Improving Performance of Single-path Code Through a Time-predictable Memory Hierarchy

Bekim Cilku*, Wolfgang Puffitsch‡, Daniel Prokesch*, Martin Schoeberl‡ and Peter Puschner*
*Vienna University of Technology, Vienna, Austria
{bekim, daniel, peter}@vmars.tuwien.ac.at
‡Technical University of Denmark, Copenhagen, Denmark
{wopu, masca}@dtu.dk

***Abstract*—Deriving the Worst-Case Execution Time (WCET) of a task is a challenging process, especially for processor architectures that use caches, out-of-order pipelines, and speculative execution. Despite existing contributions to WCET analysis for these complex architectures, there are open problems. The single-path code generation overcomes these problems by generating time-predictable code that has a single execution trace. However, the simplicity of this approach comes at the cost of longer execution times.**

This paper addresses performance improvements for single-path code. We propose a time-predictable memory hierarchy with a prefetcher that exploits the predictability of execution traces in single-path code to speed up code execution. The new memory hierarchy reduces both the cache-miss penalty time and the cache-miss rate on the instruction cache. The benefit of the approach is demonstrated through benchmarks that are executed on an FPGA implementation.

I. Introduction

Computing the Worst-Case Execution Time (WCET) of a task becomes mandatory when timing guarantees on task-completion deadlines have to be given. Unfortunately WCET computation is a complex undertaking, especially for systems that use caches, out-of-order pipelines, and control speculation. Although these features complicate timing analyses, the chase for higher execution performance makes their presence almost inevitable. For software running on such complex architectures, a high-quality WCET estimation would require a highly complex WCET analysis that would cover all possible system states that can emerge at runtime. However, exploring all possible hardware states quickly leads into an unmanageable state-space explosion.

State-of-the-art WCET tools avoid the complexity problem by modeling the real hardware with abstracted models [1]. Abstract interpretation is a static program analysis method that executes abstract version of the program on a set of abstract values [2]. In case of WCET analysis, an abstract domain and abstract transition functions are defined to reduce the complexity of hardware modeling and also reduce the set of states that have to be analyzed. The abstraction is considered successful if the set of abstract states compactly represents the real hardware states at any program point. However, abstraction also leads to information loss which, in turn, results in lots of unclassified model states and the computation of pessimistic WCET estimates. In the end, task CPU-time reservation based on these pessimistic estimates leads to poor processor utilization and overly pessimistic results on schedulability tests.

One approach that eliminates the problems of WCET analysis for traditional code is the generation and use of single-path code [3]. Single-path code generation eliminates input-dependent control flow in the code. This is achieved through conversion of all input-dependent alternatives of the source code into pieces of sequential code. Also, all loops with input-data dependent termination conditions are transformed into loops with constant iteration counts. This code generation strategy forces all executions to follow the same instruction trace, thus eliminating control-flow induced variations in execution time. WCET analysis for the resulting code is trivial—one only needs to run the code once (on its unique path) and measure. This single execution run also identifies the stream of instructions that every execution will follow. The properties of single-path code make the process of real-time system design simple, composable and scalable [4]. A major drawback is, however, that single-path code may end up with quite long execution times for programs with many input-data dependent control decisions.

This paper presents a new time-predictable memory hierarchy that improves the performance of single-path code. The memory concept exploits the pre-runtime knowledge about the execution trace that can be derived from the single-path code. Using this knowledge, the prefetching algorithm predictably brings instructions into the cache before they are required. This enables the prefetcher to work efficiently for the whole single-path code without polluting the cache at any moment. However, to reach the best performance of the memory hierarchy, the cache memory needs to be adapted. Therefore, we propose a modified cache design that allows regular instruction fetching (and caching) and prefetching to work in parallel without interference.

The paper is organized as follows. Section II provides a brief overview of single-path code, cache memory, and prefetching. Section III describes the architecture of the memory hierarchy and its components. An evaluation of the memory hierarchy follows in Section IV, while related work is presented in Section V. Section VI concludes the paper.

II. Background

A. Single-path Paradigm

The essence of single-path conversion is to eliminate the complexity of multi-path code analysis. The only hardware support required for this strategy is the provision of *predicated instructions*. Predicated instructions guard the processor-state modification with a predicate condition. At runtime, all predicated instructions are executed, but only those whose predicates evaluate to *true* change the hardware state. If a predicate evaluates to *false*, the instruction behaves like a NOP instruction and does not change the hardware state [5]. Figure 1 illustrates with an example how predicated instructions are used for *if-else* conversion. The value of variable x is conditionally updated, depending on the evaluation of *cond*. Variant (*a*) illustrates conventional code with branch instructions, where only one of the instructions is executed. Variant (*b*) executes both instructions, but only one of them changes the state of x.

The generator for single-path conversion has already been implemented in a compiler back-end [6]. Without going into too much detail, we present the main rules of the single-path conversion in the following.

1) Branch conversion: Input-dependent branches of *if* or *case* semantics consist of two or more mutually exclusive alternatives, where, depending on the program inputs, one alternative is executed. The single-path conversion serializes all alternatives into sequential code, thus forcing the execution to pass all branches [3]. Thereby the values of predicates control which instructions change the computational state. In the conversion of nested alternatives, the predicates of the inner statements are formed by a conjunction of the predicates that lead to those statements [7].

2) Loop conversion: Loops with input-dependent termination condition exit the loop dependent on the program inputs. The single-path approach transforms these loops to loops with constant iteration count, where each iteration has constant execution time [8]. To achieve this, the loop iteration count is set to a fixed maximum count and predicates, generated from exit condition of the old loop, are used to control the statements of the loop body. Hence, when the exit condition of the original loop becomes true, the new loop continues to iterate, but does not change the state anymore.

```
     cond := ...
     if (cond) goto L2
L1:  x := x + 1
     goto L3
L2:  x := x - 1
L3:  ...
```

(a) Update of x with branch instruction.

```
          cond := ...
(!cond)   x := x + 1
( cond)   x := x - 1
          ...
          ...
          ...
```

(b) Update of x with predicated instructions.

Fig. 1: Example of code with branch instruction and predicated instructions.

3) Function conversion: Function calls are unconditional, but the predicate of the call site is passed to the called function. For predicates evaluating to *false*, the function does not change the processor state, even though the whole function is executed. Hence, the function predicate forms the initial precondition for all statements of the function.

B. Understanding Cache Behavior

Caches are small and fast memories that are used to improve the performance of the memory system based on the principle of locality [9]. Locality can be observed in the temporal and spatial behavior of the execution of code. Temporal locality means that code executed now is likely to be referenced again in the near future. This type of behavior is observed in program loops and functions when instructions that are already in the cache (from prior iterations or calls) are reused. Spatial locality means that instructions whose addresses are close by will tend to be referenced in temporal proximity. Caches exploit locality by storing only a copy of code fractions while the entire executable code is held in main memory.

As an application is executed over the time, the processor references the memory by sending memory addresses. Referenced instructions that are found in the cache are called *hits*, while those that are not in the cache are called *misses*. In case of a cache miss, the processor stalls until the instructions are fetched from main memory. The time needed to transfer instructions from the main memory into the cache is called *miss penalty*. The fraction of cache accesses that result in misses is called the *miss rate*. Cache memories can be organized as fully associative (memory blocks can be stored in any cache line), set associative (cache lines are grouped into sets), or direct mapped (memory blocks can be placed in only one cache line) [10]. The very first reference to a memory block always results in a cache miss, a so-called *compulsory miss*. For a fully-associative cache, when the cache is full, cached instructions must be evicted to create space for the incoming ones. If evicted instructions are referenced again, a *capacity miss* occurs. For set-associative and direct-mapped caches, a *conflict miss* occurs when more memory blocks than the associativity map to the same set, even though the cache is not full.

C. Instruction Prefetching

Prefetching is considered an efficient strategy to mask the large latencies of memory accesses: instructions are loaded into the cache before they are needed [9]. Information that is fetched before it is referenced is called *prefetched*. Prefetching can be implemented in software or hardware. A software prefetcher initiates prefetching through explicit prefetch instructions. The challenge of software prefetching is the placement of the prefetch instructions to guarantee that prefetching happens at the right time. Hardware prefetching is initiated by hardware logic that monitors the processor accesses to the memory. In contrast to software prefetching, hardware prefetching has no code overhead, but often generates more unnecessary traffic than the software approach—hardware prefetching speculates on future memory accesses without using compile-time information.

D. Patmos and the T-CREST Platform

The proposed prefetcher is intended for time-predictable systems. Therefore, we decided to build upon a time-predictable processor architecture. We use the Patmos [11] processor, which is part of the T-CREST platform [12]. T-CREST is a multiprocessor system developed especially for real-time systems. All components are built to be time-predictable and allow WCET analysis. A T-CREST platform consists of several processor cores that are connected to two networks-on-chip (NoC): (1) the Argo NoC and (2) the memory tree. The Argo NoC [13] supports time-predictable message passing between processor local memories. For code and larger data structures, all processors are connected to shared DRAM memory through the Bluetree memory tree [14] and the Predator memory controller [15], or to a shared SRAM memory through a distributed memory arbiter [16].

Patmos is a dual-issue RISC processor with a simple 5-stage pipeline. Patmos is designed to avoid any timing anomaly. All instructions of Patmos can be predicated to support the single-path code paradigm. Furthermore, the execution time of an instruction is constant, independent of the predicate value. Patmos includes caches designed to simplify WCET analysis. Stack allocated data is cached in the so-called stack cache [17]. For instructions, the cache of Patmos is configurable to include either a method cache [18] or a standard instruction cache. In this work, we use Patmos with a standard instruction cache.

Most of the T-CREST hardware is open-source under the industry-friendly simplified BSD license. The build instructions for the whole platform can be found at https://github.com/t-crest/patmos and (in more detail) in the Patmos handbook [19].

III. Memory Hierarchy for Single-path Code

In this section, we describe the instruction path of the new designed memory hierarchy with the time-predictable prefetcher. Figure 2 shows a high-level architecture diagram of the instruction cache and the prefetcher as well as their interconnections to the CPU and main memory. Both components are designed to be time-predictable and allow efficient execution of single-path code.

A. Architecture of the Instruction Cache

The proposed instruction cache is a small dual-port on-chip memory that accepts and processes address references issued from both the processor for reads and the prefetcher for writes. Each reference that arrives at the cache is first compared with the entries of the tag table for a possible match. Depending on the source of the request, the cache performs one of two types of actions in case of a miss: (1) on a *processor cache miss*, the cache works like a conventional cache—it stalls the processor and forwards the processor request to the external main memory; (2) on a *prefetch cache miss*, the cache forwards the request of the prefetcher directly to the main memory without disturbing the operation of the processor. Distinguishing these two types of misses in the cache enables the prefetcher and the processor fetch stage to work in parallel, without interfering the execution. Thus, while the fetch stage accesses instructions that are already in the cache, the prefetcher initiates the prefetching of upcoming cache lines. Overlapping these two stages reduces both the *miss-penalty time* and therefore the overall execution time. Moreover, when the processor executes a loop that fits into the cache, the prefetcher continues to fill the rest of the cache with the upcoming cache lines. In this case, the memory hierarchy not only eliminates the miss penalty but also reduces the *cache-miss rate*, as the cache-loading of instructions is finished before the instructions will be referenced. Note however, that the number of cache misses that can be eliminated depends on the time the processor spends executing the loop and on the cache space that can be filled without evicting parts of the loop.

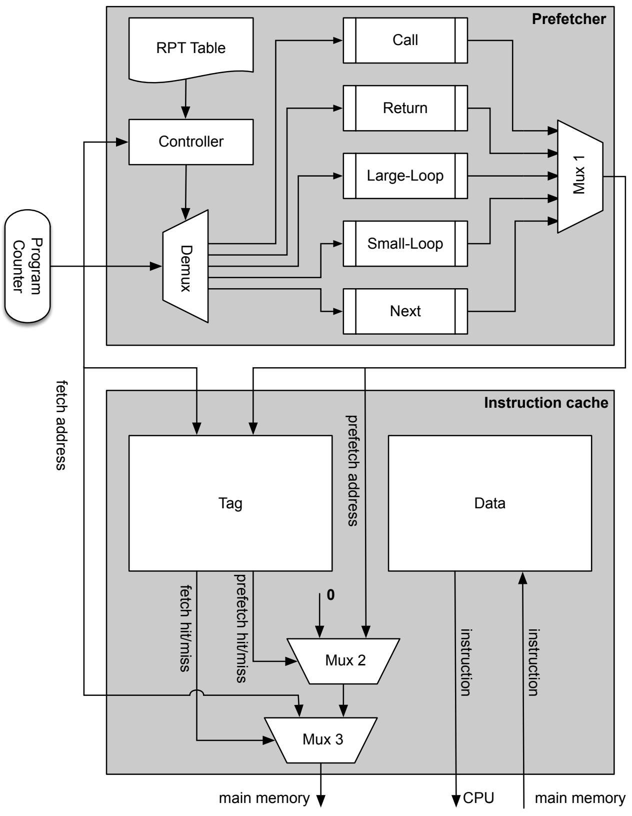


Fig. 2: Memory hierarchy architecture with prefetcher

Both types of address references can be sent to the cache at the same time. If both result in a cache miss, then only one can be forwarded to the main memory. In that case, the cache gives priority to the request from the processor, i.e., it restricts the prefetcher to read the memory only when the bus is idle. The hardware mechanism for this arbitration is implemented by a cascade of two multiplexers (Mux 2 and Mux 3 in Figure 2).

One has to make sure that the prefetcher does not evict the cache line that holds the currently executing instruction. To this end, the cache line number of each prefetching request is compared with the line number of the currently executing instruction. In case of a match, the prefetch request is dropped.

B. Time-predictable Prefetching Scheme for Single-path Code

For the sake of efficiency and time-predictability, the prefetcher combines both sequential and non-sequential prefetching. Simple, sequential prefetching is used whenever sequential fragments of code are executed. In sequential prefeching, the prefetcher pre-loads the cache lines sequentially, starting from the currently fetched cache line [20].

When the processor jumps to a new location, the prefetcher applies non-sequential prefetching. Non-sequential prefetching is, in general, tricky due to the difficulty of predicting branch targets of dynamic control-flow instructions. Single-path code, however, has no dynamic control decisions, which makes it ideal for predictable prefetching. The clearly defined control flow of single path code enables the designer to use control-flow information extracted by a simple pre-runtime analysis to control the non-sequential operation of the prefetcher at runtime. Our prefetcher therefore uses control-flow information extracted before runtime to control both the target memory addresses and the triggering times of prefetch operations, thus making the whole prefetching scheme fully predictable for every cache line.

C. Architecture of the Prefetcher

To keep the development of the prefetcher independent from the processor and compiler, and to avoid the overhead of executing additional instructions, we have decided to implement the prefetcher as a separate hardware unit. As shown in Figure 2, the prefetcher consists of the *Controller*, three non-sequential prefetch-components (*Call-prefetcher*, *Return-prefetcher*, *Large-loop-prefetcher*), the sequential prefetch-component (*Next-prefetcher*), the *Small-loop-prefetcher*, and the *Reference Prediction Table* (RPT).

The controller watches the stream of execution, to determine the prefetch-triggering times, and to select the appropriate prefetch component for the target address calculation. As the code executes, all address references issued by the program counter (PC) are monitored by the controller. Whenever the execution switches to a new cache line, the controller triggers the prefetcher and forwards the PC to the RPT to check for a possible match. If the search in the table yields a hit, which means that the upcoming cache line is non-sequential, the calculation of the target is performed by one of the non-sequential components of the prefetcher. In case of miss, the sequential component gets active.

Call-prefetcher, Return-prefetcher, and Large-loop-prefetcher form the non-sequential part of the prefetcher. Call-prefetcher is triggered when the prefetch target is a cache line of a function whose target address is stored in the RPT. This component has a small stack memory that holds the address of the caller. Thus, when the execution reaches the end of the function, the Return-prefetcher will use the return address from the stack. Using a stack eliminates the need to store function-return addresses

TABLE I: Reference Prediction Table

Index	Trigger	Type	Destination	Iteration	Next	Count	Depth
0	8310	call	8305	-	1	-	-
1	8350	ret	-	-	-	-	-
2	8780	big-loop	8205	24	2	-	1
3	9215	small-loop	-	-	-	15	-

in the RPT. Large-loop-prefetcher is triggered for loops that are larger than the cache. For such loops, the prefetecher will always prefetch the loop header, until the known number of iterations is reached. In case of nested loops, the prefetcher keeps the iteration number of outer loops in registers, in order to track if the loops are still alive.

The Next-prefetcher calculates the next consecutive target address by just incrementing the address of the current cache line. As compilers genarlly try to generate sequential code layouts and the single-path conversion does a lot of code serialization, this prefetch component is the most active one.

The last prefetch component, the Small-loop-prefetcher, utilizes the idle bus cycles when the processor is busy executing loops that are smaller than the cache size. In contrast to the other prefetch components, the Small-loop-prefetcher continues prefetching for more than one cache line. To optimize performance, the number of cache lines it prefetches is determined in the pre-runtime analysis (e.g., to avoid the eviction of cache lines that are part of the active loop) and stored in the RPT.

The RPT is a small table that holds code-related data to control the prefeching. The table has entries of eight columns that store the following information:

- Index: index of the table entry.
- Trigger: triggering address for the non-sequential prefetcher.
- Type: the type of prefetch component to work on the prefetch request.
- Destination: target address for non-sequential prefetching.
- Iteration: number of loop iterations.
- Next: next non-sequential entry.
- Count: number of cache lines to be prefetched when executing a small loop.
- Depth: loop-nesting level for nested loops.

Table I shows an example of an RPT with all four possible types of entries. Because single-path code has a single execution trace, the entries of the RPT can be ordered in the same way they are executed. This eliminates the need to organize the RPT as a look-up table. Thus, whenever a PC reference needs to be compared, only one table entry has to be tested. The position of the current entry is kept in a register; the value of this pointer changes as the execution proceeds.

D. Generating the Reference Prediction Table

The RPT generation is illustrated in Figure 3. The single-path code generator of the Patmos compiler emits single-path code for selected *single-path entry* functions. That is, the entry functions themselves and the functions called within are produced to execute on a singleton execution path. In addition, each loop is generated with a single exit edge at the

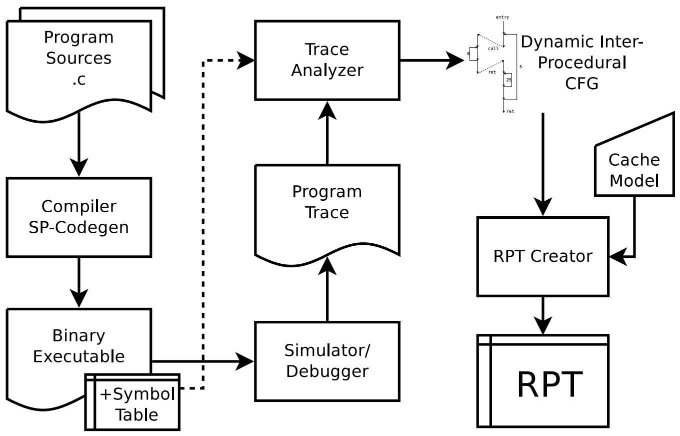


Fig. 3: Work-flow for generation of the RPT.

end of the loop body. This setting minimizes the number of control flow changes and the number non-sequential memory accesses.

The RPT is created by analyzing the singleton execution trace of the single-path code. Thereby a trace of the program counter is recorded. We use our cycle-accurate simulator `pasim`, but a simple instruction set simulator or a debugging facility could be used for this purpose as well. The program trace is passed to the trace analyzer. The analyzer uses the information of the symbol table of the executable to map address ranges to the respective functions. It extracts a subtrace for an execution of a single-path task and constructs a dynamic interprocedural control-flow graph, in which values of the program counter are represented as nodes and the transitions between machine instructions are represented as edges. Each edge is labeled with the global count for the transition. The analysis furthermore identifies transitions between functions, loop nests, and it computes the iteration count for each loop.

The RPT creator uses this information to select the points in the code that represent control-flow changes, i.e., backward branches to loop headers (at the end of the loop body), call sites, and returns from functions. As the RPT creator works on single-path code, there are no if-else constructs that would require treatment. By supplying a cache model, the table creator computes the trigger and destination values using the cache line size. Additional parameters, namely the cache size and the associativity are required for classification of loop-type entries as either large-loop or small-loop, including the count for the latter. RPT entries are grouped by function and sorted by trigger within each function.

IV. Evaluation

This section presents the results of the evaluation of the memory hierarchy. We implemented the memory hierarchy in hardware and then integrated it into the T-CREST platform. We synthesized the design and uploaded on an DE2-115 FPGA board with Cyclone EP4CE115 device. The hardware consisted of Patmos dual-issue processor, cache memory and main memory. The processor was configured as single-issue, while the main memory was an SRAM with size of 2MB. For our experiment we used the Mälardalen WCET benchmarks [21]. We generated the code for the benchmarks with the single-path code generator of the Patmos compiler for the T-CREST platform, which is based on LLVM 3.4 [6]. Following the code generation, we analyzed the binary outputs and generated the RPT for each benchmark. The size of the RPT table for Mälardalen benchmarks ranged from 2 to 178 entries. In the experiments, we evaluated the whole set of Mälardalen benchmarks, except the *fac*, *recursion*, *matmul* and *st* benchmarks. The latter were excluded as our prototype tools do not handle recursion and calls to external libraries.

Figure 4 presents normalized values of benchmark execution times to show the timing improvement that can be achieved when a memory hierarchy with the prefetcher is used. All benchmarks were run on two platforms that were identical, except for the prefetcher which was only used in one setup. On both cases the cache size was 4 KB. In the presence of the prefetcher, we observed run-time improvements ranging from less than 1% up to 16%. Note that all experiments show execution-time improvements. This is due to the pre-planning of the prefetching activities that prevents the prefetcher from generating cache pollution or useless memory traffic that could slow down the execution.

Even though the prefetcher guarantees improvement, the scale of improvement depends on the structure of the single-path code, on the size of the cache, and on the configuration of the cache. Some benchmarks have small performance improvements (see Figure 4). For those benchmarks, the rate of data-cache misses is much higher than the miss rate of the instruction cache.

To illustrate this, Figure 5 compares the execution times of a binary search algorithm for arrays of different size. The algorithm was executed with four different on-chip memory configurations with the same size of 4 KB. The first configuration uses instruction and data caches only. The second configuration additionally includes our time-predictable prefetcher. The third configuration uses an instruction cache and a data scratchpad instead of the data cache. The last configuration includes the instruction prefetcher, an instruction cache, and a data scratchpad. For the last two configurations, all required data were loaded into the scratchpad before the experimental execution was started. The result shows that as the number of elements in the array increases, the improvement gap between configurations with data cache and scratchpad also increases.

Figure 6 presents the influence of the cache size on the execution time in the presence of our prefetcher. We selected benchmarks for which the effect of changing the cache size could be observed. We used caches sizes of 1, 2, 4, 16 and 32 KB, both in direct-mapped or two-way associative configuration with LRU replacement and cache line sizes of 16 or 32 bytes. The results show that the prefetcher is most effective if the size of the code is larger than the size of cache—improvements can reach 17% for *cover* benchmark. Once the cache becomes larger than the code size, the prefetcher

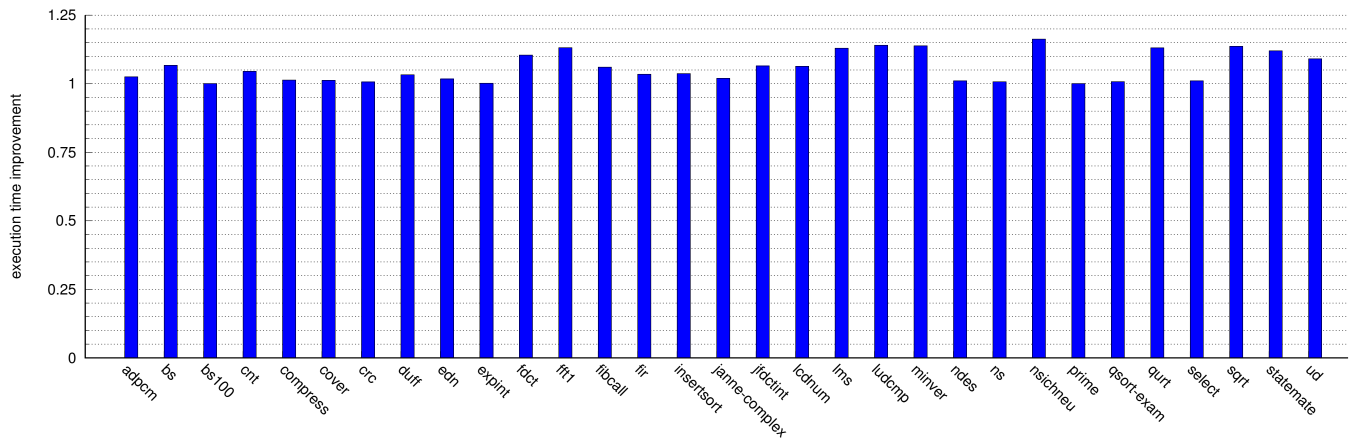


Fig. 4: Execution time improvement for Mälardalen benchmarks

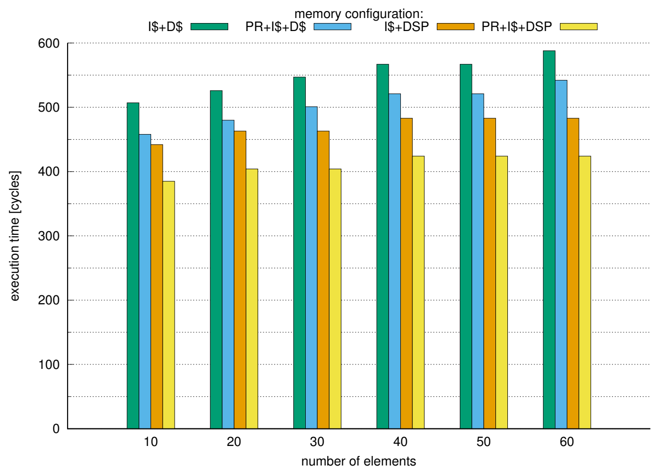


Fig. 5: Binary search algorithm for different memory configurations (I$: instruction cache, D$: data cache, PR: prefetcher, DSP: data scratchpad)

is effective only on compulsory misses and the impact on execution-time improvement gets very small. However, realistic applications will always be larger than the instruction cache. Otherwise, the program could simply be loaded into an on-chip memory.

From the evaluations we can conclude that the prefetcher always improves the execution time of single-path code. The magnitude of the improvements, however, depends on the structure of the code, the cache size, and the size of the code.

V. Related Work

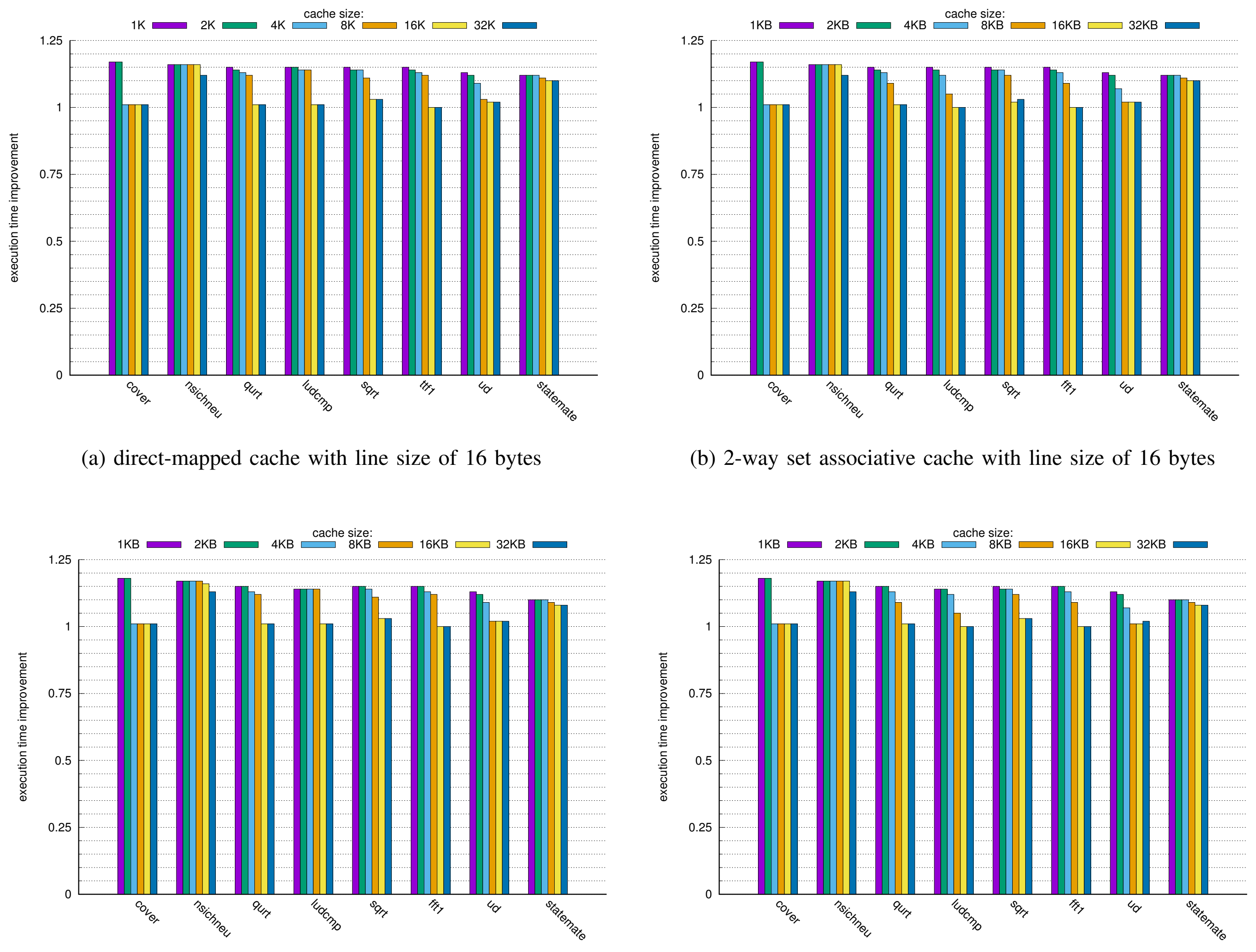
In general, research on time-predictability of the memory hierarchies can be observed to follow two directions. On one side, there is ongoing research on building analysis tools for analyzing the timing behavior of conventional memory architectures [22]. On the other side, we see the development of new techniques of using memory hierarchies in a predicable way to simplify the analysis. Here we limit the scope to techniques that improve the timing predictability of on-chip memories.

Cache locking loads memory contents into cache the and locks it to ensure that it will remain unchanged afterward. The benefit of such an approach is that all accesses to the locked cache lines will always result into cache hits. The cache content can be locked entirely [23], [24] or partially for the whole system lifetime (static cache locking) or it can be changed at runtime (dynamic cache locking) [25], [26]. Although cache locking increases predictability, it reduces performance by restricting the temporal locality of the cache to a set of locked cache lines.

Scratchpads present another alternative for on-chip memory [27]. Their content is managed through software, which makes them predictable concerning the access time. They are also smaller and consume less energy than caches of the same size. Scratchpads are used in the PRET and MERASA processors where they replace the cache [28], [29]. The disadvantage of using a scratchpad is the overhead due to code allocation and the necessity to correct the execution flow with additional control-flow instructions as code parts are dislocated into different memory address spaces.

The method cache is another alternative for replacing the conventional cache. It was first proposed by Schoeberl in [30] for Java programs and later extended for functions and procedures of procedural languages [18]. In contrast to a conventional cache, the method cache uses a granularity of a method, thus guaranteeing that except for method calls and returns all memory accesses will result in cache hits. However, the strategy generates large overheads for cases when the whole method is loaded into the cache, but only a small part of it is executed.

Lee et al. [31] suggest a dual-mode instruction prefetch scheme as an alternative to using an instruction cache. The idea is to improve the WCET by associating a thread to each instruction block that is part of the WCET. Threads are generated by the compiler and remain static during the task execution. Such a solution provides good performance only



(a) direct-mapped cache with line size of 16 bytes

(b) 2-way set associative cache with line size of 16 bytes

(c) direct-mapped cache with line size of 32 bytes

(d) 2-way set associative cache with line size of 32 bytes

Fig. 6: Cache size impact on prefetch efficiency

when the block execution times are longer than the prefetching time.

Kou et al. [32] propose TickPAD, a predictable memory for synchronous languages. This memory system consists of four components where each one is used to load the local memory based on the upcoming memory access pattern. However, having fully associative storage for the reference table makes this solution very expensive. Also, stalling the execution until the whole loop is fetched into the associative loop memory degrades system performance.

Garside and Aidsley [14] have implemented a stream prefetcher located between a memory NoC and shared main memory. Such a position enables the prefetcher to listen to the address references of all cores and to provide service for all of them. Even though the prefetcher improves the performance and is part of a time-predictable platform, the only guarantee that can be given is that the presence of prefetcher does not affect the WCET negatively.

VI. Conclusion

To overcome the problem of long execution times of single-path code, we have proposed a new memory hierarchy organization. This memory architecture reduces the cache-miss penalty time and the cache-miss rate by prefetching instructions into the cache before they are required for execution. The key idea of the proposed approach is to use the fully predictable execution behavior of single-path code to execute full control on the operation of the prefetcher both in the time domain—determining when a prefetch is initiated—and value domain—determining from which addresses instructions are prefetched.

The proposed modification of the cache enables both fetching and prefetching to be performed in parallel. The solution allows the prefetcher to prefetch every cache line of the code, which maximizes the efficiency of the prefetcher by utilizing every idle cycle of the bus. The design of the prefetcher guarantees that for code of any structure, the presence of the prefetcher will never increase the execution time of the generated single-path code.

We have evaluated the prefetcher by implementing it in an FPGA together with the Patmos processor. Execution of single-path code with the prefetcher on the FPGA shows improvement of the execution of up to 17%.

Acknowledgment

This paper was partially funded by the EU COST Action IC1202: Timing Analysis on Code Level (TACLe), the European Project FP7-ICT-2013-10-610535 AMADEOS, ARTEMIS Joint Undertaking (JU) project EMC^2 under grant agreement n° 621429 and the European Union's 7th Framework Programme under grant agreement no. 288008: Time-predictable Multi-Core Architecture for Embedded Systems (T-CREST).

Source Access

Chisel source code for the hardware can be found at: https://github.com/t-crest/patmos/tree/icache_with_prefetcher/hardware/src/icache

References

[1] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, p. 36, 2008.

[2] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 1977, pp. 238–252.

[3] P. Puschner and A. Burns, "Writing temporally predictable code," in *Object-Oriented Real-Time Dependable Systems, 2002.(WORDS 2002). Proceedings of the Seventh International Workshop on*. IEEE, 2002, pp. 85–91.

[4] P. Puschner, B. Cilku, and D. Prokesch, "Constructing time-predictable MPSoCs: Avoid conflicts in temporal control," in *Embedded Multicore/Many-core Systems-on-Chip (MCSoC), 2016 IEEE 10th International Symposium on*. IEEE, 2016, pp. 321–328.

[5] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W.-M. W. Hwu, "A comparison of full and partial predicated execution support for ILP processors," in *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*. IEEE, 1995, pp. 138–149.

[6] D. Prokesch, S. Hepp, and P. Puschner, "A generator for time-predictable code," in *2015 IEEE 18th International Symposium on Real-Time Distributed Computing*. IEEE, 2015, pp. 27–34.

[7] P. Puschner, R. Kirner, B. Huber, and D. Prokesch, "Compiling for time predictability," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2012, pp. 382–391.

[8] P. Puschner, "Transforming execution-time boundable code into temporally predictable code," in *Design and Analysis of Distributed Embedded Systems*. Springer, 2002, pp. 163–172.

[9] A. J. Smith, "Cache memories," *ACM Computing Surveys (CSUR)*, vol. 14, no. 3, pp. 473–530, 1982.

[10] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[11] M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, C. W. Probst, S. Karlsson, and T. 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)*, Grenoble, France, March 2011, pp. 11–20.

[12] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, S. Hepp, B. Huber, A. Jordan, E. Kasapaki, J. Knoop, Y. Li, D. Prokesch, W. Puffitsch, P. Puschner, A. Rocha, C. Silva, J. Sparsø, and A. Tocchi, "T-CREST: Time-predictable multi-core architecture for embedded systems," *Journal of Systems Architecture*, vol. 61, no. 9, pp. 449–471, 2015.

[13] E. Kasapaki, M. Schoeberl, R. B. Sørensen, C. T. Müller, K. Goossens, and J. Sparsø, "Argo: A real-time network-on-chip architecture with an efficient GALS implementation," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, pp. 479–492, 2016.

[14] J. Garside and N. C. Audsley, "Investigating shared memory tree prefetching within multimedia noc architectures," in *Memory Architecture and Organisation Workshop*, 2013.

[15] M. D. Gomony, B. Akesson, and K. Goossens, "Architecture and optimal configuration of a real-time multi-channel memory controller," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, 2013, pp. 1307–1312.

[16] M. Schoeberl, D. V. Chong, W. Puffitsch, and J. Sparsø, "A time-predictable memory network-on-chip," in *Proceedings of the 14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014)*, Madrid, Spain, July 2014, pp. 53–62.

[17] S. Abbaspour, F. Brandner, and M. Schoeberl, "A time-predictable stack cache," in *Proceedings of the 9th Workshop on Software Technologies for Embedded and Ubiquitous Systems*, 2013.

[18] P. Degasperi, S. Hepp, W. Puffitsch, and M. Schoeberl, "A method cache for Patmos," in *2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. IEEE, 2014, pp. 100–108.

[19] M. Schoeberl, F. Brandner, S. Hepp, W. Puffitsch, and D. Prokesch, "Patmos reference handbook," Technical University of Denmark, Tech. Rep., 2014.

[20] A. J. Smith, "Sequential program prefetching in memory hierarchies," *Computer*, vol. 11, no. 12, pp. 7–21, 1978.

[21] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET benchmarks – past, present and future," B. Lisper, Ed. Brussels, Belgium: OCG, Jul. 2010, pp. 137–147.

[22] M. Lv, N. Guan, J. Reineke, R. Wilhelm, and W. Yi, "A survey on static cache analysis for real-time systems," *Leibniz Transactions on Embedded Systems*, vol. 3, no. 1, pp. 05–1, 2016.

[23] H. Falk, S. Plazar, and H. 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*. ACM, 2007, pp. 143–148.

[24] S. Plazar, J. C. Kleinsorge, P. Marwedel, and H. Falk, "WCET-aware static locking of instruction caches," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. ACM, 2012, pp. 44–52.

[25] I. Puaut, "WCET-centric software-controlled instruction caches for hard real-time systems," in *18th Euromicro Conference on Real-Time Systems (ECRTS'06)*. IEEE, 2006, pp. 10–pp.

[26] H. Ding, Y. Liang, and T. Mitra, "WCET-centric dynamic instruction cache locking," in *Proceedings of the conference on Design, Automation & Test in Europe*. European Design and Automation Association, 2014, p. 27.

[27] L. Wehmeyer and P. Marwedel, "Influence of memory hierarchies on predictability for time constrained embedded software," in *Design, Automation and Test in Europe*. IEEE, 2005, pp. 600–605.

[28] I. Liu, J. Reineke, and E. A. Lee, "A PRET architecture supporting concurrent programs with composable timing properties," in *2010 Conference Record of the Forty Fourth Asilomar Conference on Signals, Systems and Computers*. IEEE, 2010, pp. 2111–2115.

[29] S. Metzlaff, S. Uhrig, J. Mische, and T. Ungerer, "Predictable dynamic instruction scratchpad for simultaneous multithreaded processors," in *Proceedings of the 9th workshop on MEmory performance: DEaling with Applications, systems and architecture*. ACM, 2008, pp. 38–45.

[30] M. Schoeberl, "A time predictable instruction cache for a java processor," in *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer, 2004, pp. 371–382.

[31] M. Lee, S. L. Min, C. Y. Park, Y. H. Bae, H. Shin, and C.-S. Kim, "A dual-mode instruction prefetch scheme for improved worst case and average case program execution times," in *Real-Time Systems Symposium, 1993., Proceedings*. IEEE, 1993, pp. 98–105.

[32] M. Kuo, P. Roop, S. Andalam, and N. Patel, "Precision timed embedded systems using tickpad memory," in *2013 13th International Conference on Application of Concurrency to System Design*. IEEE, 2013, pp. 206–215.