Designing a Time-Predictable Memory Hierarchy for Single-Path Code

Bekim Cilku
Institute of Computer Engineering
Vienna University of technology
A 1040 Wien, Austria
bekim@vmars.tuwien.ac.at

Peter Puschner
Institute of Computer Engineering
Vienna University of technology
A 1040 Wien, Austria
peter@vmars.tuwien.ac.at

ABSTRACT

Trustable Worst-Case Execution-Time (WCET) bounds are a necessary component for the construction and verification of hard real-time computer systems. Deriving such bounds for contemporary hardware/software systems is a complex task. The single-path conversion overcomes this difficulty by transforming all unpredictable branch alternatives in the code to a sequential code structure with a single execution trace. However, the simpler code structure and analysis of single-path code comes at the cost of a longer execution time. In this paper we address the problem of the execution performance of single-path code. We propose a new instuction-prefetch scheme and cache organization that utilize the "knowledge of the future" properties of single-path code to reduce the main memory access latency and the number of cache misses, thus speeding up the execution of single-path programs.

Keywords

hard real-time systems, time predictability, memory hierarchy, prefetching, cache memories

1. INTRODUCTION

Embedded real-time systems need safe and tight estimations of the Worst Case Execution Time (WCET) of time-critical tasks in order to guarantee that the deadlines imposed by the system requirements are meet. Missing a single deadline in such a system can lead to catastrophic consequences.

Unfortunately, the process of calculating the WCET bound for contemporary computer systems is, in general, a complex undertaking. On the one hand, the software is written to execute fast – it is programmed to follow different execution paths for different input data. Those different paths, in general, have different timing, and analyzing them all can lead to cases where the analysis cannot produce results of the desired quality. On the other hand, the inclusion of hardware features (cache, branch prediction, out-of-order execution, and pipelines) extend the analysis with state dependencies and mutual interferences; a high-quality WCET analysis has to consider the interferences of all mentioned hardware features to obtain tight timing analysis. The state-of-the-art tools for WCET analysis are using a highly integrated approach by considering all interferences caused by hardware state interdependencies [4]. Keeping track of all possible interferences and also the hardware state history for the whole code in an integrated analysis can lead to a state-space explosion and will make the analysis infeasible. An effective approach that would allow the tool to decompose the timing analysis into compositional components is still lacking [1].

One strategy to avoid the complexity of the WCET analysis is the single-path conversion [12]. The single-path conversion reduces the complexity of timing analysis by converting all input-depended alternatives of the code into pieces of sequential code. This, in turn, eliminates control-flow induced variations in execution time. The benefit of this conversion are the predictable properties that are gained with the code transformation. The new generated code has a single execution trace that forces the execution time to become constant. To obtain information about the timing of the code it is sufficient to run the code only once and to identify the stream of the code execution which is repeated on any other iteration.

Large programs that have been converted into single-path code can be decomposed into smaller segments where each segment can be easily analyzed for its worst-case timing in separation. This contrasts the analysis of traditional code, where a decomposition into segments may lead to highly pessimistic timing-analysis results, because important information about possible execution paths and information about how these execution paths within one segment influence the feasible execution paths and timings in subsequent segments gets lost at segment boundaries. In single-path code, each code segment has a constant trace of execution and the initial hardware states for each segment can be easily calculated, because there are no different alternatives of the incoming paths that can lead to a loss of information during a (de)compositional analysis. However, the advantage of generating compositional code that allows for a highly accurate, low-complexity analysis comes at the cost of a longer execution time of the code.

The long latency of memory accesses is one of the key performance bottlenecks of contemporary computer systems. While the inclusion of an instruction cache is a crucial first step to bridge the speed gap between CPU and main memory, this is still not a complete solution – cache misses can result in significant performance losses by stalling the CPU until the needed instructions are brought into the cache.

For such a problem, prefetching has been shown to be an effective solution. Prefetching can mask large memory latencies by loading the instructions into the cache before they are actually needed [15]. However, to take advantage of this improvement, the prefetching commands have to be issued at the right times – if they are issued too late memory latencies are only partially masked, if they are issued too early, there is the risk that the prefetched instruction will

evict other useful instructions from the cache.

Prefetching mechanisms also have to consider the accuracy, since speculative prefetching may pollute the cache. Mainly the prefetching algorithms can be divided into two categories: correlated and non-correlated prefetching. Correlated prefetching associates each cache miss with some predefined target stored in a dedicated table [6, 16], while non-correlated ones predict the next prefetch line according to some simple predefined algorithms [11, 7, 14].

For all mentioned techniques, the ability to guess the next reference is not fully accurate and prefetching can result in cache pollution and unnecessary memory traffic. In this paper we propose a new memory hierarchy for single-path code that consists of a cache and a hardware prefetcher. The proposed design is able to prefetch sequential and non-sequential streams of instructions with full accuracy in the value and time domain. This constitutes an effective instruction prefetching scheme that increases the execution performace of single-path code and reduces both cache pollution and useless memory traffic.

The rest of the paper is organized as follows. Section 2 gives a short description of predicated instruction and presents some simple rules used to convert conventional code to single-path code. The new proposed memory hierarchy is presented in Section 3. Section 4 discusses related work. Finally, we make concluding remarks and present the future work in Section 5.

2. GENERATING SINGLE-PATH CODE

The goal of the single-path code-generation strategy is to eliminate the complexity of multi-path code analysis, by eliminating branch instructions from the control flow of the code. Different paths of program execution are the result of branch instructions which force the execution to follow different sequences of instructions. Branch instructions can be unconditional branches which always result in branching, or conditional branches where the decision for the execution direction depends on the evaluation of the branching condition.

The single-path conversion transforms conditional branches, i.e., those branches whose outcome is influenced by program inputs [12]. Before the actual single-path code conversion is done, a data-flow analysis [3] is run to identify the input-dependent instructions of the code. Branches which are not influenced by the input values are not affected by the transformation. After the data-flow analysis, the single-path conversion rules are applied and the new single-path code is generated. The only additional requirement for executing single-path converted code is that the hardware must support the execution of predicated instructions.

2.1 Predicated execution

Predicated instructions are instructions whose semantics are controlled by a predicate (or guard), where the predicate can be implemented by a specific predicate flag or register in the processor. Instructions whose predicate evaluate to "true" at runtime are executed normally, while those which evaluate to "false" are nullified to prevent that the processor state gets modified.

Predicated execution is used to remove all branching operations by merging all blocks into a single one with straight-line code [10]. For architectures that support predicated (guarded) execution the compiler converts conditional branches into (a) predicate-defining instructions and (b) sequences of predicated instructions – the instructions along the alternative paths of each branch are converted into sequences of predicated instructions with different predicates.

```
if(a)              beq    a,0,L1      pred_eq  p,a
  x=x+1            add    x,x,1       add      x,x,1 (p)
else               jump   L2          add      y,y,1 (not p)
  y=y+1        L1:
                   add    y,y,1
               L2:
```

Figure 1: if-conversion

Figure 1 shows an example of an *if-then-else* structure translated in assembler code with and without predicated instructions. In the first assembler code, depending on the outcome of the branch instruction, only part of the code will be executed, while in the second, single-path case all instruction will be executed but the state of the processor will be changed only for instructions with true predicated value.

2.2 Single-Path Conversion Rules

In the following we describe a set of rules to convert regular code into a single-path code [13]. Table 1 shows the single-path transformation rules for sequences, alternatives and loops structures. In this table we assume that conditions for alternatives and loops are simplified in boolean variables. The precondition for statement execution is represented with σ, while in cases of recursion the δ counter is used to generate unique variable name.

Simple Statement. If precondition for simple statement S is always true then the statement will be executed in every execution. Otherwise the execution of S will depend on the value of the precondition σ, which becomes the execution predicate. The same rule is used for statement sequences, by applying the rule sequentially to each part of the sequence.

Conditional Statement. For input-dependent (*ID(cond)* is *true*) branching structures, we serialize the S_1 and S_2 alternatives, where the precondition parameters of the alternatives $S_1$and S_2 are formed by a conjunction of the old precondition (σ) and the outcome of the branching condition that is stored in $guard_\delta$. If branching is not dependent on program inputs then the *if-then-else* structure is conserved and the set of rules for single-path conversion are applied individually to S_1 and S_2.

Loop. Input-data dependent loops are transformed in two steps. First, the original loop is transformed into a *for-loop* and the number of iterations N is assigned – the iteration count N of the new loop is set to the maximum number of iterations of the original loop code. The termination of the new new loop is controlled by a new counter variable ($count_\delta$) in order to force the loop to iterate always for the constant number N. Further, a variable end_δ is introduced. This variable is used to enforce that the transformed loop has the same semantics as the original one. The end_δ-flag stored in this variable is initialised to $true$ and assumes the value $false$ as soon as the termination condition of the original loop evaluates to $true$ for the first time. The value of end_δ-flag can also be changed to $false$ if a break is embedded into the loop. Thus S is executed under the same condition as in the original loop.

Table 1: Single-Path Transfromation Rules

Construct S		Translated Construct $\mathcal{SP}[\![S]\!]\sigma\delta$
S	if $\sigma = T$	S
	otherwise (σ)	S
$S_1 ; S_2$		$\mathcal{SP}[\![S_1]\!]\sigma\delta$; $\mathcal{SP}[\![S_2]\!]\sigma\delta$
`if` *cond* `then` S_1 `else` S_2	if *ID(cond)*	$guard_\delta$ `:=` *cond*; $\mathcal{SP}[\![S_1]\!]\langle\sigma \wedge \ guard_\delta\rangle\langle\delta+1\rangle$; $\mathcal{SP}[\![S_2]\!]\langle\sigma \wedge \neg guard_\delta\rangle\langle\delta+1\rangle$
	otherwise	`if` *cond* `then` $\mathcal{SP}[\![S_1]\!]\sigma\delta$ `else` $\mathcal{SP}[\![S_2]\!]\sigma\delta$

`while` *cond*
`max` N `times`
`do` S
	if *ID(cond)*	end_δ `:= false`
		`for` $count_\delta$ `:=` 1 `to` N `do begin`
		$\mathcal{SP}[\![$ `if` $\neg$*cond* `then` end_δ `:= true` $]\!]\sigma\langle\delta+1\rangle$;
		$\mathcal{SP}[\![$ `if` $\neg end_\delta$ `then` S $]\!]\sigma\langle\delta+1\rangle$
		`end`

	otherwise	`while` *cond* `do` $\mathcal{SP}[\![S]\!]\sigma\delta$

3. MEMORY HIERARCHY FOR SINGLE-PATH CODE

This section presents our novel architecture of the cache memory and the prefetcher used for single-path code.

3.1 Architecture of the Cache Memory

Caches are small and fast memories that are used to improve the performance between processors and main memories based on the principle of locality. The property of locality can be observed from the aspects of temporal and spatial behavior of the execution. Temporal locality means that the code that is executed at the moment is likely to be referenced again in the near future. This type of behavior is expected from program loops in which both data and instructions are reused. Spatial locality means that the instructions and data whose addresses are close by will tend to be referenced in temporal proximity because the instructions are mostly executed sequentially and related data are usually stored together [15].

As an application is executed over the time, the CPU makes references to the memory by sending the addresses. At each such step, the cache compares the address with tags from the cache. References (instructions or data) that are found in cache are called *hits*, while those that are not in the cache are called *misses*. Usually the processor stalls in case of cache misses until the instructions/data have been fetched from main memory.

Figure 2 shows an overview of the cache memory augmented with the single-path prefetcher. The cache has two banks, each consisting of *tag*, *data*, and *valid bit* (V) entries. Separation of the cache into two banks allows us to overlap the process of fetching (by the CPU) with prefetching (by the prefetch unit) and also cost less than dual-port cache of the same size. At any time, one of the banks is used to send instructions to the CPU and the other one to prefetch instructions from the main memory. Both, CPU and prefetcher can issue requests to the cache memory. Whenever a new value in program counter (PC) is generated the

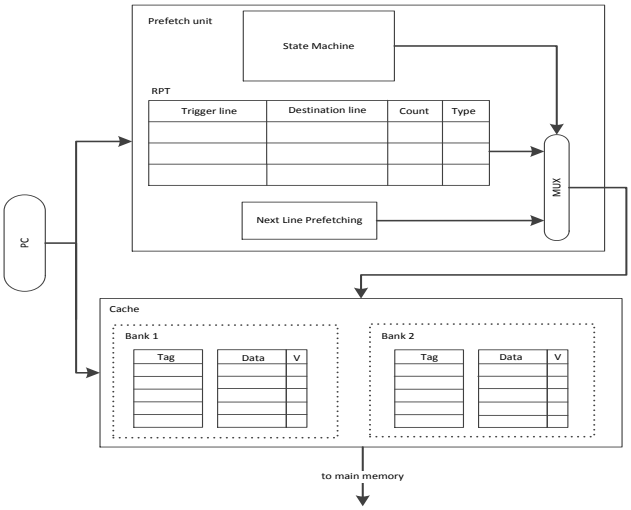

Figure 2: Prefetch-Cache architecture

value is sent to the cache and to the prefetcher. There are three different cases of cache accesses when the CPU issues an instruction request:

- No match within tag columns - the instruction is not in the cache. The cache stalls the processor and forwards the address request to the main memory;
- Tag match, V bit is zero - the instruction is not in the cache but the prefetcher has already sent the request for that cache line and the fetching is in progress. The cache stalls the processor and waits for the ongoing prefetching operation to be finished (V value to switch from zero to one).

- Tag match, V bit is one - the instruction is already in the cache (cache hit).

The V bit prevents the cache to replicate requests issued to main memory if the same request has already been issued by the prefetcher.

Instructions in the cache can be mapped to any location (fully associative), to a dedicated set of cache lines (set-associative) or to only one cache line location (direct-mapped). For single-path code, direct mapped is the most conventional cache solution. In single-path code, loop have sequential bodies and if the loop size is $cache_{size} < loop_{size} < 2 * cache_{size}$ then a minimal number of instructions will be evicted from the cache. To illustrate this, let us assume that a cache has a size of four cache lines $(1, 2, 3, 4)$ and the loop has a size equal to six cache lines (a, b, c, d, e, f). In each iteration the loop will be mapped in cache as: $(a, e) \rightarrow 1$, $(b, f) \rightarrow 2$, $(c) \rightarrow 3$, and $(d) \rightarrow 4$. In each iteration lines (c, d) are in the cache.

3.2 Prefetching Algorithm for Single-Path Code

The prefetching algorithm for single-path code considers two forms of prefetching: sequential and non-sequetial prefetching. Sequential instruction streams are a trivial pattern to predict since the address of the next prefetching is an increment of the current address line. A simple next-line prefetcher [14] is a suitable solution for such a pattern of instructions.

In contrast, a non-sequential prefetcher needs input information to determine the target address to be prefetched. Single-path code has a strong advantage in this part of the prefetching, because the outcome of every branch target is statically known. This target information can be given to the prefetcher in form of instructions (software prefetching) or it can be kept in a local memory (organized as a table) and used by the prefetcher when it is needed (hardware prefetching). For the software solution, special prefetch instructions are needed and the CPU hardware has to be modified.

In order to keep the development of the prefetcher independent form the CPU and the compiler, and also to avoid the overhead of executing fetch instructions in software, we have decided to use a hardware solution for the single-path prefetcher. Since the single-path code consists of serial segments and loops only, the subject of treatment from non-sequential algorithm are only the branch instructions of the loop back-edge. Loops larger than a cache size are easily handled by the prefetching algorithm, where the loop body is prefetched with the sequential algorithm while for the loop header with non-sequential one. If loops fully fit into the cache then they do not need to be prefetched on each iteration. Thus these loops are identified and the prefetcher does not generate any prefetching requests until the last iteration, when the execution stream exits the loop.

The granularity of prefetching is defined as one cache line. For larger amounts of prefetched instructions, the probability of overshoting the end of sequence would increase, thus resulting in cache pollution with useless prefetching. The granularity also determines that the prefetching distance is one cache line ahead.

3.3 Reference Prediction Table

The Reference Prediction Table (RPT) is the part of the prefetcher that holds information about the instruction stream (Figure 2). The RPT entries consist of *trigger address, destination address, count* and *type* column. *Trigger address* is the program counter address that triggers the non-sequential algorithm of the prefetcher. *Destination address* is the target address that is prefetched. Since loops in single-path code have a constant number of iterations the *counter* data is used to inform the prefetcher for how many times the target address should be prefetched. The *type* field indicates which loops fit into the cache and which loops are bigger than the cache. If the value of *type* is zero then the prefetcher will not take any action since the loop is smaller than cache and is completely in it. When the *counter* of that loop reaches zero, the loop iterations are finished, and the prefetcher triggers the prefetching of the next cache line.

A profiling process is needed to identify loops (loop header and back-edge branch of the loop), the number of iterations and the size of the loops in order to fill the RPT table.

3.4 Architecture of the Prefetcher

As shown in Figure 2 the prefetch hardware for single-path code consist of the Reference Prediction Table (RPT), the next-line prefetcher, and the prefetch controller (state machine). The next-line prefetcher serves for prefetching the sequential parts of the code, while the state machine in association with the RPT is used for pefetching targets in distance.

At run-time, when a new address is generated, its value is passed to the RPT table and the next-line prefetcher. In cases when the PC value matches an entry in the RPT table, the prefetch controller reads the *type* bit and the *counter* value to check if the loop is smaller/bigger than cache and on each interation if the final iteration has been reached. If there is no match with the RPT table entry, the next-line prefetching will increment the address for one cache line and issue that address to the cache. The RPT output has precedence over next-line prefetching.

4. RELATED WORK

Designers have proposed several strategies to increase the performance of cache-memory systems. Some of these approaches use software support to perform prefetching, while others are strictly hardware-based. Software solutions need explicit fetch instructions to be issued from the compiler to do prefetching. In this section we discuss only related hardware solutions.

The simplest form of prefetching comes passively from the cache itself. When a cache miss occurs, besides the missed instruction the cache fetches also instructions that belong to the same line into the cache. An extention of the cache line size implies a larger granularity – more instructions are fetched on a cache miss. The disadvantages are that longer cache line take longer to fill, generate useless memory traffic, and also they contribute to cache pollution [5].

The "one block look ahead" prefetching, later extended to "next-N-line", prefetches cache lines that are following the one currently being fetched by the CPU [14]. The scheme requires small additional hardware to compute the next sequential addresses. Unfortunately, "next-n-line" is unlikely to improve performance when execution proceeds through non-sequential execution paths. In this case the prefetching guess can be incorrect.

Tagged prefetching has a tag bit associated with each cache line [7]. When a line is prefetched, its tag bit is set

to zero. If the line is used then the bit is set to one and the next sequential line is immediately prefetched. The stream buffer is a similar approach except that the buffer stands between main memory and cache in order to avoid polluting the cache with data that may never be needed.

The target prefetching scheme addresses the problem of non sequential code [16]. This approach comprises a next-line prediction table consisting of two entries (current line address and target line address). When the program counter changes the value, the prefetcher searches the prediction table. If there is a match, then the target address is a candidate for prefetching.

The hybrid scheme that combines both next-line and target prefetching offers a cumulative accuracy for reducing cache misses. However, this solution has limited effectiveness since the predicted direction is defined from the previous execution. A similar approach is also used on "wrong-path" prefetching except that instead of target table this approach prefetches immediately the target of conditional branch after the branch instruction is recognized in in the decode stage [11]. This solution can be effective only for non-taken branches. The Markov prefetcher [6] prefetches multiple reference predictions from the memory subsystem, by observing the miss-reference stream as an Markov model.

Loop-based instruction prefetcing [2] is similar to our solution, except that the loop headers are always prefetched and the prefetching is issued at the end of the loop with no prefetch distance. The cooperative approach [9] also considers sequential and non-sequential prefetching by using a software solution for non-sequential prefetch. A dual-mode instruction prefetch scheme [8] is an alternative to improve worst-case execution time by associating a thread to each instruction block that is part of WCET. Threads are generated by the compiler and they are static during task execution.

5. CONCLUSION AND FUTURE WORK

To overcome the problem of long execution times of single-path code, we have proposed a new memory hierarchy organization that attempts to reduce the memory access time by bringing the instructions into the cache before they are required.

The single-path prefetching algorithm combines a sequential and a non-sequential prefetching scheme with the full accuracy in the predicted instruction stream based on the predictable properties of the single-path code. Designed as a hardware solution, the prefetcher does not produce an additional timing overhead for the instruction prefetching. Also, our solution allows the prefetcher functionality to be independent without interfering with any stage of CPU. The dual-bank cache makes it possible to pipeline the CPU and prefetcher accesses into the cache memory in order to fully utilize the memory bandwidth. By using a prefetch granularity of one cache line we eliminate the possibility for cache pollution and useless memory traffic.

In our future work we plan to show the feasibility of the memory hierarchy by implementing it in an FPGA platform and also to extend the prefetcher for input-independent if-else structures that are not converted to sequential code.

Acknowledgments

This work has been supported in part by the European Community's Seventh Framework Programme [FP7] under grant agreement 287702 (MultiPARTES) and the EU COST Action IC1202: Timing Analysis on Code Level (TACLe).

6. REFERENCES

[1] P. Axer, R. Ernst, H. Falk, A. Girault, D. Grund, N. Guan, B. Jonsson, P. Marwedel, J. Reineke, C. Rochange, et al. Building timing predictable embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4):82, 2014.

[2] Y. Ding and W. Zhang. Loop-based instruction prefetching to reduce the worst-case execution time. *Computers, IEEE Transactions on*, 59(6):855–864, 2010.

[3] J. Gustafsson, B. Lisper, R. Kirner, and P. Puschner. Code analysis for temporal predictability. *Real-Time Systems*, 32(3):253–277, 2006.

[4] S. Hahn, J. Reineke, and R. Wilhelm. Towards compositionality in execution time analysis-definition and challenges. In *6th International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, 2013.

[5] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012.

[6] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *ACM SIGARCH Computer Architecture News*, volume 25, pages 252–263. ACM, 1997.

[7] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on*, pages 364–373. IEEE, 1990.

[8] 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.*, pages 98–105. IEEE, 1993.

[9] C.-K. Luk and T. C. Mowry. Architectural and compiler support for effective instruction prefetching: a cooperative approach. *ACM Transactions on Computer Systems*, (1):71–109, 2001.

[10] J. C. Park and M. Schlansker. On predicated execution. Technical report, Technical Report HPL-91-58, HP Labs, 1991.

[11] J. Pierce and T. Mudge. Wrong-path instruction prefetching. In *Microarchitecture, 1996. MICRO-29. Proceedings of the 29th Annual IEEE/ACM International Symposium on*, pages 165–175. IEEE, 1996.

[12] 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*, pages 85–91. IEEE, 2002.

[13] P. Puschner, R. Kirner, B. Huber, and D. Prokesch. Compiling for time predictability. In *Computer Safety, Reliability, and Security*, pages 382–391. Springer, 2012.

[14] A. J. Smith. Sequential program prefetching in memory hierarchies. *Computer*, 11(12):7–21, 1978.

[15] A. J. Smith. Cache memories. *ACM Computing Surveys (CSUR)*, 14(3):473–530, 1982.

[16] J. E. Smith and W.-C. Hsu. Prefetching in supercomputer instruction caches. In *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 588–597. IEEE Computer Society Press, 1992.