A Stack Cache for Real-Time Systems

Martin Schoeberl and Carsten Nielsen
Department of Applied Mathematics and Computer Science
Technical University of Denmark
Email: masca@imm.dtu.dk, carstenlau.nielsen@uzh.ch

***Abstract*—Real-time systems need time-predictable computing platforms to allow for static analysis of the worst-case execution time. Caches are important for good performance, but data caches are hard to analyze for the worst-case execution time.**

Stack allocated data has different properties related to locality, lifetime, and static analyzability of access addresses compared to static or heap allocated data. Therefore, caching of stack allocated data benefits from having its own cache.

In this paper we present a cache architecture optimized for stack allocated data. This cache is additional to the normal data cache. As stack allocated data has a high locality, even a small stack cache gives a high hit rate. A stack cache added to a write-through data cache considerably improves the performance, while a stack cache compared to the harder to analyze write-back cache has about the same average case performance.

I. Introduction

Data caches are hard to analyze for the worst-case execution time (WCET). The main issue with data cache analysis is the static prediction of addresses for load and store instructions. A strong, and therefore expensive, value analysis is needed to find some addresses statically. While stack addresses and addresses of simple data types allocated statically are relatively easy to compute, data structures with pointers are harder to analyze, and addresses of data structures that are dynamically allocated on the heap are impossible to analyze.

One improvement of this situation is splitting the data cache for different data areas [20]. This splitting into several caches simplifies WCET analysis, as those caches can be analyzed independently. A first candidate is a cache for stack allocated data. In the context of Patmos [21], we have implemented a stack cache [1] with supporting instructions and compiler support.

This paper presents a cache for stack allocated data that simplifies WCET analysis for loads from and stores to the stack area. The presented stack cache needs no instruction set changes and no compiler support. It can be added to any standard microprocessor.

Stack data has a high spatial locality. Therefore, the proposed stack cache serves as cache for a contiguous region in the memory. Two address pointers mark the part of the main memory that is in the stack cache. This window into the main memory is the cached region. As those two pointers clearly mark the cached region, no tag memory is needed.

The cached region is moved according to function calls and returns. On a function call the new stack frame moves this stack cache window. A new stack frame needs no fill from main memory as the stack allocated data is by definition undefined. If needed, stack data from outer frames is spilled to main memory. Traversing up the stack on functions return, stack frames are becoming inaccessible. This data is simply dropped and not written back to main memory. On function returns the stack frame of a function might need to be filled back into the stack cache with data from main memory.

In the stack cache for Patmos those operations are explicit and available as instructions. Those three operations are: (1) *reserve* words on the stack for the current function stack frame; data might be spilled to main memory when the stack is full. (2) *free* the current stack frame just before a return; only pointers are manipulated. (3) *ensure* that the stack frame is in the cache; executed after the call in the caller.

While these instructions present a clean interface to the stack cache, they need a change in the instruction set and in the compiler as well. In our proposal we derive those functions implicitly from the executed instructions. Therefore, no instruction set change or compiler change is needed. Our proposed stack cache can be added to any microprocessor.

The presented design is part of a research line towards time-predictable computer architecture [19] and extends prior work on exploring stack caches for average case performance [12]

This paper is organized in 6 sections: The following section presents related work. Section III presents the design and implementation of the stack cache. Section IV evaluates the stack cache. Section V presents three different approaches to include the stack cache in the WCET analysis. Section VI concludes.

II. Related Work

Hardware support for stack operations and explicit stack storage is uncommon in mainstream microprocessors. However, in the early days of computing, stack architecture was common.

With the introduction of the Java programming language, hardware implementations of the Java virtual machine, i.e., Java processors, appeared. As the Java virtual machine is a stack machine, those processors provide hardware support for stack storage and operations. picoJava [13], the first Java processor, provides 16 stack registers used as a ring buffer with automatic spilling to and filling from main memory. The Java processor jamuth [22] uses on-chip memory for stack allocated data.

The Java processor JOP [17] implements a stack cache with two caching levels. The two top elements are cached in dedicated registers and the elements deeper in the stack

are cached in on-chip memory. This architecture allows for a simple 3-stage pipeline for the stack operations and needs only an on-chip memory with one read and one write port. The result is more efficient, in both area and speed, than approaches caching only in a small register file [4] and approaches using only three-port on-chip memory for caching [7].

Stack caching has been explored using several approaches. Most of these use novel microarchitectures for stack caching, and some pair a new microarchitecture with static compiler optimizations to increase effectiveness.

Lu, Bai, and Shrivastava implemented scratchpad memory (SPM) stack caching for software managed multicore (SMM) architectures [11]. SMMs are multicore architectures where each core has an SPM, but no conventional cache, meaning all caching must be managed by software. Lu et al. focused on caching stack data and developed a scheme where stack frames, the size of the SPM, are swapped in and out of the SPM so that loads and stores always hit. To avoid thrashing that can occur when stack accesses are made to addresses on both sides of an SPM frame boundary, they developed a compiler heuristic named Smart Stack Data Management, which carefully chooses the locations of loads and stores to the SPM in order to minimize thrashing and SPM manager overhead. They compared SSDM with circular stack management, which keeps the top few stack frames in SPM, rotating bottom frames out to main memory when new frames are pushed and loading them back when the upper frames are popped. SSDM showed speedups between 15% and 1% compared to circular stack management. A similar approach, optimized for recursive functions, has also been explored [3].

Kannan, Shrivastava, Pabalkar, and Lee considered an SPM stack cache manager swapping stack frames the size of function frames instead of the entire size of the SPM [9]. This approach is more fine-grained than the one of Lu et al. but requires more management overhead since the SPM must be managed on every function call. Kannan et al. developed a static compile scheme for reserving space in the SPM based on the estimated call tree, maximizing the amount of space reserved at a time. They obtained significant speedup of over 40% using SPM caches of a size larger than the maximum stack frame created during benchmark execution, compared to a core with only a 1 KB normal cache. However, the SPM cache does not work for programs that create stack frames larger than the SPM.

Park, Park, and Ha explored caching stack data in an SPM acting as a cache that slides up and down in memory, following the stack pointer [15]. This is done dynamically without need for compiler support, using MMU faults to detect when the SPM must allocate space for new data or retrieve data from DRAM. Park et al. compared this form of stack caching with a standard cache architecture, configured to only cache stack data, and showed that the SPM stack cache was both faster and more energy efficient.

Lee, Smelyanskiy, Newburn, and Tyson developed the Stack Value File (SVF) microarchitecture [10]. The SVF is a circular buffer to which all memory accesses offset from the stack pointer are diverted. Since the stack is a contiguous data structure in memory the SVF is more area efficient than a comparable cache, as it only requires one tag line for every page it contains. It also reduces the memory traffic since unnecessary loads and stores of invalid data on stack allocation and dirty replacements can be avoided. The SVF microarchitecture requires no compiler support and produces large speedups compared to a baseline architecture with only a data cache, mostly because accesses are faster.

Olson, Eckert, Manne, and Hill examined the energy efficiency of using implicit and explicit stack caches [14]. An implicit stack cache constrains stack data items to be stored in only part of the available L1 data cache by limiting the available ways of associativity. While an explicit stack cache is a separate cache that handles only stack data accesses. Olson et al. identified that the separate stack cache need not be large compared to the L1 data cache and showed a reduction in dynamic cache energy consumption of 36% using explicit stack caching without negatively affecting performance. They also discussed making the explicit stack cache virtually addressed, removing 40% of address translations, which they found to be the average amount of memory accesses directed to the stack.

For real-time systems, it has been proposed to split the data cache [18]. The argument is that cache hits for heap allocated data is unpredictable, but that cache hits and misses for stack allocated data is relative easy to predict. Therefore, a split of the data cache into several caches (e.g., for stack, static, and heap allocated data) simplifies the worst-case execution time analysis.

In the real-time domain, Abbaspour, Brandner, and Schoeberl [1] implemented a stack cache for the Patmos processor [21] that requires compiler support. Their scheme uses three additional hardware instructions: reserve, free, and ensure. The compiler emits those instructions to make sure that the stack frame belonging to a function is in the cache. Therefore, cache misses can only happen at those stack cache manipulation instructions. All other loads and stores in the stack area are guaranteed hits. This allows entire stack frames to be kept in the stack cache to ensure time-predictable access times.

Abbaspour et al. showed that this scheme provides a large execution speedup of many benchmarks, even for small cache sizes (256 bytes). They also identified that the cached stack frames do not need to be held consistent with external memory, since data below the stack pointer is by definition invalid and therefore has no need of being written back to main memory. Tracking the stack allocated data within worst-case execution time analysis is simplified when the data cache is split [8]. An improvement of the stack cache with a lazy spilling pointer has been presented in [2]. There, an additional pointer tracks whether some stack allocated data is still coherent with the main memory and does not need to be spilled when space is needed in the stack cache.

Our proposed stack cache is similar to the stack cache in Patmos, as it also uses two pointers to indicate the cache content. In contrast, our proposed stack cache does not need any

special instructions and no compiler support. As optimization we use a dirty bit per cache line instead of a single lazy spilling pointer and it is therefore more fine-grained in detecting non-dirty cache entries that need no spilling to main memory.

III. The Stack Cache

Common programming languages use a stack-oriented structure to support function calls. This area is used for: (1) function arguments, (2) storing the return address, (3) providing storage for function local variables,[1] (4) locations for register saving, and (5) space for statically and dynamically allocated data structures. Some processors, e.g., x86, have explicit stack pointers and push and pop instructions to manipulate data on the stack. RISC style processors dedicate one general-purpose register by convention as the stack pointer and a second register as the frame pointer to allow dynamic data allocation on the stack. Stack allocated data can then be accessed with displacement addressing relative to the stack pointer or the frame pointer.

For historical reasons, the stack grows downwards and the top of the stack has the lowest address within the stack area. We keep this notion for the discussion in the paper.

A. Design

Stack allocated data is different from non-stack allocated data in two important ways. Firstly, it is always accessed in a relatively small region; the current stack frame. The size of the stack frame is determined as the memory region between the stack and frame pointers. If a frame pointer is not used, the size must be determined at compile time. Secondly, any data below the stack pointer in memory is by definition invalid, as it is not connected to any function in the call tree. Furthermore, data on the stack is initially undefined. This implies that we do not need to fill a cache line from memory when pushing data onto the stack. The individual words in the stack cache become valid when the program stores values, such as the return address and register spill slots, into the stack. Likewise, we do not need to write back a cache line when the data it contains has been popped from the stack. This data is by definition invalid.

The cache system of the processor is configured with the address range that is defined for the stack. All loads and stores to this address range are routed to the stack cache instead of the data cache.

As access to the stack allocated data is performed in a small, contiguous space in the memory at intervals in time, we can optimize the stack cache by using two pointers instead of tag memories to indicate the stack content. We use two pointers: (1) the stack cache top (*scTop*) pointer and the stack cache bottom (*scBot*) pointer. The two pointers define a window into the main memory that is currently cached in the stack cache. This window is a sliding window that changes when the program goes down and up in the call tree.

[1]With optimizing compilers, often used variables (and arguments) are allocated in registers and stack slots are used for register spills.

```
def access(addr)
  if (addr >= scTop && addr <= scBot)
    // a hit, nothing to do
  else if (addr < scTop)
    // Miss on a new cache frame
    // Maybe needs some spilling
    nspill = addr - scBot - SIZE + 1
    for i in range nspill
      M[scBot] = S[scBot % SIZE]
      scBot -= 1
    // Now fill the stack cache
    nfill = scTop - addr
    for i in range nfill
      scTop -= 1
      S[scTop % SIZE] = M[scTop]
  else if (addr > scBot)
    // Fill back a stack frame after a
    // return after spilling the words
    // that will be overwritten.
    nfill = addr - scBot
    for i in range nfill
      M[scTop] = S[scTop % SIZE]
      scBot += 1
      scTop += 1
      S[scBot % SIZE] = M[scBot]
  end if
```

Fig. 1. Stack cache handling when accessing data at address `addr`.

Access to a data item not yet in the cache leads to a cache miss, a change in one or both of the two pointers, a possible cache spill, and a cache fill. When a function is called and access to function local data leads to a miss, the *scTop* pointer is moved downwards till the address of the missed data item and all cache lines up to that one are loaded into the stack cache. When this loading of a cache line exceeds the size of the stack cache, the *scBot* pointer is moved as well and cache lines are spilled to the memory accordingly.

On a miss after a return, the *scBot* pointer is moved up to the address of the missing cache line and the cache is filled up to that line. If this filling would exceed the cache size, the *scTop* pointer is moved accordingly. For the non-optimized version the cache lines are spilled to main memory.

Figure 1 shows the stack cache function in pseudo code. `S` is the stack memory, `M` the main memory, and `SIZE` the size of the stack cache. Any access (load or store) to the stack address range with address `addr` is checked for cache hit or miss in the stack cache.

The pseudo code shows the stack cache with byte addressing. However, in our implementation the granularity of pointer movement and spilling and filling is in full cache lines. The length of the cache line depends on the property of the main memory and is usually in the range of 16 to 32 bytes.

Furthermore, the pseudo code omits the two optimizations we explored: (1) avoiding filling in stack data from main memory when accessing a new stack frame after a call and (2) avoiding spilling unchanged data to main memory.

B. Implementation

We have implemented the stack cache with small state machines, representing the stack cache controller. As proces-

sor for the implementation we use Patmos [21], which also supports the stack cache with compiler support. Therefore, we can compare these different stack cache architectures.

When a stack access with an address smaller than the *scTop* pointer is detected, the stack cache controller will begin moving the *scTop* pointer, line by line, until it reaches the accessed address. No cache lines are filled during this process, but cache lines may be spilled to memory if the cache is full. The *scBot* pointer always points to the cache line that must be spilled when the cache is full, because this represents the bottom of the area of the stack currently cached.

When popping items off the stack, the pointers will not move until an access is made to an address that is larger than the *scBot* pointer. At this point the *scBot* pointer will move one cache line at a time, filling in stack data that has previously been spilled to memory. During this phase, it will not write back any cache lines to memory, as whatever data is overwritten in the cache is invalid. When moving either *scBot* or *scTop* would cause the difference between the pointers to become larger than the cache size, both pointers are incremented or decremented depending on what action is being performed.

Any access between the two pointers is a hit because the *scBot* and *scTop* pointers guarantee that a contiguous region in memory is held in the stack cache. A side effect of this is that no tag memory is needed, as the tag addresses can be inferred from the two pointers.

As an optimization we added dirty bits to the cache lines to avoid spilling clean cache lines to main memory. This is similar to the optimization presented in [2], but works at a finer granularity.

A further optimization we explored is to not write back data when accessing items below the currently cached part on the stack. With this optimization, stack frames larger than the size of the stack cache must not exist. If an access was made to the top of the stack frame, followed by an access to the bottom, and then an access to the top again, the stack cache would not deliver the correct result as it has been thrown away.

C. Software Interface

The stack cache is intended to work completely transparent to the program in execution and needs no change in the compiler, compared to the Patmos stack cache. However, the detection of stack accesses depends on the address of the load or store instruction. Therefore, the stack cache needs to be configured at program load time with the address range of the stack area. However, this is only a minor modification of the program loader and no change in the application program is needed. No other assumptions on the application code or how the compiler generates code are made for this design.

IV. Evaluation

We have implemented the stack cache in the Patmos processor. Patmos is a RISC style processor intended as research platform for time-predictable computer architecture. Patmos already contains the original stack cache with instruction set and compiler support, making it the ideal platform to compare the different designs. The initial stack cache and the compiler support can be disabled. Stack allocated data can then be cached either in the data cache or in our proposed stack cache.

The original version of Patmos contains a write-through data cache, as the status of dirty flags is not tracked by current state-of-the-art WCET analysis tools. However, to enable another comparison we added a write-back data cache to Patmos.

Our main design goal is a time-predictable solution optimized for WCET analysis. However, this shall not result in a slow design in general. Therefore, in the following section we compare average-case performance measurements against a data cache, a variation of the stack cache, and against the Patmos stack cache. We sketch possible ways for WCET analysis in the following section and consider a WCET analysis based comparison as future work.

A. Evaluation Setup

We perform all the evaluation in the hardware implementation of Patmos. This is in contrast to former stack cache papers, where all the evaluation has been performed in a software simulation. Evaluation in a software simulation is a valid first step, but the definite evaluation is best done in real hardware.

We evaluate the performance of the stack caches with benchmarks from the TACLeBench benchmark collection.[2] In this paper we use the version 1.0 of the TACLeBench, which consists of a collection of 101 programs. All TACLeBench programs are completely self-contained without the need for external library or file IO. This feature makes this benchmark collection especially interesting for bare-bone embedded systems like our Patmos system. However, some benchmarks provide no results and no side effects. Therefore, our compiler at optimization level O2 will optimize most of the code away. The TACLeBench group is aware of this issue and will adapt the benchmarks for version 2.0.

As the benchmark suite contains currently 101 programs we need to perform a selection. We selected benchmarks that have a runtime higher than 10000 and less than 1000000 clock cycles. The first restriction is to avoid toy examples and programs optimized to merely a return statement. The second restriction is to make the benchmarking practical.

The data cache for non-stack allocated data is in most setups a 2 KB write-through cache. It is write-through as this is the proposed architecture for time-predictable processors [23]. As the benchmarks have a very small memory footprint, we select relatively small cache sizes. With larger caches all data will fit into the cache and we would not observe any cache misses. For measurements with larger caches we would need larger, more realistic, application benchmarks.

We use a Patmos configuration for the Altera DE2-70 FPGA board with 2 MB of main memory. This memory is a relative fast, 32-bit wide synchronous SRAM that is accessed in pipeline mode. A transfer of 4 32-bit words takes 7 cycles to complete. To this configuration we add a combination of

[2]https://github.com/tacle/tacle-bench

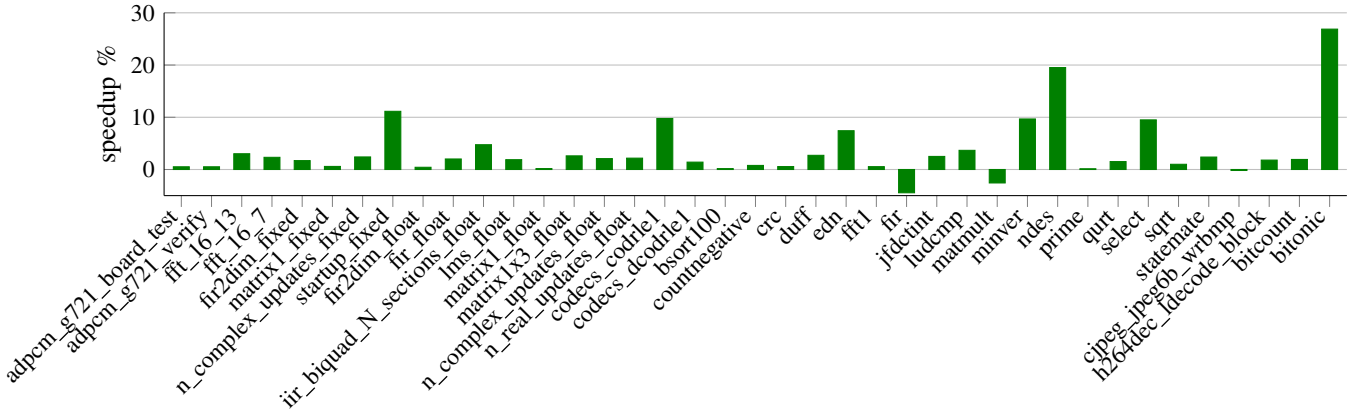

Fig. 2. Speedup obtained by replacing a 4 KB write-through data cache with a 2 KB write-through data cache and a 2 KB stack cache.

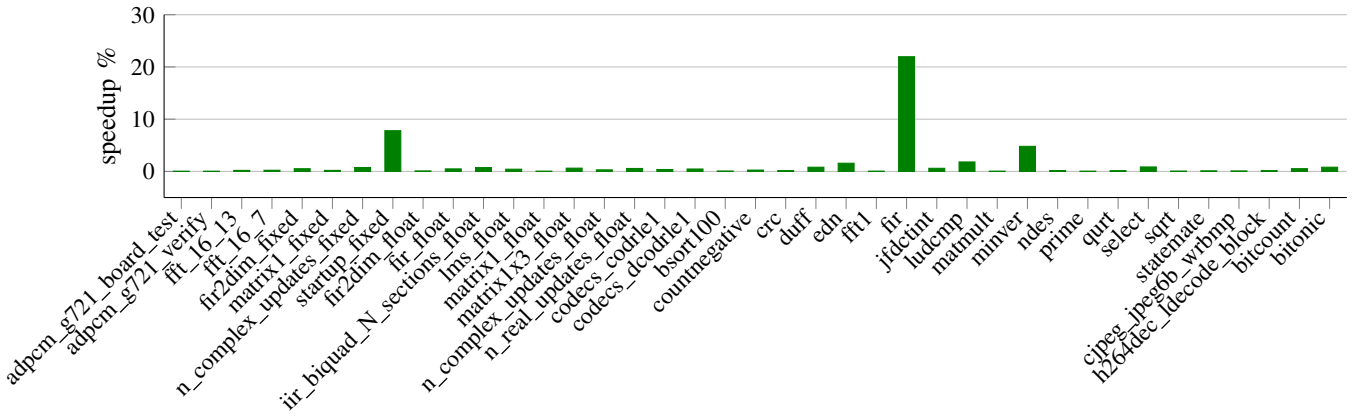

Fig. 3. Comparing the optimized version of the stack cache (dirty flags and no fill on allocation) with the non-optimized version.

data and/or stack caches. The size of the stack cache is 2 KB in all experiments.

A hit in all caches takes 1 cycle to complete, while a miss in the write-through data cache takes 8 cycles to complete. A miss in the write-back data cache takes double cycles when the cache line is dirty. Misses in our stack cache take a variable amount of time to complete, determined by whether the access was below the *scTop* pointer or above the *scBot* pointer, and the distance to the relevant pointer. In the original Patmos stack cache, misses may occur only at the *reserve* or *ensure* instructions.

B. Average Case Performance

First we compare the combination of our stack cache, that does not fill or spill invalid data to memory and uses dirty bits, combined with a 2 KB write-through data cache against a 4 KB write-through data cache. The results can be seen in Figure 2 and they show speedups for all but the `fir` and `matmult` benchmarks. Those two benchmarks access static arrays and benefit from a larger data cache. The speedup observed is probably the result of the stack cache being a write-back cache instead of using a write-through data cache for stack allocated data. A similar speedup has been observed in [1]. We assume that this speedup is as well due to the write-back characteristics of the original Patmos stack cache beating the write-through data cache.

The effectiveness of not filling cache lines from memory when writing new items to the stack and not writing undefined data back to memory, will depend on the memory access pattern of the benchmarks. Benchmarks that do not use more stack memory than the size of the stack cache will not be greatly affected by the optimization. Figure 3 compares a stack cache with filling optimization and using dirty bits to avoid spilling of clean data with a stack cache where those optimizations are turned off. Except in three cases, the speedup obtained by those two optimizations is negligible. Therefore we assume that: (1) active stack data is small and usually fits completely into the stack cache and (2) lines fetched into the stack cache are usually modified and therefore need to be spilled to memory when there is need for more space in the cache.

From this result we derive that the unsafe optimization of avoiding filling the stack cache on allocation can be turned off. The dirty bits for avoiding spilling clean cache lines to main memory are relative cheap and can be used anyway. The usage

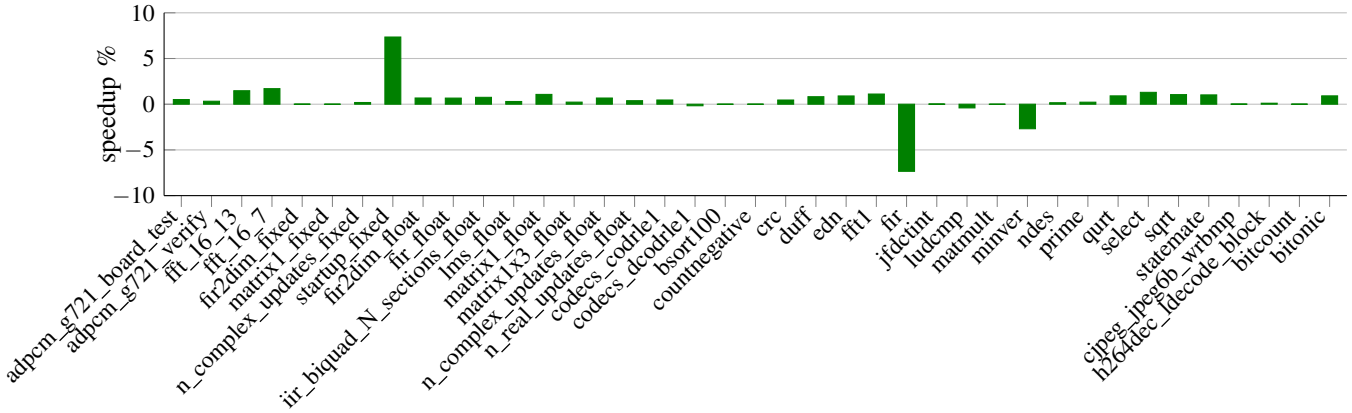

Fig. 4. Comparison of our stack cache with the original Patmos stack cache.

of dirty bits is similar, but with a higher accuracy, to the lazy spilling pointer presented in [2]. The authors of [2] observe a higher speedup with their lazy pointer than we observed with dirty bits. This might be due to the very small stack cache sizes of 128 and 256 bytes used in their evaluation.

Our stack cache is different from the original Patmos stack cache in that it does not need compiler-generated instructions to reserve and free stack data in the stack cache. It also does not need to ensure that the entire stack frame of the currently executing function is in the stack cache; instead it only contains the area of the stack that the function has tried to access. However, since we have implemented our stack cache within Patmos, we can easily compare those two caches within the same processor with the same instruction set. We compile the benchmarks for our stack without the stack cache instructions and for the original Patmos stack cache with the stack cache instructions.

Figure 4 shows the difference in execution time between a no fill/no spill stack cache with dirty bits and the original Patmos stack cache. Our stack cache performs similar to the original Patmos stack cache, but does not need compiler support. On most benchmarks it even performs a little bit faster (in the range of 1–2 %) than the original Patmos stack cache. The reason is that our stack cache handles all stack accesses, whereas the original Patmos stack cache excludes some stack allocated data from the stack cache. The following data types are not allocated on the Patmos stack: too large data, dynamic allocated data (with `alloca()`), and data that might escape (a pointer to stack allocated data is passed to a called function). These objects are directed to a so-called shadow stack. This shadow stack is cached in the (write-through) data cache.

One would expect that the new stack cache should always be better than the original stack cache as it can handle more data in the stack cache and the processor does not need to execute the stack manipulation instructions. However, our stack cache is not as fast as the original stack cache when initializing new stack frames. With the original stack cache this *reserve* operation can be executed in a single cycle when there is no need to spill cache content to the main memory. With our new stack cache, this operation is performed sequentially for each cache line.

This effect of of the higher cost for the *reserve* operation can be observed in benchmarks `fir` and `minver`. The `fir` benchmark is a single function with a loop including a division function. Division in Patmos is implemented in software and results in a function call. The division function reserves data on the stack cache, but as this benchmark needs not much stack allocated data, the stack cache never spills to main memory. Therefore, the reserve without spilling results in the overhead for the new stack cache. The `minver` benchmark is of similar structure. It is a program where all stack data fits into the stack cache and library calls are executed for software floating point operations.

We also compared the combination of a 2 KB write-through data cache and 2 KB stack cache with a 4 KB write-back data cache. In the average case the combined 4 KB write-back cache performs always better. However, a write-back data cache is not analyzable by current WCET analysis tools. Furthermore, we have then again the mix of different data types in the same cache. This is why a write-through cache is used as the standard data cache in the Patmos processor. The stack cache on the other hand, is simple to analyze with respect to WCET, and can therefore accompany a write-through data cache.

C. Resource Usage

The post-fitting resource consumption on an Altera EP4CE115 FPGA can be seen in Table I. The table shows consumption of logic cells (LC), which are the basic building blocks in an FPGA, and memory bits, which are mapped to on-chip memory blocks. The resource consumption is shown for the stack cache only; it does not include the resources for the data cache. To put this in perspective, a standard MIPS style 5-stage processor pipeline can be built in 3000–4000 LCs. The Patmos processor configured with a dual issue pipeline and implementing the register file in logic cells consumes about 12000 LCs and as a single-issue processor 6000 LCs.

TABLE I
RESOURCE USAGE.

Stack cache	Logic cells	Memory bits
2 KB non-optimized w/ dirty	1250	16384
2 KB spill/fill optimized w/ dirty	1440	16384

Therefore, we consider the resource consumption of the stack cache moderate.

V. WCET ANALYSIS

The main purpose of time-predictable architectures is to simplify, or sometimes even enable, WCET analysis of advanced features in a processor.

The stack cache has the purpose to simplify static WCET analysis of programs. The main topic of this paper is the design and hardware implementation of the proposed stack cache in a RISC style processor and provide a measurement based evaluation and comparison with the original Patmos stack cache. In this section, we discuss possibilities of WCET analyses for the proposed stack cache. We consider the concrete implementation of a WCET analysis tool out of scope for this paper and consider it as future work. We outline three routes to analyze the WCET behavior of the presented stack cache: (1) a simple all fit analysis, (2) data-flow based fill level analysis, and (3) scope based analysis.

A. Simple All Fit Analysis

The call depths of many embedded applications are very shallow. Even so shallow that all stack allocated data fits into the stack cache. Various Java processors, such as JOP [17] and jamuth [22], use this approach. Both Java processors contain a configurable, but fixed on-chip stack memory. And both have been used in real-world embedded applications, showing that a fixed stack size is not so restricted.

For embedded systems without virtual memory maximum stack depths (and maximum dynamic memory consumption) needs to be analyzed to know whether the program can be executed in the available physical memory.

Industrial WCET analysis tool providers, such as AbsInt and Tdorum Ltd., also provide tools for statically analyzing maximum stack size. AbsInt's StackAnalyzer is integrated with their aiT WCET analysis in a single tool called a^3. Calculation of stack usage bounds is integrated in Bound-T [5].

B. Dataflow Analysis

For the original Patmos stack cache [1] we have presented an intra-procedural data-flow analysis and path searches on the call-graph to find worst-case bounds on stack cache spilling and filling [8]. The original stack cache has dedicated instructions that reserve, free, or ensure that stack frames are cached. The analysis uses those individual instructions for the analysis.

However, the automatic stack cache, as presented in this paper, behaves only in the worst case as "bad" as the original stack cache. When the original stack cache executes a reserve instructions to reserve 10 words on the stack cache it will do so independently of the actual usage. Our stack cache might use up to 10 words in the same function, but may require less, depending on the actual execution path within the function. The same is true on ensuring a stack frame after the return: the original stack cache ensures that the full stack frame is in the cache, while our stack cache will only load in used parts of the stack frame.

We can conclude that the original Patmos stack cache is a model for an upper bound of spills and fills for our stack cache. Therefore, we can reuse that WCET analysis, but we do not need the compiler adaptions for the stack manipulation instructions. Therefore, our presented stack cache can be added to a standard RISC processor without changing of the instruction set and compiler.

Furthermore, with a composable architecture [16] it is not important that we know the exact instruction where the stack cache might spill or fill. Therefore, we can use the analysis of the original stack cache as a first upper bound analysis for our stack cache.

C. Scope-Based Analysis

Another option for WCET analysis for caches is the scope-based analysis [6]. We have applied this scope-based analysis for an object cache for a Java processor [20]. We can envision using this technique for the stack cache as well.

Scope-based cache analysis may also be called local persistence analysis. Scope-based analysis tries to find (potentially large) scopes within the control-flow graph of a program where all cache elements fit into the cache; they stay persistent after being loaded into the cache. For the method cache or object cache, ILP constraints are added to those scopes that all methods or objects may miss maximal once per execution of the scope.

We can envision adapting this scope-based analysis for the stack cache. We search for scopes, bottom up in the call tree, where all live stack frames fit into the stack cache. For such a scope we have in the worst case the spill cost of the entire stack size. If the depth towards the scope is less than the stack size, we can reduce the spill cost accordingly.

Similar to the spill cost analysis we can search for scopes top down to bound the fill cost of stack frames after function returns.

We plan in future work to implement a scope-based WCET analysis for the presented stack cache and compare it with the Patmos stack cache analysis [8].

VI. CONCLUSION

Caches are important for good performance, but data caches are hard to analyze for the worst-case execution time. Access to statically unknown addresses destroys analysis information about the data cache content. To alleviate this problem we present a split cache where access to stack allocated data is redirected to its own cache. This allows for independent analysis of the data cache and the stack cache for the worst-case behavior. Furthermore, the highly spatially close access pattern

to the stack allocated data allows for a hardware optimization that avoids tag memories. Instead of tag memories two pointers define the valid blocks in the cache.

The evaluation performed in a hardware implementation of the stack cache in a RISC style processor shows average case performance gains compared to a write-through data cache. Compared to write-back data caches the average case performance is on par, but being a WCET friendly cache solution.

Acknowledgment

We would like to thank Wolfgang Puffitsch for his help on integrating and debugging the stack cache in the Patmos processor pipeline.

The work presented in this paper was partially funded by the Danish Council for Independent Research | Technology and Production Sciences under the project RTEMP, contract no. 12-127600.

Source Access

The Patmos processor and the compiler are available in open source as part of the T-CREST project and is hosted on GitHub:

https://github.com/t-crest

The described stack cache is available in a fork of the Patmos project at:

https://github.com/clauniel/patmos/tree/clauniel_scache/hardware/src/datacache

References

[1] Sahar Abbaspour, Florian Brandner, and Martin Schoeberl. A time-predictable stack cache. In *Proceedings of the 9th Workshop on Software Technologies for Embedded and Ubiquitous Systems*, 2013.

[2] Sahar Abbaspour, Alexander Jordan, and Florian Brandner. Lazy spilling for a time-predictable stack cache: Implementation and analysis. In Heiko Falk, editor, *14th International Workshop on Worst-Case Execution Time Analysis, WCET 2014, July 8, 2014, Ulm, Germany*, volume 39 of *OASICS*, pages 83–92. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2014.

[3] Angel Dominguez, Nghi Nguyen, and Rajeev K. Barua. Recursive function data allocation to scratch-pad memory. In *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems - CASES '07*, page 65, New York, New York, USA, 2007. ACM Press.

[4] DS Hardin. Real-time objects on the bare metal: An efficient hardware realization of the Java (TM) Virtual Machine. *Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, Proceedings*, pages 53 – 59, 2001.

[5] Niklas Holsti, Thomas Långbacka, and Sami Saarinen. Bound-T time and stack analyzer, reference manual. Technical report, Tidorum Ltd, 2013. available at http://www.bound-t.com/manuals/ref-manual.pdf.

[6] Benedikt Huber, Stefan Hepp, and Martin Schoeberl. Scope-based method cache analysis. In *Proceedings of the 14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014)*, pages 73–82, Madrid, Spain, July 2014.

[7] S.A. Ito, L. Carro, and R.P. Jacobi. Making Java work for microcontroller applications. *IEEE Design & Test of Computers*, 18(5):100–110, 2001.

[8] Alexander Jordan, Florian Brandner, and Martin Schoeberl. Static analysis of worst-case stack cache behavior. In *Proceedings of the 21st International Conference on Real-Time Networks and Systems (RTNS 2013)*, pages 55–64, New York, NY, USA, 2013. ACM.

[9] Arun Kannan, Aviral Shrivastava, Amit Pabalkar, and Jong-eun Lee. A software solution for dynamic stack management on scratch pad memory. In *2009 Asia and South Pacific Design Automation Conference*, pages 612–617. IEEE, January 2009.

[10] HHS Lee, M Smelyanskiy, CJ Newburn, and GS Tyson. Stack value file: Custom microarchitecture for the stack. *HPCA: Seventh International Symposium on High-Performance Computing Architecture, Proceedings*, pages 5 – 14, 2001.

[11] Jing Lu, Ke Bai, and Shrivastava Aviral. SSDM: Smart Stack Data Management for software managed multicores (SMMs). *DAC*, pages 1 – 8, 2013.

[12] Carsten Nielsen and Martin Schoeberl. Stack caching using split data caches. In *Proceedings of the 11th Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS 2015)*, pages 36–43, Aukland, New Zealand, April 2015. IEEE.

[13] J. Michael O'Connor and Marc Tremblay. picoJava-I: The Java virtual machine in hardware. *IEEE Micro*, 17(2):45–53, 1997.

[14] Lena E Olson, Yasuko Eckert, Srilatha Manne, and Mark D Hill. Revisiting stack caches for energy efficiency.

[15] Soyoung Park, Hae-woo Park, and Soonhoi Ha. A Novel Technique to Use Scratch-pad Memory for Stack Management. In *2007 Design, Automation & Test in Europe Conference & Exhibition*, pages 1–6. IEEE, April 2007.

[16] Peter Puschner and Martin Schoeberl. On composable system timing, task timing, and WCET analysis. In *Proceedings of the 8th International Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 91–101, Prague, Czech Republic, July 2008. OCG.

[17] Martin Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.

[18] Martin Schoeberl. Time-predictable cache organization. In *Proceedings of the First International Workshop on Software Technologies for Future Dependable Distributed Systems (STFSSD 2009)*, pages 11–16, Tokyo, Japan, March 2009. IEEE Computer Society.

[19] Martin Schoeberl. Time-predictable computer architecture. *EURASIP Journal on Embedded Systems*, vol. 2009, Article ID 758480:17 pages, 2009.

[20] Martin Schoeberl, Benedikt Huber, and Wolfgang Puffitsch. Data cache organization for accurate timing analysis. *Real-Time Systems*, 49(1):1–28, 2013.

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

[22] Sascha Uhrig and Jörg Wiese. jamuth: an IP processor core for embedded Java real-time systems. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, pages 230–237, New York, NY, USA, 2007. ACM Press.

[23] Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 28(7):966–978, 2009.