.[1] However, a large execution time does not necessarily mean that those benchmarks would have a larger memory foot print. Furthermore, we dropped benchmarks where we cannot generate single-path code, e.g., recursive benchmarks. Also, we removed two outliers (`ludcmp`

[1] The simulation of the remaining benchmarks, just for a single cache size configuration, still takes 6–8 hours on a contemporary notebook.

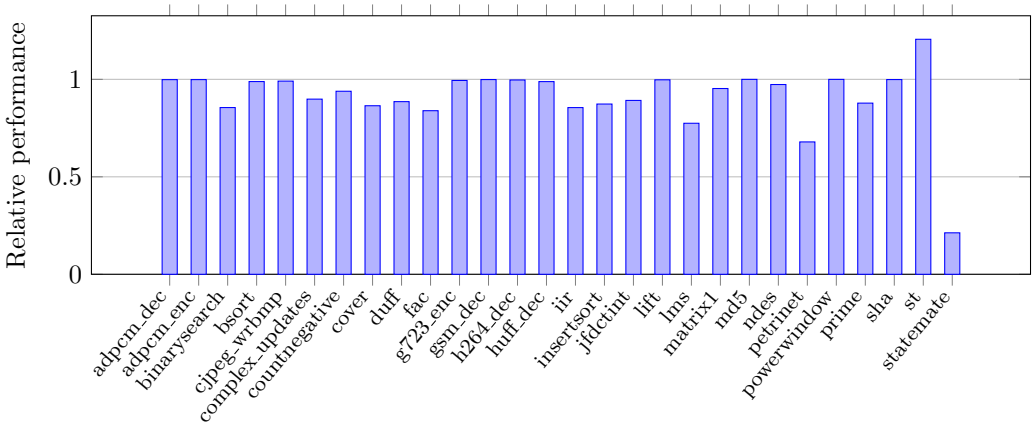

Figure 2 Relative average-case performance comparing the method cache with a standard cache on normal programs

and `minver`) as their results showed improvements of factors 3 to 4 for the method cache compared to a standard instruction cache.

All the following figures show measured performance improvements, where a bigger number means a better result. The figures show relative performance as one execution time scaled by the other execution time. That means a number larger than 1 is an improvement, and a number less than 1 is a regression. E.g., when comparing the method cache with the standard cache, the figure shows how much better (or worse) the configuration of the method cache is compared with the standard cache. For this example, the relation is the execution time of the benchmark with the standard cache divided by the execution time of the benchmark with the method cache (as a shorter execution time is better).

5.1 Baseline

As a baseline, we show the performance difference between using a method cache and a direct mapped instruction cache on normal compiled code. Figure 2 shows the execution time relation between those two configurations (normalized to the execution time with the standard cache). The geometric mean of the comparison is 0.88, which means that on average the configuration with the direct mapped instruction cache performs better. Those measurements are average case measurements and cannot be an indication of WCET analysis bounds. In these average case measurements, we see that some benchmarks perform equally for the two cache configurations. We assume those cases are when the benchmark fits entirely into the cache. Several benchmarks perform better with a normal instruction cache than with the method cache. However, this is an average case measurement and the method cache was designed to simplify WCET analysis.

5.2 Single-Path Comparison and Prefetching

Figure 3 shows the performance comparison between a method cache and a standard cache with single-path generated code. The figure is now more diverse than the average-case figure. Some benchmarks gain and some lose when using a method cache. There is no clear winner. The geometric mean of the comparison is 0.96, which means that on average the standard cache configuration performs slightly better.

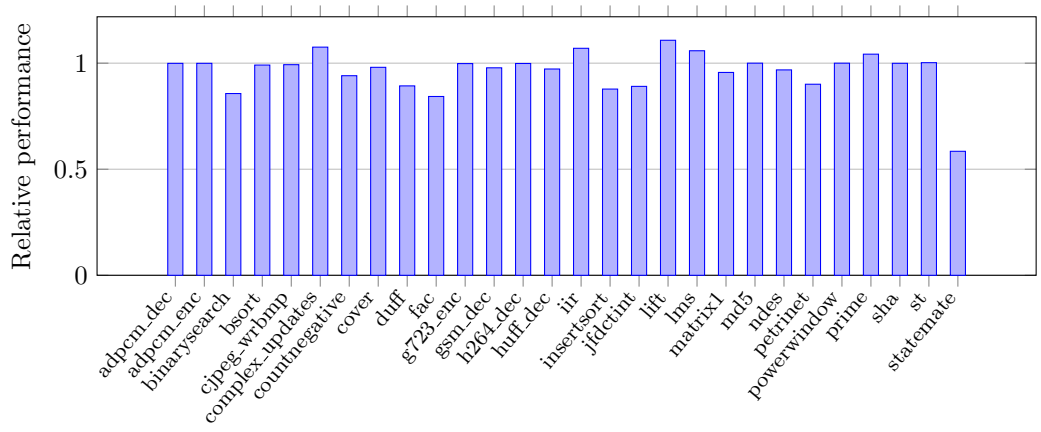

Figure 3 Relative single-path performance comparing the method cache with a standard cache

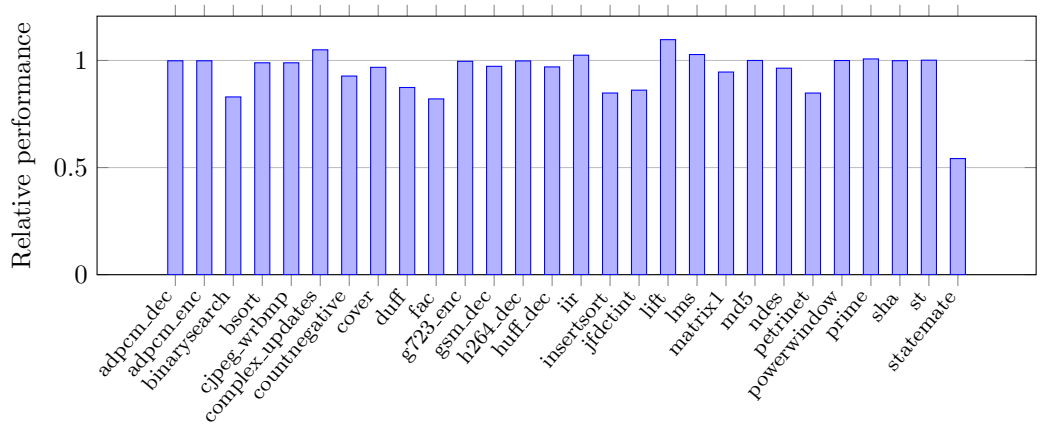

Figure 4 Relative single-path performance comparing the method cache with a prefetching cache

Figure 4 show the performance comparison between a method cache and an instruction cache that includes the prefetching unit. The results are similar to the results in Figure 3. Some benchmarks gain a little bit with the prefetching unit. The geometric mean of the comparison is 0.94. We assume that most benchmarks are almost fitting into the cache and leaving not enough room for improvement by prefetching. It has been shown that smaller caches benefit most from the prefetcher [3].

5.3 Associativity

Figure 5 shows the comparison of a 2-way cache with LRU replacement with a direct mapped instruction cache. Originally we assumed that a direct mapped cache is a better fit for single-path code as is avoids cache thrashing on loops that are larger than the cache. However, we see in the figure that some benchmarks benefit from a higher associativity. Only `statemate` performs better with a direct mapped cache. Therefore, we deduct that the 4 KB of one way is large enough for the larger loops in the benchmarks. The geometric mean of the comparison is 1.014, which means that on average the configuration with the 2-way instruction cache performs slightly better than the configuration with the direct mapped cache.

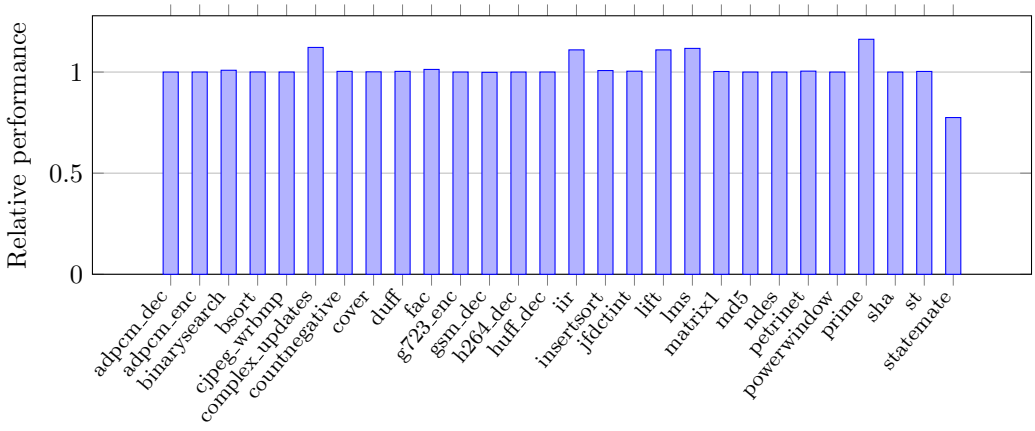

Figure 5 Relative single-path performance comparing a 2-way cache with a direct mapped cache

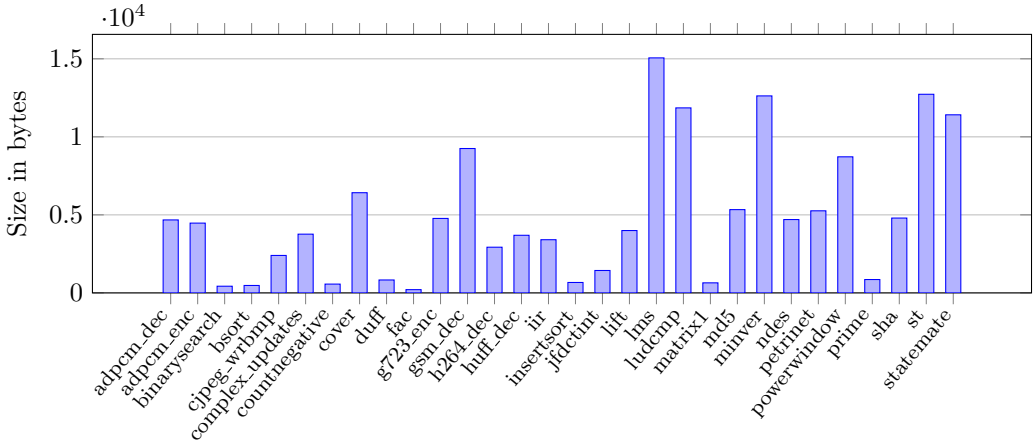

Figure 6 Benchmark dynamic sizes (instruction memory footprint)

5.4 Benchmark Sizes

To better understand the benchmark results, we measure the instruction memory footprint of the benchmarks. We collect actual runtime data of which code is executed. We trace the execution of a benchmark with the Patmos simulator and collect which functions are executed. We extract the function sizes from the executable. As single-path code executes the whole function, we simply add all sizes of the executed functions to retrieve the instruction memory footprint. We exclude startup and exit code.

Figure 6 shows the memory footprint of the benchmarks. We can see three typical sizes of benchmarks: (1) very small benchmarks, such as `binarysearch` or `duff` where the memory footprint is less than 1 KB, (2) medium sized benchmarks with a memory footprint of 4–5 KB, and (3) larger benchmarks with a memory footprint of around 10 KB. The largest benchmark is `lms` with 14.7 KB. These numbers do not include any startup or exit code executed, which by itself is 7048 bytes. That means for the small benchmarks the startup and exit code dominates the memory footprint.

With our standard configuration of 8 KB instruction cache, 23 out of the 30 benchmarks will fit into the cache (excluding startup and exit code). This means that many of the

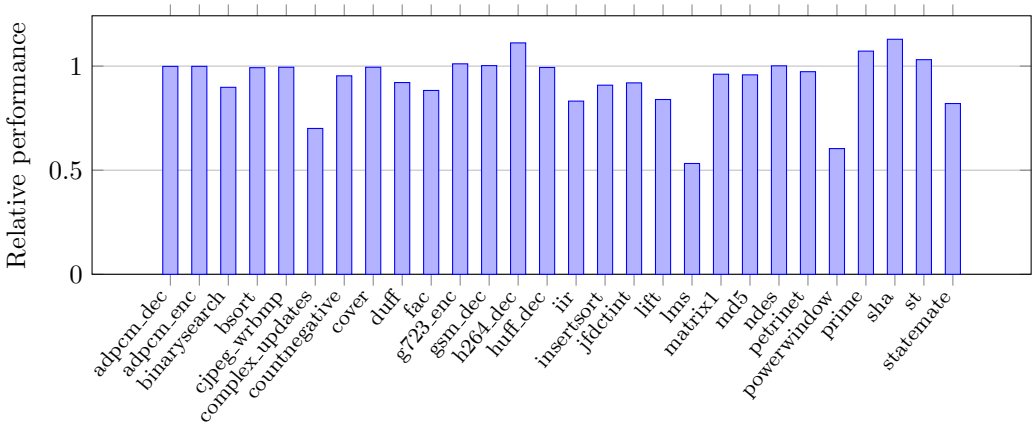

Figure 7 Relative single-path performance comparing the method cache with a standard cache (2 KB size)

TACLeBench benchmarks are too small for the evaluation of different instruction caches. Therefore, we explore artificial small caches in the following subsection.

5.5 Small Caches

For further experiments we reduce the size of the instruction and method cache to just 2 KB, a very small size for current embedded processors. Figure 7 compares the method cache with the direct mapped instruction cache. For this evaluation we leave the compiler parameter preferred subfunction size at the default value of 256. The geometric mean of the comparison is 0.92, which means that on average a direct mapped instruction cache is a slightly better solution for small caches.

Figure 8 compares a 2-way cache with LRU replacement with a direct mapped instruction cache. Compared to Figure 5 we see different benchmarks benefitting from associativity. Furthermore, with the larger cache only a single benchmark (`statemate`) performed worse with the 2-way cache. For the small configuration, several benchmarks perform worse with a 2-way set associative cache. Therefore, for small caches there is no clear winner when comparing a direct mapped and a 2-way set associative instruction cache. The geometric mean of the comparison is 1.05, which means that on average a 2-way set associative instruction cache is a slightly better solution than a direct mapped instruction cache for small cache sizes.

5.6 Discussion

Single-path code has different characteristics than normal code. We see some of the different characteristics when comparing different caching methods. The method cache, which works not so well in the average-case, is a better fit when using single-path code. Prefetching with a standard cache provides some benefit.

As we see in the results, there is no clear winner for all benchmarks. Therefore, if we use an FPGA as execution platform, we can select an application specific caching method. This is like an application specific instruction set in a processor.

It might also be interesting to compare the different caching methods for single-path code with either cache locking or allocation in a scratchpad memory (SPM). In earlier work, we compared two alternatives to a standard instruction cache: an SPM and a method cache [22].

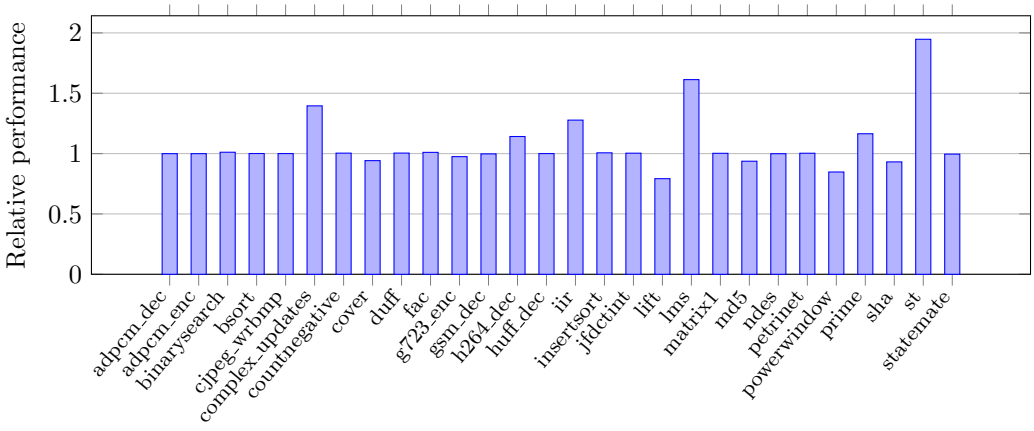

Figure 8 Relative single-path performance comparing a 2-way cache with a direct mapped cache (2 KB size)

The comparison considers the true WCET and the estimated WCET bound. We found that a method cache is preferable to an SPM for the true WCET. However, if WCET bounds are derived by analysis, the WCET bounds for an instruction SPM are often lower than the bounds for a method cache. As there is no WCET overestimation in single-path code, it would be interesting to repeat this comparison with single-path code.

5.7 Reproducing the Results

We think reproducibility is of primary importance in science. As we are working in the context of an open-source project, it is relative easy to provide pointers and a description how to reproduce the presented results.

The T-CREST project is open-source and the README[2] of the Patmos repository provides a brief introduction how to setup an Ubuntu installation for T-CREST and how to build T-CREST from the source. More detailed installation instructions, including setup on Mac OS X, are available in the Patmos handbook [20]. To simplify the evaluation, we also provide a VM[3] where all needed packages and tools are already preinstalled. However, that VM is currently used in teaching and does not contain the latest version of T-CREST, including the scripts for the experiments. Therefore, you need to reinstall and build T-CREST as described in the README.

We have scripted all experiments and host those scripts in the `misc` repository of T-CREST. Details to rerun the experiments are described in a README.[4] The `Makefile` is setup to run the base experiments and for producing the figures as PDFs. Variations can be obtained by changing the respective variables.

[2] `https://github.com/t-crest/patmos`
[3] `http://patmos.compute.dtu.dk/`
[4] `https://github.com/t-crest/patmos-misc/tree/master/experiments/cache_prefetch_lock/wcet2017`

6 Conclusion

In this paper, we compared different caching methods for single-path code. We found that the method cache, which performs not so well in the average case, shows an improvement on some benchmarks when compared to a standard instruction cache. With small cache configurations the best solution is in most cases a set associative instruction cache. When we use an FPGA as execution platform we have the freedom to choose the best caching solution for each individual application.

References

1 J. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of Control Dependence to Data Dependence. In *Proc. 10th ACM Symposium on Principles of Programming Languages*, pages 177–189, Jan. 1983.

2 Bekim Cilku, Daniel Prokesch, and Peter Puschner. A time-predictable instruction-cache architecture that uses prefetching and cache locking. In *Proc. 18th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC) Workshops, 11th IEEE/IFIP International Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS)*, pages 74–79. IEEE CS Press, 2015.

3 Bekim Cilku, Wolfgang Puffitsch, Daniel Prokesch, Martin Schoeberl, and Peter Puschner. Improving performance of single-path code through a time-predictable memory hierarchy. In *Proceedings of the 20th IEEE International Symposium on Real-Time Computing (ISORC 2017)*, Toronto, Canada, May 2017. IEEE.

4 Philipp Degasperi, Stefan Hepp, Wolfgang Puffitsch, and Martin Schoeberl. A method cache for Patmos. In *Proceedings of the 17th IEEE Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC 2014)*, pages 100–108, Reno, Nevada, USA, June 2014. IEEE. doi:10.1109/ISORC.2014.47.

5 Huping Ding, Yun Liang, and Tulika Mitra. Wcet-centric dynamic instruction cache locking. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–6. IEEE, 2014.

6 Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. TACLeBench: A benchmark collection to support worst-case execution time research. In Martin Schoeberl, editor, *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, volume 55 of *OpenAccess Series in Informatics (OASIcs)*, pages 2:1–2:10, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

7 Heiko Falk, Sascha Plazar, and Henrik Theiling. Compile-time decided instruction cache locking using worst-case execution paths. In *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 143–148. ACM, 2007.

8 Reinhold Heckmann and Christian Ferdinand. Worst-case execution time prediction by static program analysis. Technical report, AbsInt Angewandte Informatik GmbH. [Online, last accessed November 2013].

9 Reinhold Heckmann, 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.

10 Stefan Hepp and Florian Brandner. Splitting functions into single-entry regions. In Karam S. Chatha, Rolf Ernst, Anand Raghunathan, and Ravishankar Iyer, editors, *2014*

International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES 2014, Uttar Pradesh, India, October 12-17, 2014, pages 17:1–17:10. ACM, 2014. doi:10.1145/2656106.2656128.

11 Stefan Hepp, Benedikt Huber, Jens Knoop, Daniel Prokesch, and Peter P. Puschner. The platin tool kit - the T-CREST approach for compiler and WCET integration. In *Proceedings 18th Kolloquium Programmiersprachen und Grundlagen der Programmierung, KPS 2015, Pörtschach, Austria, October 5-7, 2015*, 2015.

12 Mingsong Lv, Nan Guan, Jan Reineke, Reinhard Wilhelm, and Wang Yi. A survey on static cache analysis for real-time systems. *Leibniz Transactions on Embedded Systems*, 3(1):05–1, 2016.

13 Daniel Prokesch, Benedikt Huber, and Peter P. Puschner. Towards automated generation of time-predictable code. In Heiko Falk, editor, *14th International Workshop on Worst-Case Execution Time Analysis, WCET 2014, July 8, 2014, Ulm, Germany*, volume 39 of *OASICS*, pages 103–112. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014.

14 Isabelle Puaut and David Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE*, pages 114–123. IEEE, 2002.

15 Peter Puschner and Alan Burns. Writing temporally predictable code. In *Proceedings of the The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, pages 85–94, Washington, DC, USA, 2002. IEEE Computer Society. doi:10.1109/WORDS.2002.1000040.

16 Peter Puschner, Daniel Prokesch, Benedikt Huber, Jens Knoop, Stefan Hepp, and Gernot Gebhard. The T-CREST approach of compiler and WCET-analysis integration. In *9th Workshop on Software Technologies for Future Embedded and Ubiquitious Systems (SEUS 2013)*, pages 33–40, 2013.

17 Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, 2007.

18 Martin Schoeberl. A time predictable instruction cache for a Java processor. In *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004)*, volume 3292 of *LNCS*, pages 371–382, Agia Napa, Cyprus, October 2004. Springer. doi:10.1007/b102133.

19 Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, Stefan Hepp, Benedikt Huber, Alexander Jordan, Evangelia Kasapaki, Jens Knoop, Yonghui Li, Daniel Prokesch, Wolfgang Puffitsch, Peter Puschner, André Rocha, Cláudio Silva, Jens Sparsø, and Alessandro Tocchi. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471, 2015. doi:http://dx.doi.org/10.1016/j.sysarc.2015.04.002.

20 Martin Schoeberl, Florian Brandner, Stefan Hepp, Wolfgang Puffitsch, and Daniel Prokesch. Patmos reference handbook. Technical report, Technical University of Denmark, 2014.

21 Martin Schoeberl, Pascal Schleuniger, Wolfgang Puffitsch, Florian Brandner, Christian W. Probst, Sven Karlsson, and Tommy Thorn. Towards a time-predictable dual-issue microprocessor: The Patmos approach. In *First Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011)*, pages 11–20, Grenoble, France, March 2011.

22 Jack Whitham and Martin Schoeberl. WCET-based comparison of an instruction scratchpad and a method cache. In *Proceedings of the 10th Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS 2014)*, Reno, Nevada, USA, June 2014. doi:10.1109/ISORC.2014.48.