WCET-Based Comparison of an Instruction Scratchpad and a Method Cache

Jack Whitham
Department of Computer Science
University of York
jack@cs.york.ac.uk

Martin Schoeberl
Department of Applied Mathematics
and Computer Science
Technical University of Denmark
masca@dtu.dk

***Abstract*—This paper compares two proposed alternatives to conventional instruction caches: a scratchpad memory (SPM) and a method cache. The comparison considers the true worst-case execution time (WCET) and the estimated WCET bound of programs using either an SPM or a method cache, using large numbers of randomly generated programs. For these programs, we find that a method cache is preferable to an SPM if the true WCET is used, because it leads to execution times that are no greater than those for SPM, and are often lower. However, we also find that analytical pessimism is a significant problem for a method cache. If WCET bounds are derived by analysis, the WCET bounds for an instruction SPM are often lower than the bounds for a method cache. This means that an SPM may be preferable in practical systems.**

I. Introduction

Recent trends towards *time-predictable computer architectures* [1], [2], [3] have led to the parallel development of two new types of local memory for storing instructions: a scratchpad memory (SPM) and a method cache (M$). Local instruction memory is nothing new, as instruction caches have been in use for decades, but caches present problems for *worst-case execution time* (WCET) analysis, which is necessary in a time-predictable architecture to ensure that all tasks will meet their deadlines.

While *direct-mapped* and *least-recently used* (LRU) set-associative instruction caches are certainly amenable to WCET analysis [4], [5], [6], some problems remain. For instance, manufacturers often prefer cache designs that are not so easy to analyze, such as those using *pseudo least-recently used* (PLRU) replacement policies, which are a known source of *timing anomalies* [7].

The n-way set-associative design allows an instruction with a particular physical memory address to be stored in n possible places within the cache [8]. The *average-case execution time* (ACET) of a program is greatly improved by set-associative LRU caches, as the set-associative design means that *conflict misses* between different pieces of code are less likely [9]. Conflict misses occur when useful information is evicted from the cache because it shares a cache line with other useful information. Set-associative caches are not entirely free of this effect, but it is much less likely to occur, and hence they are widely used [8]. However, the LRU mechanism is expensive in terms of hardware and energy, and so manufacturers often prefer a PLRU mechanism. With PLRU the ACET of a program is similar to true LRU. Unfortunately, WCET analysis of such PLRU caches is not straightforward and may be highly pessimistic [10].

This suggests that local instruction memory should be implemented in a way that is simple in terms of hardware (like PLRU), but can minimize ACET (like a set-associative design) *and* is also amenable to safe and precise WCET analysis. Two proposed alternatives are the *scratchpad memory* (SPM) and the *method cache* (M$). Both may be used as local memory, replacing a conventional direct-mapped or set-associative cache [11], [12]. Both may be used in a preemptive multitasking system [13], [14]. Both are amenable to WCET analysis [15], [16].

However, which of the two is truly preferable? There is no doubt that both can be implemented using simple hardware, as both SPM-based [17] and M$-based [14] systems have been implemented in FPGA hardware. We can also observe that both are free of conflict misses because the memory contents are not stored in cache lines. The question is therefore about the WCET: which alternative leads to a lower WCET?

We carry out a comparison of the M$ and the SPM that relies on synthetic programs expressed as call trees and a simple, but realistic, hardware model. We determine that lower *true* WCETs are achieved using an M$, but that lower WCET *bounds* are achieved using an SPM, as a result of the sort of WCET analysis that is necessary for M$. While an M$ may have an advantage over an SPM in reality, this advantage is not necessarily demonstrable by WCET analysis, and therefore an SPM may be preferable in practical systems.

The comparison of the SPM and the M$ is performed in the context of the T-CREST[1] project. T-CREST aims at building a time-predictable multicore system. T-CREST provides a timing-compositional VLIW processor, called Patmos [18], and a time-predictable memory controller [19]. A network-on-chip provides message-passing capabilities [20] for the multi-core system. For shared memory a memory tree network [21], with precise timing guarantees, is provided. An adapted LLVM compiler [22] supports all architectural features of Patmos. This work shall help to decide on which local memory solution shall be used in the Patmos processor. Both solutions, the SPM and the M$, are already implemented. A configuration decides which one (or both) is used.

[1] Time-predictable Multi-Core Architecture for Embedded Systems, see http://www.t-crest.org/

The paper is organized in 9 sections. Section II defines the SPM and the M$ architecture; Section III describes the program model to be used for the remainder of the paper, and Sections IV show how this model applies to the SPM and the M$ respectively. Section V describes experiments in which the true WCET of programs using an SPM and using an M$ are compared. Section VI repeats an experiment using WCET bounds determined by analysis. Section VII presents related work and Section VIII concludes.

II. Definitions

In this paper, a scratchpad memory (SPM) is a *local memory where content replacement is under program control*. That is, the information within an SPM is only changed upon the explicit direction of the program. The program specifies exactly what information is to be copied into the SPM; the programmer or compiler adds instructions to the program to make this happen. Therefore, the exact state of the local memory is known at all points throughout the program, and WCET analysis can make use of this information.

In contrast, a cache is *local memory where contents are not precisely determined until execution*. WCET analysis, which is performed before execution, must make use of incomplete or abstract information about the cache state [4]. Therefore, it may be necessary to make pessimistic assumptions about the cache state, for example if it is not certain if a particular instruction is present.

A method cache (M$) is organized to cache full methods (or functions) [12]. It was originally developed for a Java processor [14]. Therefore, it is called method cache, but is also applicable for functions and procedures.

A method may be loaded into the cache on a call instruction or when returning from a method. On a call the called method is checked whether it is in the cache. If it is a miss, the method is loaded into the cache. On a return from a method the caller is checked and on a miss loaded. The advantage is that all other instructions are guaranteed hits and can be ignored by the cache analysis. Only the call tree needs to be considered to analyze cache hits and misses.

The basic organization of an M$ is a local memory that is divided int blocks. A loaded method can span multiple blocks, but the method needs to be loaded into contiguous blocks. This organization allows two replacement policies: (1) first-in-first-out (FIFO), which works like a ring buffer, and (2) a stack oriented replacement where allocation of blocks follows the same regime as the allocation of stack frames. However, the stack oriented replacement leads to conflicts of methods at the same hight in the call tree. This conflict results in continuous replacement of methods when called in a loop.

For a block oriented M$, a tag entry is associated for each block. The entry for the first block of the method contains the tag entry (address of the method in the memory). The other blocks are cleared on a method load. With a FIFO replacement, this mechanism automatically removes the tag entry of a method when the first block of a method is overwritten by a newly loaded method.

Another variant of the M$ is to reduce the block size to single instruction words and have a tag memory with one entry per method. The dynamic instruction scratchpad (D-ISP) is organized in this way [23], [24]. With this variant, the tag memory also contains the length of the cached method. The number of tag entries limits the number of methods that can be in the cache at the same time.

The cache memory is better used, as there is only a single area in the memory that might not be used due to fragmentation. On the block oriented M$, each loaded method has one block that might have unused space due to fragmentation.

The implementation of the M$ in Patmos [25] uses the organization with tag entries for individual methods and allocation granularity of single instruction words. However, this paper considers the *original* M$ organization in individual blocks as the SPM is also organized with block granularity.

III. Program Model

The program model used in this paper is a *call tree* in which each node represents a *method* with an associated size, and each edge represents a call/return relationship between two methods. There is a root method representing the entry point.

For instance, in a typical C program, we may have an entry point named `main()` which calls `printf()`. This gives a call tree with two nodes (`main`, `printf`), each labelled with a size, and linked by an edge. The edge will be labelled with a call frequency, which may be constant, or related in some way to the program input. Edges are *unordered*; the call tree gives no indication of the order in which a node's children are called.

Methods are *virtually inlined* [4] so that the identity of each method is dependent on its path to the root method. If two methods `foo()` and `bar()` both call `baz()`, there will be two copies of `baz` in the tree: one representing the `foo-baz` path, the other representing the `bar-baz` path. However, the two copies of `baz` will share the same space in local memory.

We use a call tree because it is the program representation used for an M$ [12] and one of the representations that may be used for an SPM [26].

While this program model might look simplistic and would not be adequate for comparison with a standard instruction cache, it fits for our comparison. The SPM and the M$ exchange local memory content with new content from the main memory at method granularity. Therefore, we consider modeling a program just as a collection of interconnected methods as a valid abstraction for our comparison.

IV. Execution Models and WCET Analyses

This Section describes the execution models and the resulting WCET analyses for the SPM and the M$.

A. Execution Model for the SPM

A program (modeled as a call tree) can be mapped to an SPM by grouping methods into sub-units named *regions* [15], [26]. Each region needs to be small enough to be stored in the SPM. Formally, regions are elements in a *set partition* of the call tree [27]. The regions are disjoint subsets of the nodes in the call tree. Each region has a size determined as the sum of the sizes of the methods within it; the maximum size is the SPM size k.

Partitioning is equivalent to deciding which edges should represent region transitions. An edge is a region transition if the call/return operation represented by that edge causes the contents of the SPM to change. For instance, if `main()` calls `atoi()`, but there is not enough SPM space for both `main` and `atoi`, the `main-atoi` edge must be a region transition.

There are up to 2^n possible partitions for a call tree containing n edges, though some of these may violate the maximum size constraint. The program would usually be mapped to the SPM with the intention of minimizing either the ACET or the WCET, the latter being more useful for a time-predictable system.

Call tree partitioning can be performed by exhaustive search, by hill climbing [15], by a branch-and-bound algorithm [28], or by a polynomial-time optimal algorithm such as ELA-1 [26]. ELA-1 is a k-partitioning algorithm for call trees. It determines which call/return edges should be region boundaries in order to minimize the total cost of copying information from external memory to the SPM. The upper bound k on the available SPM space is respected.

B. WCET Analysis for the SPM

WCET analysis can be performed on a call tree as soon as the region transitions are known. The WCET of the program may be accurately represented as an *integer linear program* (ILP) based on the call tree [29]. In this ILP the execution frequency of each edge is represented as a variable. The WCET is the sum of the WCET of each method multiplied by the execution frequency of its input edge, *plus* the costs of taking that edge: zero for edges that are not region transitions, and the loading cost otherwise. This is an accurate form of WCET analysis, because there is no pessimism about the loading costs. If the partitioned call tree requires a region transition, then the loading cost is precisely accounted by the ILP. Therefore, the WCET bound matches the true WCET.

As we are interested in the comparison of the cost for the SPM versus the M$, we simplify the WCET analysis by ignoring the WCET of the individual methods, i.e., they are set to 0. The resulting WCET contains only method load cost and will overemphasize the difference between an SPM and an M$.

C. Execution Model for the M$

A program (modeled as a call tree) is mapped to the M$ dynamically during execution. The cache state is only updated when the program calls a method, or when a method returns. Each call (and return) triggers a search of the cache, usually carried out in hardware, which determines if the target method is present. If it is not, then a cache miss occurs, and the method is loaded from external memory.

The allocation units of M$ are entire methods. Because the size of method vary, these cannot be allocated to fixed-size cache blocks, and are instead stored within a ring buffer, so newly added methods overwrite older methods. There is a separate index buffer that allows the cache to be searched quickly for the location of a particular method. These memories are managed by a *first-in first-out* (FIFO) policy, in which the order of eviction from cache is the same as the order of addition to the cache. This is necessary in order to prevent memory fragmentation. The M$ has been implemented as part of the JOP CPU [14], as part of the SHAP CPU [30], as part of Patmos [25], and independently by Metzlaff et al. [31]. The M$ hardware is more complex than the SPM hardware, as the search functionality is usually implemented in hardware. However, for a resource constraint system this search could also be implemented in software.

D. WCET Analysis for the M$

The WCET analysis included with JOP operates by generating an ILP model of the control flow graph (CFG) [16]. Cache load times (miss costs) are added as additional blocks to invoke and return instructions.

The static M$ analysis is performed on the call tree. It is static and scope based. Scopes are built bottom-up in the call tree, i.e., from leaves towards the root. If the memory consumption of the methods contained in a scope is less than the M$ size than those methods are marked as *single miss* methods. All other methods (further up in the call tree) are marked as always miss. The scopes are increased to their maximum size while still containing methods that fit into the M$.

This single miss information is then used with loop information to incorporate the M$ load cost into the overall WCET. As an example assume a method `m1` that contains a loop, which invokes `m2` and `m3`. All three methods need to fit into the cache to mark them single miss. The single miss cost is accounted for all three methods, even when methods `m2` and `m3` are invoked multiple times. We know that they can miss at most once.

Due to the FIFO replacement a miss does not need to happen on the first invocation, it can already be in the cache, but later replaced and on a further iteration loaded for the first time. The same can happen for the outer method `m1`: it is already in the cache when `m2` or `m3` are invoked, but `m1` could be evicted by an invocation of `m2` or `m3`. In that case `m1` is loaded into the cache again on a return from `m2` or `m3`. Therefore, the only knowledge we have is that each method is loaded at most once within this loop scope, but we don't know in which iteration.

This analysis may be pessimistic, particularly if all the methods in the subtree are actually executed together. For instance, some methods may be mutually exclusive, or executed only once. However, comparison with exhaustive search, based on model checking, has shown that this introduced pessimism is in the range of 2% to 7% for small, but real-world, embedded programs [16].

V. Experiments

Our experiments make use of synthetic programs that are generated at random. They consist of methods and call/return edges. Each method has a size and may call one or more other methods. Each call/return edge has a frequency, which may be constant or an input-dependent variable.

On purpose we decided to not use common WCET benchmarks, such as the well-established MRTC benchmark suit.[2]

[2]see http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

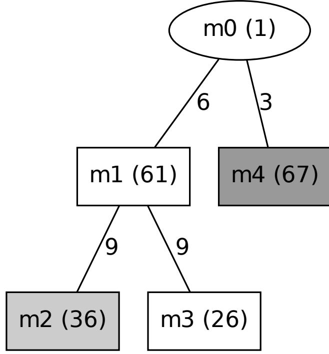

Fig. 1. An example of a generated program containing five methods (m0..m4). Each method has a size (e.g. m1 has size 61). Each call/return edge has a frequency: m0-m4 has frequency 3, so m4 is called 3 times in the worst case by the root method m0.

First, these benchmarks are small (the active instruction memory footprint is the range of 1–3 KB for most benchmarks). Second, the call tree is mostly small and shallow (many benchmarks just call leave functions from `main`). Third, and most important, with synthetic benchmarks we can generate a very large amount of test cases – in this paper 100000 test samples.

Figure 1 gives an example of a generated program. The experiment software generates call trees containing between 5 and 20 methods; each number of methods is equally likely, and the choice is made using a pseudorandom number generator. A root method is generated, and then for each subsequent method, a parent is chosen from amongst the existing methods. The size of each method is chosen from 1 to 100 blocks with the local memory size fixed at 100 blocks. The call count for each method is generated from 1 to 10.

A. True WCET versus WCET Bound

A program has a true WCET: this is the actual worst-case execution time that can ever occur for that program. In some simple cases, such as single-path programs [32], this true WCET is easily determined. In others, it is only feasible to find an estimated *bound* on the true WCET, typically an overestimation [33]. Overestimates may occur because of the complexity of the hardware, which leads to pessimism in the hardware model [7], and because of the complexity of software, where behavior may not be perfectly represented by loop bounds and other relationships within the WCET model.

It is important to distinguish between the true WCET and the WCET bound, so we introduce the term TC to represent a true WCET, and WB to represent a WCET bound. The WCET bound of a program i executed with an SPM is WB_i^{spm}, while the true WCET is TC_i^{spm}. We assume that WCET bounds are safe, so $TC_i^{\text{spm}} \leq WB_i^{\text{spm}}$. The TC is of theoretical interest when comparing SPM and M\$, but the WB is of greater practical value, because the TC cannot always be determined. Note that WB is determined by the hardware *and* the WCET tool capabilities. A smarter WCET tool can deliver a tighter WB than a simpler (and maybe faster) WCET tool.

In the following sections we compare TC^{spm} and TC^{mc} by ensuring that programs are sufficiently simple that the true WCET can be determined.

B. Program Model Assumptions

We assume that there is no time or space overhead to initiate a call or return, other than that introduced by copying instructions from external memory. With realistic hardware implementations, the copying process proceeds a block at a time [17]. Each block copy moves a fixed amount of information (usually a small power of two, e.g. 16 bytes) and takes a constant length of time (e.g. 400 ns).

We assume that methods have zero WCET and the whole cost of running the program is therefore the cost of loading it from external memory. This assumption over emphasis the cost difference between an SPM and an M\$. A WCET value that would include instruction cost as well would diminish the difference. However, for a comparison of the SPM and the M\$ we are interested in this isolated cost.

Furthermore, this isolation simplifies our experimental setup. This setup allows us to treat the WCET as a copy count, independent of any time unit, and method sizes as a number of blocks, independent of any size unit.

Let TC_i^{spm} be the WCET of a program i when executed with an SPM, and let TC_i^{mc} be the WCET of the same program when executed with an M\$. Figure 2 shows the difference between TC^{spm} and TC^{mc} for 100000 synthesized programs. For each program we computed the ratio

$$\frac{TC^{\text{spm}}}{TC^{\text{mc}}} \tag{1}$$

C. Single-Path Programs

Single-path programs are a special case in which TC and WB are easily determined, as the execution time of the program is constant [32]. We synthesize single-path programs by ensuring that all call/return edges have constant frequencies (like Figure 1).

Having generated a program, we produce an SPM allocation that minimizes the cost of inter-region transitions. Because the programs are small, this may be done using exhaustive search, however, in order to save time, we use the optimal ELA-1 algorithm described in [26]. In Figure 1, shading is used to indicate region membership. In this example `m0`, `m1`, and `m3` are in one region, and `m2` and `m4` are placed in different regions. Therefore, the edges `m0-m4` and `m1-m2` are region transition edges. The path `m1-m2` incurs a loading cost of 36 blocks on call, and a loading cost of $1+61+26=88$ blocks on return (the combined size of `m0`, `m1`, and `m3`).

We "execute" each program using a simulator. The simulator begins at the root method and "executes" its children recursively. The children are "executed" in an arbitrary order, with each being "executed" the number of times given by its edge label.

The first simulation run uses a model of the SPM. The number of copies from external memory is TC^{spm}. The second

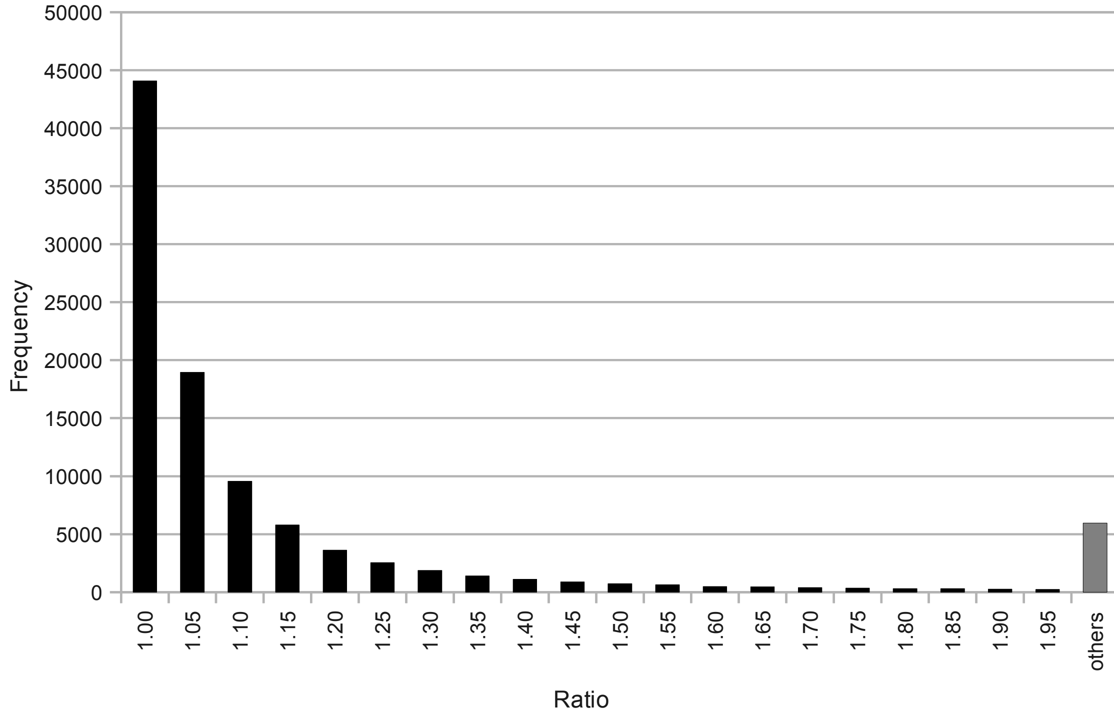

Fig. 2. Histogram of the WCET ratio between SPM and M$ (Equation 2: $TC^{\text{spm}}/TC^{\text{mc}}$) for 100000 synthesized single-path programs. The leftmost column represents all values from 1.00 to 1.05, the second 1.05 to 1.10, and so on. The "others" column represents all values greater than 2.0.

simulation run uses a model of the M$. The number of copies from external memory is TC^{mc}. For each program execution we computed the ratio

$$\frac{TC^{\text{spm}}}{TC^{\text{mc}}} \tag{2}$$

and plotted its value on the histogram shown in Figure 2. Larger WCETs are of course undesirable, so our results here show us that M$ is either preferable to (or the same as) SPM. Though it is common for TC^{spm} to be within a few percent of TC^{mc}, a significant number of programs have a substantially larger TC^{spm}.

We found that $TC^{\text{spm}} \geq TC^{\text{mc}}$, i.e., there is no situation in which SPM is preferable to M$, at least for the program model considered in this paper. Consider Figure 1 once more: `m1`, `m2`, and `m3` cannot coexist in a single region, because it would be too large. But because `m1` repeatedly calls `m2`, and then repeatedly calls `m3`, it is best if a region is first formed containing `m1` and `m2`, and then a second region is formed containing `m1` and `m3`. This is not possible with the SPM program model, but it is possible with the M$.

D. Discussion

The single-path programs do not include any situation where the M$ loads more information than the SPM. Therefore, the M$ is preferable in this case. With the M$, region boundaries change dynamically as the program is executed, and this results in the same number of region transitions *or fewer* in relation to the SPM. It is better to allow the region assignment to adapt to the behavior of the program.

However, in general we see that for very many cases (45 %) the M$ and the SPM behave almost identical. And for 99.5 % of the test cases the difference is below a factor of 2. This factor is the method load time only. When the WCET of the individual methods is included, this factor will be reduced.

E. Multi-Path Programs

A multi-path program is one in which the execution path depends on the input. Most programs are multi-path. Our synthesized programs become multi-path if one or more edge weights are variable rather than constant.

The value of a variable edge weight may have an impact on TC, but if the number of possible combinations of values is small, we can test all possible combinations in order to determine TC.

Suppose that exactly one edge is picked from each synthetic program, and its value n is taken as an upper bound rather than a constant. The program is then "executed" with SPM and with M$ whilst the value of that edge is assigned each value $x \in [0,n]$ inclusive. One value of x will maximize the execution time in each case. This gives TC^{spm} (for SPM) and TC^{mc} (for M$).

We executed the benchmarks with the variations on one edge for all 100000 test programs. The histogram for multi-path programs turns out to be indistinguishable from Figure 2, and therefore omitted from the paper. However, this is an indication that the previous comparison generalizes for multi-path programs.

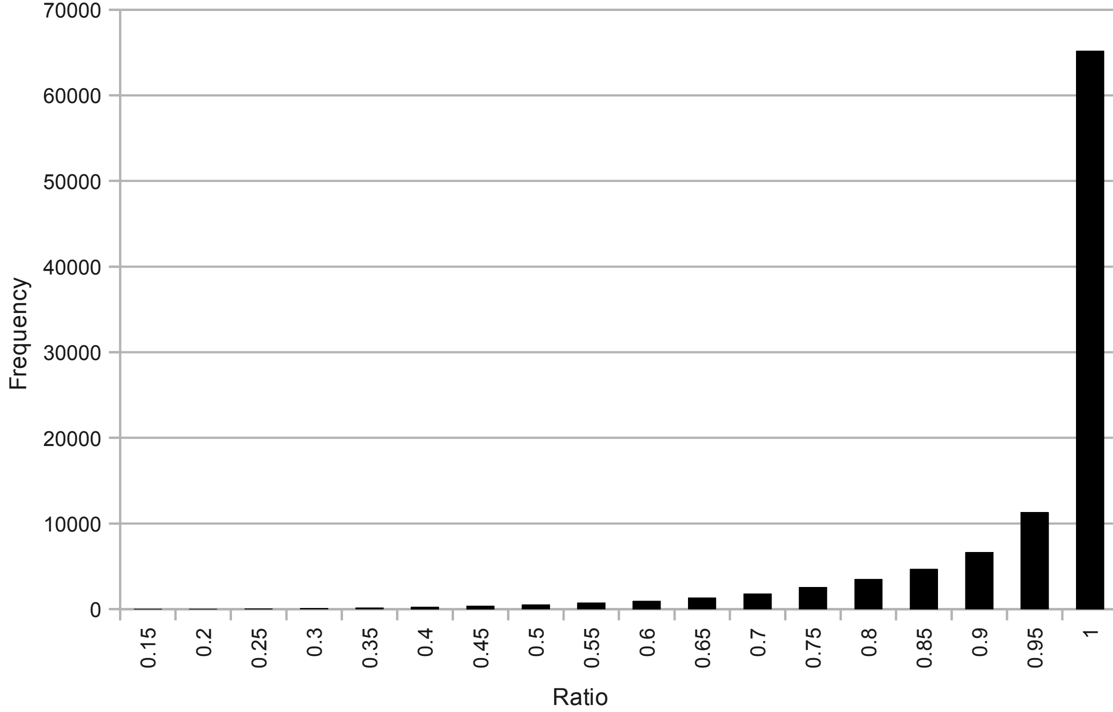

Fig. 3. Histogram of the value Histogram of the WCET bound ratio between SPM and M$ (Equation 4: $WB^{\text{spm}}/WB^{\text{mc}}$) for 100000 multi-path programs. The first column contains all values from 0.10 to 0.15, the second 0.15 to 0.20, and so on up to 0.95 to 1.00.

F. Summary

Our experiments tell us that the true WCET TC^{spm} of a program executed with an SPM is greater than or equal to the true WCET TC^{mc} of the program executed with an M$, i.e.:

$$\forall i. TC_i^{\text{spm}} \geq TC_i^{\text{mc}} \tag{3}$$

As an M$ is barely more complicated than an SPM, and as it requires no special algorithm to partition the call tree at compile time, it would seem that an M$ is preferable to an SPM.

VI. The Problem of WCET Bounds

Having determined that the true WCET of a program is often improved by using an M$ instead of an SPM, we may ask why it is worth considering an SPM at all. The problem is that the true WCET, the TC, is only useful if it is known, and in general, it is not. For most programs, we will only have WB, the WCET bound. Unfortunately, Equation 3 does not also apply to WB.

State-of-the-art WCET analysis for the M$ assigns a *single miss* classification to edges that are within a subtree where all methods fit in the cache (see Section IV-D). Other edges are considered to be misses, because it is not possible to prove that they are anything else. This is an unfortunate consequence of the FIFO policy, which evicts elements from cache in the order they were added, rather than the order that they were used. In turn, the FIFO policy is necessary in order to avoid memory fragmentation within the cache space.

Figure 3 shows the results of a comparison using WCET bounds determined by analysis for the synthesized programs. For each program we computed the ratio

$$\frac{WB^{\text{spm}}}{WB^{\text{mc}}} \tag{4}$$

and as Figure 3 shows, the advantage of M$ that exists in reality is now invisible, having been lost to analytical *pessimism*. Pessimism is the difference $WB^{\text{mc}} - TC^{\text{mc}}$: the degree of overestimation involved in WB. SPM WCET analysis has no pessimism at all, i.e., $WB^{\text{spm}} = TC^{\text{spm}}$. But the M$ WCET analysis may involve pessimism for any program that is larger than the local memory size.

Our further experiment tells us that while the true WCET TC_i^{spm} may be greater than TC_i^{mc} for a program i, the WCET bound WB_i^{spm} is less than or equal to WB_i^{mc} for the same program, i.e.:

$$\forall i. WB_i^{\text{spm}} \leq WB_i^{\text{mc}} \tag{5}$$

Though an M$ does have an advantage as far as the true WCET is concerned, this advantage is lost because of pessimism of real WCET analysis.

The current comparison uses optimization for the SPM allocation, but leaves the methods as they are for the M$. As a compiler might inline methods those methods can become too large for the M$ (and SPM). Therefore, the Patmos compiler [22] contains a so-called *function splitter*. In future work we might adapt the function splitter to perform optimization of the method inlining and splitting to optimize for the M$ structure. Furthermore, there is space for improvement of the WCET analysis for the M$. A scope-based persistence analysis for the method cache has been developed.[3]

[3]The paper “Scope-based Instruction Cache Analysis” is under submission.

VII. Related Work

This paper builds on existing work on M$ and SPM for use in time-predictable architectures [11], [12]. M$ and SPM are both intended to solve the same problem, namely finding an implementation of local memory that is simple in terms of hardware, but is also effective in minimizing average-case execution time (ACET) and worst-case execution time (WCET). Both M$ and SPM are amenable to WCET analysis [15], [16].

However, the two have not previously been brought together and compared against each other, though earlier comparisons between cache and SPM have been carried out [15], [34], indicating that the average-case execution time with cache and SPM are similar, assuming an effective allocation of SPM space.

The comparison is particularly important because of the growing interest in time-predictable memory architectures. There are now several implementations of M$ [30], [31], [25], [14], WCET analysis of caches is an important topic [33], and research into SPM allocation algorithms and implementations continues [13], [26].

Metzlaff and Ungerer compared the WCET bounds for different instruction cache architectures [35]. They compare the I-SPM (the method cache) with a static SPM and a standard instruction cache. In contrast to our findings their M$ outperforms the static SPM. Their M$ also outperforms a standard instruction cache.

An average case comparison of the M$ against a standard, direct mapped instruction cache has been performed in the original M$ paper [12]. In that comparison the M$ can even outperform an instruction cache in the average case when the memory bandwidth is high and there is considerable latency before a memory burst, as it is in current SDRAM memory organizations.

Organizing the SPM content under program control implies a more complicated programming model, which reminds us to overlay techniques from the 80's. However, the SPM can be automatically partitioned [36], [37], [38]. A similar approach for time-predictable caching is to lock cache blocks. The control of the cache locking [39] and the allocation of data in the scratchpad memory [40], [41], [42] can be optimized for the WCET. A comparison between locked cache blocks and a scratchpad memory with respect to the WCET can be found in [15]. While former approaches rely on the compiler to allocate the data or instructions in the scratchpad memory an algorithm for runtime allocation is proposed in [43].

VIII. Conclusion

We compared two approaches to time-predictable local memories for instructions: a scratchpad memory (SPM) and a method cache (M$). Tests using large numbers of synthetic programs have been used to study the differences in the true WCET (TC) and the WCET bound (WB).

The tests indicate that lower WCETs may be achieved using an M$. It is able to make good use of the local memory because the usage of local memory is adjusted dynamically as the program runs. Because an SPM is not able to take advantage of this, and uses the local memory according to a predetermined plan, it does not use the local memory as well. The result is (often) a greater true WCET, i.e. $TC_i^{\text{spm}} \geq TC_i^{\text{mc}}$ for a program i.

However, further tests indicate that an SPM *appears* to lead to a lower WCET if WCET analysis is used. Because of the nature of the FIFO replacement policy used by the M$, safe WCET estimates are often pessimistic, and the result of this pessimism is that the WCET bound is *better* (or no worse) for an SPM than for an M$, i.e. $WB_i^{\text{spm}} \leq WB_i^{\text{mc}}$ for a program i.

We therefore conclude that an SPM may be preferable in practical systems unless and until WCET analysis for the M$ can be improved. This could also involve some change to the FIFO mechanism used by the M$. A hybrid architecture of an SPM and a M$ may be worth considering as a way to achieve the advantages of the M$ without losing the precise WCET analysis that is also needed.

Acknowledgment

This work was partially funded under the European Union's 7th Framework Programme under grant agreement no. 288008: Time-predictable Multi-Core Architecture for Embedded Systems (T-CREST).

References

[1] M. Schoeberl, "Time-predictable computer architecture," *EURASIP Journal on Embedded Systems*, vol. vol. 2009, Article ID 758480, p. 17 pages, 2009.

[2] J. Whitham and N. Audsley, "Time-Predictable Out-of-Order Execution for Hard Real-Time Systems," *IEEE Transactions on Computers*, vol. 59, no. 9, pp. 1210–1223, 2010.

[3] I. Liu, J. Reineke, D. Broman, M. Zimmer, and E. A. Lee, "A PRET microarchitecture implementation with repeatable timing and competitive performance," in *Proceedings of IEEE International Conference on Computer Design (ICCD 2012)*, October 2012.

[4] F. Mueller, "Timing analysis for instruction caches," *Real-Time Syst.*, vol. 18, no. 2-3, pp. 217–247, 2000.

[5] C. Ferdinand and R. Wilhelm, "Efficient and precise cache behavior prediction for real-time systems," *Real-Time Systems*, vol. 17, no. 2-3, pp. 131–181, 1999.

[6] J. Reineke, D. Grund, C. Berg, and R. Wilhelm, "Timing predictability of cache replacement policies," *Journal of Real-Time Systems*, vol. 37, no. 2, pp. 99–122, Nov. 2007.

[7] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm, "The influence of processor architecture on the design and results of WCET tools," *Proceedings of the IEEE*, vol. 91, no. 7, pp. 1038–1054, Jul. 2003.

[8] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach, 4th ed.* Morgan Kaufmann Publishers, 2006.

[9] M. D. Hill and A. J. Smith, "Evaluating associativity in cpu caches," *IEEE Trans. Comput.*, vol. 38, no. 12, pp. 1612–1630, Dec. 1989.

[10] J. Reineke, D. Grund, C. Berg, and R. Wilhelm, "Timing predictability of cache replacement policies," *Real-Time Syst.*, vol. 37, no. 2, pp. 99–122, 2007.

[11] J. Whitham and N. Audsley, "Implementing Time-Predictable Load and Store Operations," in *Proc. EMSOFT*, 2009, pp. 265–274.

[12] M. 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)*, ser. LNCS, vol. 3292. Agia Napa, Cyprus: Springer, October 2004, pp. 371–382.

[13] J. Whitham and N. Audsley, "Explicit Reservation of Local Memory in a Predictable, Preemptive Multitasking Real-time System," in *Proc. RTAS*, 2012.

[14] M. Schoeberl, "A Java processor architecture for embedded real-time systems," *Journal of Systems Architecture*, vol. 54/1–2, pp. 265–286, 2008.

[15] I. Puaut and C. Pais, "Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison," in *Proceedings of the conference on Design, Automation and Test in Europe (DATE 2007)*. San Jose, CA, USA: EDA Consortium, 2007, pp. 1484–1489.

[16] M. Schoeberl, W. Puffitsch, R. U. Pedersen, and B. Huber, "Worst-case execution time analysis for a Java processor," *Software: Practice and Experience*, vol. 40/6, pp. 507–542, 2010.

[17] J. Whitham and N. Audsley, "The Scratchpad Memory Management Unit for Microblaze: Implementation, Testing, and Case Study," University of York, Tech. Rep. YCS-2009-439, 2009.

[18] 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.

[19] 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.

[20] M. Schoeberl, F. Brandner, J. Sparsø, and E. Kasapaki, "A statically scheduled time-division-multiplexed network-on-chip for real-time systems," in *Proceedings of the 6th International Symposium on Networks-on-Chip (NOCS)*. Lyngby, Denmark: IEEE, May 2012, pp. 152–160.

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

[22] P. Puschner, D. Prokesch, B. Huber, J. Knoop, S. Hepp, and G. 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)*, 2013, pp. 33–40.

[23] 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 (MEDEA 2008)*. New York, NY, USA: ACM, 2008, pp. 38–45.

[24] S. Metzlaff, I. Guliashvili, S. Uhrig, and T. Ungerer, "A dynamic instruction scratchpad memory for embedded processors managed by hardware," in *Architecture of Computing Systems - ARCS 2011*, ser. Lecture Notes in Computer Science, M. Berekovic, W. Fornaciari, U. Brinkschulte, and C. Silvano, Eds., vol. 6566. Springer Berlin / Heidelberg, 2011, pp. 122–134.

[25] P. Degasperi, S. Hepp, W. Puffitsch, and M. Schoeberl, "A method cache for Patmos," in *Proceedings of the 17th IEEE Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC 2014)*. Reno, Nevada, USA: IEEE, June 2014.

[26] J. Whitham and N. Audsley, "Optimal Program Partitioning for Predictable Performance," in *Proc. ECRTS*, 2012.

[27] E. W. Weisstein, "Set Partition," http://mathworld.wolfram.com/SetPartition.html, 2012.

[28] R. Leupers and P. Marwedel, "Function inlining under code size constraints for embedded processors," in *Proc. ICCAD*, 1999, pp. 253–256.

[29] P. Puschner and A. Schedl, "Computing maximum task execution times - a graph-based approach," *Real-Time Syst.*, vol. 13, no. 1, pp. 67–91, 1997.

[30] T. B. Preusser, M. Zabel, and R. G. Spallek, "Bump-pointer method caching for embedded java processors," in *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems (JTRES 2007)*. New York, NY, USA: ACM, 2007, pp. 206–210.

[31] S. Metzlaff, I. Guliashvili, S. Uhrig, and T. Ungerer, "A dynamic instruction scratchpad memory for embedded processors managed by hardware," in *Proc. ARCS*, 2011, pp. 122–134.

[32] P. Puschner, "Experiments with WCET-oriented programming and the single-path architecture," in *Workshop on Object-Oriented Real-Time Dependable Systems*, Feb. 2005.

[33] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution time problem – overview of methods and survey of tools," *Trans. on Embedded Computing Sys.*, vol. 7, no. 3, pp. 1–53, 2008.

[34] J. Whitham and N. Audsley, "Investigating average versus worst-case timing behavior of data caches and data scratchpads," in *Proc. ECRTS*, ser. ECRTS '10, 2010, pp. 165–174.

[35] S. Metzlaff and T. Ungerer, "A comparison of instruction memories from the WCET perspective," *Journal of Systems Architecture*, no. 0, pp. –, 2013.

[36] O. Avissar, R. Barua, and D. Stewart, "An optimal memory allocation scheme for scratch-pad-based embedded systems," *Trans. on Embedded Computing Sys.*, vol. 1, no. 1, pp. 6–26, 2002.

[37] F. Angiolini, L. Benini, and A. Caprara, "Polynomial-time algorithm for on-chip scratchpad memory partitioning," in *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES-03)*. New York: ACM Press, Oct. 30 Nov. 01 2003, pp. 318–326.

[38] M. Verma and P. Marwedel, "Overlay techniques for scratchpad memories in low power embedded processors," *IEEE Trans. VLSI Syst*, vol. 14, no. 8, pp. 802–815, 2006.

[39] I. Puaut, "WCET-centric software-controlled instruction caches for hard real-time systems," in *ECRTS '06: Proceedings of the 18th Euromicro Conference on Real-Time Systems*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 217–226.

[40] L. Wehmeyer and P. Marwedel, "Influence of memory hierarchies on predictability for time constrained embedded software," in *Proceedings of Design, Automation and Test in Europe (DATE2005).*, March 2005, pp. 600–605 Vol. 1.

[41] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen, "WCET centric data allocation to scratchpad memory," in *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS)*. IEEE Computer Society, 2005, pp. 223–232.

[42] J.-F. Deverge and I. Puaut, "Wcet-directed dynamic scratchpad memory allocation of data," in *ECRTS '07: Proceedings of the 19th Euromicro Conference on Real-Time Systems*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 179–190.

[43] R. McIlroy, P. Dickman, and J. Sventek, "Efficient dynamic heap allocation of scratch-pad memory," in *ISMM '08: Proceedings of the 7th international symposium on Memory management*. New York, NY, USA: ACM, 2008, pp. 31–40.