A Time-predictable Stack Cache

Sahar Abbaspour, Florian Brandner, Martin Schoeberl

Department of Applied Mathematics and Computer Science
Technical University of Denmark
Email: sabb@imm.dtu.dk, flbr@imm.dtu.dk, masca@imm.dtu.dk

***Abstract*—Real-time systems need time-predictable architectures to support static worst-case execution time (WCET) analysis. One architectural feature, the data cache, is hard to analyze when different data areas (e.g., heap allocated and stack allocated data) share the same cache. This sharing leads to less precise results of the cache analysis part of the WCET analysis.**

Splitting the data cache for different data areas enables composable data cache analysis. The WCET analysis tool can analyze the accesses to these different data areas independently.

In this paper we present the design and implementation of a cache for stack allocated data. Our port of the LLVM C++ compiler supports the management of the stack cache. The combination of stack cache instructions and the hardware implementation of the stack cache is a further step towards time-predictable architectures.

I. Introduction

To allow for static analysis of the worst-case execution time (WCET), we need time-predictable architectures. The EC funded project T-CREST[1] develops time-predictable multi-core systems for future embedded real-time systems [22]. T-CREST considers time predictability in (a) the processor, (b) the on-chip interconnect, (c) the memory hierarchy, and (d) the compiler. AbsInt's WCET analysis tool aiT will support the architecture features that are developed within T-CREST.

The basis of a time-predictable system is the processor. We use the processor Patmos [24] for the T-CREST platform. Patmos is a dual-issue RISC-style pipeline with full predication support. Patmos is intended as a research platform for time-predictable architecture features. In this paper we extend Patmos with time-predictable caching of stack allocated data.

Caches are essential parts of any embedded system nowadays, in order to shorten the average memory access time. However, in real-time systems, the WCET is more important than the average-case execution time. Cache analysis tries to predict hits and misses statically for the calculation of the WCET [4]. Without timing anomalies, unknown cache accesses can be classified as misses. A safe, but not practical, approach would be to classify all accesses as potential misses. In that case an architecture without a cache at all would perform better in the worst-case.

Storage for data structures can be categorized in three major types: global static data, stack allocated data, and heap allocated data. Global static data is easy to analyze since the addresses are determined during program linking. Heap allocated data are the most difficult types to analyze since the addresses of objects are unknown before running the program [21].

The stack area contains, besides the return address information and callee saved registers, also function local variables and data structures. As the access frequency on this data area is very high, the stack benefits from caching. A WCET analysis tool can statically determine the addresses of stack allocated data when the call tree can be determined and when there is no dynamically sized allocation on the stack.

This paper describes the design and implementation of a cache for stack allocated data. We call this cache the *stack cache*. The stack cache is optimized to simpliy the cache analysis of static WCET analysis. The stack cache is managed in program scopes. These program scopes can be, in the simplest form, functions. We will use functions to explain the stack cache in the rest of the paper.

On a function call a stack frame is *reserved* in the stack cache. On the function return this same stack frame is *freed*. On returning to the caller, the stack frame of the caller is *ensured* to be in the stack cache. These three operations are instruction in the Patmos instruction set and are emitted by the compiler. Exchange with the main memory can only happen during a reserve operation or during the ensure operation. Therefore, all load and store instruction that access the stack cache are guaranteed hits. This simplifies the cache analysis, as fewer instructions need to be tracked within the data-flow analysis.

Furthermore, a cache that serves only stack allocated data can be optimized for this type of data. E.g., a new stack frame for a function call does not need to be cache consistent with the main memory as the local variables in a C function or Java method are undefined after the call or invoke and need to be initialized. The cache area for the new stack frame can be allocated without a cache fill from the main memory. On a return, the previously used cache blocks are marked invalid by the free instruction, as function local data is not accessible after the return. As a result, there is no need to write back cache lines after returning from a function.

When the cache overflows, the oldest frames of the stack cache are spilled to the main memory. Therefore, when a function returns and the program goes up in the call tree, there may be cache misses (triggered by the ensure instruction) and the spilled frames are loaded back into the stack cache.

The above, brief explanation of the stack cache assumed op-

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

erations on stack frames on function entry and exit. However, as it will be explained later, the processor has dedicated stack manipulations instructions to reserve stack space and to ensure that a region of the stack is in the cache. Therefore, this mechanism is not limited to function call and return instructions. An optimizing compiler can place space reservation, freeing, and ensuring at any feasible points in the program scope.

The paper is organized as follows. Section II presents related work. Section III describes the mechanics of the stack cache and Section IV the implementation of the stack cache in the RISC-style processor Patmos and in the compiler. Section V shows the evaluation of the stack cache with embedded benchmarks on a cycle accurate simulation of Patmos. Section VI discusses the findings and presents ideas for future work. Section VII concludes the paper.

II. Related Work

Several groups are working on time-aware architectures. The precision timed (PRET) architecture is first presented in [8]. PRET implements a RISC pipeline and performs chip-level multithreading for several threads to eliminate data forwarding and branch prediction [11]. The first FPGA version of PRET implements the ARM instruction set [9], [10]. Current work is ongoing to implement a PRET processor with the RISC-V [26] instruction set and to allow a variable number of hardware threads.[2] In contrast to our proposal, PRET contains no caches, but uses a compiler- or programmer-managed on-chip scratchpad memory.

One approach to avoid conservative cache miss assumptions in the WCET analysis is to mark data, where the address cannot be statically predicted, as un-cached [12]. The WCET analysis tool is used to determine predictable cache accesses and bypass unpredictable ones. The intention is similar to our approach. In Patmos we support typed memory load and store operations to distinguish between different on-chip memories/caches.

The Bell Labs C machine is an approach to support the C programming language [2]. In this machine a stack cache, implemented as a circular buffer, is replacing the registers of a processor. The argument is that register allocation is hard for compilers. However, current compilers perform well on register allocation. Therefore, in our Patmos processor we support a standard RISC registers set. The stack cache serves the on-chip memory for fast register spills and fills.

Similar to the Bell labs research group, the RISC research group at Berkeley considered hardware support for function calls necessary. From the first RISC architecture on, register windows were supported [17]. In RISC I [15], a new window of registers in the large register file is allocated to functions upon calling them. If a call overflow/underflow happens, a software routine adjusts the stack area in the main memory to save/restore the registers. Patterson [16], presents a hardware solution to keep operands in registers. In this method, many register windows are considered for different function calls. The register windows overlap to improve performance and avoid the copying of parameters from one register window to the next on a function call. The architecture of register windows is used in the follow-up SPARC architectures by Sun Microsystems. In contrast to register windows, the callee saved registers need to be spilled and restored explicitly using a stack cache. However, a cache can be made larger than a register file.

The Java virtual machine (JVM) is a stack machine. Therefore, hardware implementations of the JVM have usually on-chip memory to cache the stack content. PicoJava uses a hardware stack to support the stack-based architecture of the JVM [14]. A circular register file implements the stack cache. A dribbling mechanism handles the validity of data in the cache. The spilling and filling to respectively from memory is triggered by *watermarks*, which are defined in control registers.

Several other Java processors [20], [25], [27] support a simpler form of the stack cache. The on-chip memory needs to be large enough to hold the full stack of a thread. The stack content is only exchanged with the main memory on a thread switch. Therefore, the stack cache is part of the processor state. For a stack machine the two top elements of the stack cache can be implemented as dedicated registers, which can be directly accessed for the ALU operations [19].

M Huang et al. [7] argue that smaller caches consume less energy and propose to break the L1 cache into smaller structures to reduce the power consumption. The issue with cache power consumption is also addressed with using specialized caches such as the stack cache. The compiler usually inserts substantial amounts of spill codes during register allocation. Cooper and Harvey [1] use a small random access compiler-controlled memory to hold these spilled values. Using this method, most of the memory traffic due to compiler-inserted instructions is eliminated.

Gonzalez et al. [5] propose a dual data cache, including a spatial cache and a temporal cache. A locality prediction table, which includes information related to the load/store instructions, predicts the type of locality for each memory access. The main advantage of this method is that it can decide on temporal or spatial locality of vectors too. A similar idea of cache splitting was explored by Milutinovice et. al [13]. Data tagging is used to separate data with temporal locality from data with spatial locality. This eliminates fetching the entire block into the temporal cache. Therefore, a smaller prefetch buffer for spatial data cache and an overall smaller cache satisfy the needs. Our approach is to split the data cache to increase predictability [23].

Despite promising initial performance results of split data caches, mainstream architectures rely on unified data caches. The flexibility of using a single cache for different data areas is probably better for the average-case performance for a wide range of applications. Our hypothesis is that the benefit of better predictable data accesses is more important for real-time systems then the flexible usage of a single data cache.

[2]Private communication with the PRET research group.

```
void reserve(int n) {

    int nspill, i;

    sc_top -= n;
    nspill = m_top - sc_top - SC_SIZE;
    for (i=0; i<nspill; ++i) {
        mem[m_top] = sc[m_top & SC_MASK];
        --m_top;
    }
}
```

Fig. 1: The reserve instruction provides n free words in the stack cache. It may spill data into main memory.

III. The Stack Cache

The stack cache is a processor-local, on-chip memory. The stack cache operates similar to a ring buffer. It can be seen as a stack-cache-sized window into the main memory address range. To manage the stack cache, we use three additional instructions: reserve, ensure, and free. Two hardware registers define which part of the stack area is currently in the stack cache.

A. Stack Cache Manipulation

We present the mechanics of the stack cache in C code for easier readability. However, the hardware implementation is a synchronous design and the algorithm is implemented by a state machine that handles the memory spill and fill operations. In the C code following data structures are used:

- mem is an array representing the main memory,
- sc is an array representing the stack cache,
- m_top is the register pointing to the top of the saved stack content in the main memory, and
- sc_top points to the top element in the stack cache.

The two pointers are full-length address registers. However, when addressing the stack cache, only the lower n bits are used for a stack cache of a size of 2^n words. The constant SC_SIZE represents the stack cache size and SC_MASK is the bit mask for the stack cache addressing. The stack cache is managed in 32-bit words. Therefore, the pointers count in 32-bit words.

At program start the stack cache is empty and both pointers, m_top and sc_top, point to the same address, the address that is used for the stack area. m_top points to the next not-yet-used word in main memory. Similar, sc_top points to the next unused word of the stack cache. Therefore, the number of currently valid elements in the stack cache is m_top - sc_top.

The compiler generates code to grow the stack downward, as it is common for many architectures. Growing the stack downwards has historical reasons. However, for multi-threaded systems each thread needs a reserved, fixed memory area for the stack and there is no benefit from growing the stack downwards.

a) Reserve: The reserve instruction, as shown in Figure 1, reserves space in the stack cache. Typed load and store instructions use this reserved space. The reserve instruction may spill data to the main memory. This spilling happens when there are not enough free words in the stack cache to reserve the requested space.

The processor reads the number of words to be reserved (the immediate operand of the instruction) in the decode stage. The processor adjusts the sc_top register in the execution stage and also computes how many words need to be spilled in the execution stage. The processor spills to the main memory in the memory stage, as shown by the for loop in Figure 1.

```
void free(int n) {

    sc_top += n;
    if (sc_top > m_top) {
        m_top = sc_top;
    }
}
```

Fig. 2: The free instruction drops n elements from the stack cache. It may change the top memory pointer m_top.

```
void ensure(int n) {

    int nfill, i;

    nfill = n - (m_top - sc_top);
    for (i=0; i<nfill; ++i) {
        ++m_top;
        sc[m_top & SC_MASK] = mem[m_top];
    }
}
```

Fig. 3: The ensure instruction ensures that at least n elements are valid in the stack cache. It may need to fill data from main memory.

b) Free: The free instruction frees the reserved space on the stack. It does not fill previously spilled data back into the stack cache. It just changes the top of the stack pointer and may change the top of the memory pointer, as shown in Figure 2.

c) Ensure: Returning into a function needs to ensure that the stack frame of this function is available in the stack cache. The ensure instruction, as shown in Figure 3, guarantees this condition. This instruction may need to fill back the stack cache with previously spilled data. This happens when the number of valid words in the stack cache is less than the number of words that need to be in the stack cache. Filling the stack cache is shown in the loop in Figure 3.

One processor register serves as stack pointer and points to the end of the stack frame. Load and store instructions use displacement addressing relative to this stack pointer to access the stack cache.

B. Function Call Example

We illustrate the stack cache with an example of three functions: A calls B, which in turn calls C. Figure 4 shows the mapping of the stack cache to the main memory and the spilled content of the stack area in the main memory when the program is in function C. The blocks A, B, and C in the

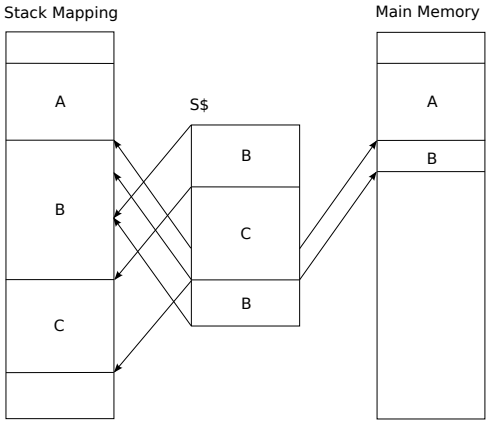

Fig. 4: Stack mapping, the stack cache content, and the spilled stack area.

figure are the stack frames for functions A, B, and C. The stack mapping shows the stack usage as it would be without a stack cache. The main memory figure shows which parts of the stack frames have been spilled into the main memory. And the middle box (S$) shows the content of the stack cache. We can see that stack frame A is fully spilled to main memory. Frame B is partially spilled and it also wraps around in the stack cache. Stack frame C is fully in the stack cache.

The program in this example starts with an empty stack cache. After entering function A, its stack frame is allocated using a reserve instruction. This stack frame is represented as block A in Figure 4. During execution of function A, function B is called. As before, B's stack frame is allocated by executing a reserve instruction, which checks whether the required space is available. Assuming that there is not enough free space in the stack cache, the stack frame of A is partially spilled to main memory. Then, another function call to C (Figure 4) causes the rest of the A's stack frame and a part of B's stack frame to be spilled to main memory. Before returning from C, its stack frame is freed. Since B's stack frame was partially spilled, an ensure instruction reloads the spilled parts of B's stack frame back into the stack cache. Similarly, the whole stack frame of A is reloaded after returning from B. Before returning from A, all the allocated space on the stack cache is freed and the stack cache becomes empty again.

IV. Implementation

We implemented the stack cache in hardware for the processor Patmos [24], extended the cycle-accurate software simulator of Patmos to support the stack cache, and adopted the LLVM-based compiler for Patmos to make use of the stack cache.

A. Instruction Set

To use the stack cache, we extended the RISC-style instruction set of Patmos with three stack cache manipulation instructions: reserve, ensure, and free. Furthermore, we added reading and writing of the stack cache registers for the program start and for a thread context change.

Load and store instruction in Patmos use displacement addressing with a normal register. One possibility is to assign a general purpose register as stack pointer and address the stack cache with this register. However, the stack cache register sc_top already points to the top of the stack. Therefore, we use addressing relative to this register for the stack cache. Typed load and store instructions distinguish between the stack cache access and other data areas.

Patmos is open source and we have the tool chain under control. Therefore, it is possible to use instruction set extensions for the usage and implementation of the stack cache. If the instruction set cannot be changed, e.g., it has to be ARM or MIPS compatible, a stack cache can be implemented as follows: an alternative to typed load store instructions is to use different address ranges to select different caches. The address range for the stack area needs to be loaded into the stack cache on program start. Memory mapped control registers for the stack cache can implement the stack cache manipulation instructions.

B. Hardware Implementation

To support the stack cache in Patmos, we added the required functionality to the execution and the memory stages. The execution stage computes new values for the stack cache registers and the condition if a spill or fill is needed. The memory stage contains the stack cache and performs the normal load and store operations. Furthermore, the memory stage contains the state machine for the spill and fill. On a fill or spill, the rest of the memory stage stalls the rest of the pipeline. In the following we describe the changes in the pipeline in more detail.

1) Decode Stage: According to the algorithms introduced in Section III, three instructions manage the stack cache. We extended the decode stage to support these instructions.

2) Execution Stage: The execution stage performs the necessary computations to update the two stack cache registers. If necessary, the execution stage determines also the number of words that the memory stage will spill of fill. The execution stage sets the spill/fill signals for the memory stage.

3) Memory Stage: In case of spill/fill operations (triggered by spill/fill signals from the execution stage), the memory stage stalls the other pipeline stages until the data transfer to/from main memory is completed. A state machine manages the spill/fill operations in the memory stage. This state machine keeps track of the number of spilled/filled bytes and deactivates the stall signal after all the bytes are spilled/filled. Furthermore, based on the current state of the state machine, the state machine adjusts the pointer to the top of the memory to access different words of the main memory and the stack cache on each spill/fill operation. There are two different types of transfer to/from the stack cache: (1) one directly through the load and store instructions, and (2) one caused by fill/spill state machine. Thus a multiplexer in the memory stage determines if the input data for the stack cache is coming from the main memory or from a normal store instruction. Similar, the write

path to the main memory is multiplexed between normal store instructions and the stack cache spill operation.

4) Write Back Stage: The write back stage is not affected by the stack cache instructions. The only difference is that the stall signal disables writing to the register file during the stall cycles.

C. Compiler Support

We have added support for the stack cache to the LLVM compiler, which supports the Patmos instruction set. First, loads and stores that shall end up in the stack cache use (stack) typed load and store instructions. Second, the compiler emits the reserve, free, and ensure instructions.

Local variables and data structures that are not accessed by a pointer cannot leak out of a function and can therefore be safely allocated on the stack cache. The compiler must also be able to figure out the maximum number of words used in a function. Therefore, memory allocated with alloca() with a non-constant size cannot be allocated on the stack cache. To fully support all legal operations on function local data of the C language, the compiler uses a second stack, the so-called *shadow* stack, for data which cannot be allocated in the stack cache.

As part of the function entry code, the compiler emits a reserve instruction to prepare the stack cache for the following load and store instructions. As part of the exit code, the compiler emits a free instruction to return the space on the stack cache. A ensure instruction after a call restores the call frame of the caller. Note that most ensure operations can be eliminated by the compiler in practice (see Section V).

It has to be noted that the usage of the stack manipulation instructions around function call and returns is only one way to use the stack cache. Any well-formed part of the program can serve as a region for stack cache manipulation. These scopes can be several regions within a function or it can be a cross-function scope.

V. Evaluation

We have implemented the stack cache in the Patmos processor in hardware and in the cycle accurate software simulation. In the evaluation section we report on the hardware size, compare with a standard caches. With the software simulation of Patmos we collect runtime statistics on stack and data cache usage with embedded benchmarks.

A. Hardware Resource Consumption

The Patmos processor and the stack cache are implemented in an FPGA. For the evaluation we use the Altera Cyclone II FPGA on the DE2-70 FPGA board. Without the stack cache, Patmos consumes about 2400 logic cells and can be clocked at 77 MHz. Adding the stack cache increases the total size of Patmos to about 3200 logic cells and reduces the maximum clock frequency to 73 MHz. We consider the additional 800 logic cells on the high side for this cache implementation. We intend to optimize the code for size and pipeline speed.

		Line Size (Words)		
	Cache Size (KB)	2	4	8
Direct Mapped	1	1.4	1.2	1.1
	2	2.7	2.3	2.2
	4	5.3	4.7	4.3

TABLE I: Total cache size in KB for different line sizes and cache sizes.

The size of the stack cache is configurable and for first spill and fill tests the default configuration is 64 32-bit words. Therefore, it consumes a single on-chip memory block.

The main differences between a normal data cache and the stack cache are:

- A normal data cache needs hit detection by comparing the tag memory of each way with part of the address. This hit detection is usually on the critical path. In can be performed in parallel on data read, but needs to performed before a write, which might add an additional cycle for a store instruction.
  With the stack cache we have guaranteed hits on load and store instructions and no need to compare with a tag memory. The equivalence to hit detection in the stack cache happens on reserve and ensure, which happens less often than loads and stores.
- A normal data cache needs a tag memory, which can consume a considerable amount of memory. In the stack cache only two pointers into the address space mark which data is in the cache and which data is only in the main memory.

In a standard cache, each cache line contains a tag word and a valid bit. The size of the tag word depends on the cache size, the cache line length, and the associativity. Table I shows the total memory size (tag and data memory) for different configurations of a direct mapped cache. We can see that the tag memory adds up to 40% of memory consumption to the data cache. In contrast our stack cache needs only the two registers to mark the address range that is in the cache. The two pointers also serve for the valid bit. Therefore, the total size of the stack cache is equal to size of the data memory used for a cache.

B. Cache Performance

The intention of the stack cache is to simplify WCET analysis by splitting different data areas to specialized data caches. Within the T-CREST project the WCET analysis tool aiT form AbsInt will be adapted to support the stack cache and other features of Patmos.

In the mean time we can evaluate the stack cache by average-case measurements. The T-CREST project also contains a cycle accurate software simulator of Patmos [3]. Therefore, it is possible to collect usage data and hit/miss rates for different stack cache configurations and compare them with a standard data cache.

In order to evaluate the stack cache we compiled and executed a subset of the publicly available benchmark suite MiBench [6] on the Patmos simulator. These benchmarks

cover a representative set of tasks often encountered in embedded systems, e.g., in telecommunication and automotive domains.

Each benchmark was first compiled to LLVM bitcode by the `clang` C frontend using optimization level `-O3` and then linked and optimized using `llvm-ld`. LLVM's `llc` tool generated Patmos machine code using two configurations: (1) with stack cache support enabled and (2) with stack cache support disabled, i.e., all stack data is kept in the main memory. With stack cache disable, the normal data cache caches the stack allocated data. The `gold` linker finalizes the code layout using a default memory layout, where all code and data sections are placed into Patmos' main memory.

The Patmos simulator executes the benchmarks and is configured with a 1/4 KB stack cache, a 8 KB data cache, and a 64 KB instruction cache, organized as method cache [18]. A larger stack cache of 1 KB is big enough to cache stack data without the need to spill stack frames to main memory. Therefore, we reduced the size of the stack cache to observe some spill and fill operations.

The stack cache is organized in word-sized blocks (4B), while the data and method caches are organized in 32-byte blocks. The 4-way set-associative data cache uses a least-recently-used (LRU) replacement policy and a write-through strategy with no-write allocation. The method cache likewise uses an LRU replacement policy. Transferring a cache block (32B) to or from main memory is assumed to take 40 cycles.

MiBench offers a small and a large data set for most benchmarks. We run most of the benchmarks with the default data set. For some benchmarks we use smaller data sets. This is indicated by the suffix in the figures.

C. Runtime

In our first experiment we compare the total number of execution cycles for each benchmark with stack cache support enabled against a variant with the stack cache disabled. Enabling the stack cache allows us to (partially) allocate the stack frame of functions to the stack cache instead of the main memory. This reduces the number of accesses through the data cache, but at the same time increases the number of instructions, because dedicated instructions, emitted by the compiler, manage the the stack cache explicitly. As shown by Figure 5, enabling the stack cache reduces the total number of execution cycles for certain benchmarks.

The gains are explained by (a) the additional cache space in the stack cache and (b) the handling of writes by the data cache. The additional cache space (1/4 KB) in the stack cache reduces the number of loads from the data cache and thus the number of accesses to the slow main memory. Additional gains are due to the long latency of stores to the data cache, which uses a write-through strategy with no-write allocation.

Some benchmarks profit less from the stack cache, most notably `crc-32`, `rawcaudio`, and `rawdaudio`. Loops that do not contain any spill code or function calls dominate these benchmarks. The simple stack cache allocation strategy thus cannot find any data to allocate to the stack cache.

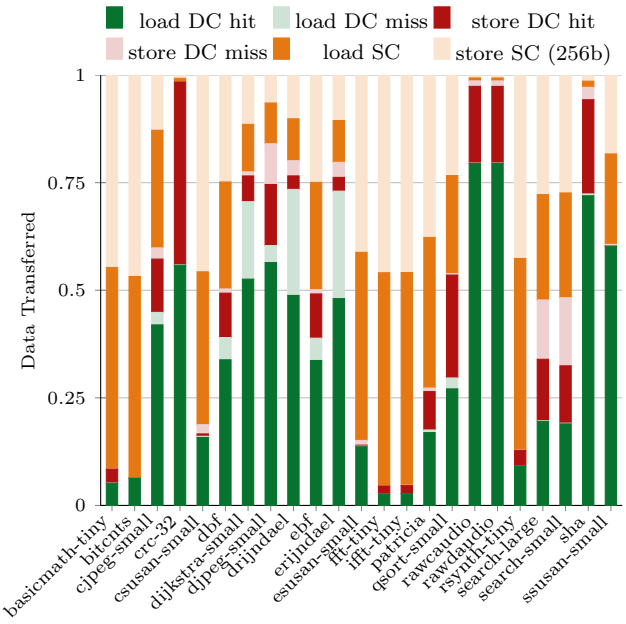

Fig. 6: Normalized transfer volume of data accesses to the data and stack cache for each benchmark (data cache 8 KB, stack cache 256 bytes).

We also evaluated the benefits of a simple compiler optimization that removes useless ensure operations (see SC optimized in Figure 5). An ensure can be eliminated after a call whose worst-case stack cache occupancy combined with the ensure's size does not exceed the stack cache size.

D. Stack Cache Utilisation

As shown before, using the stack cache can be quite profitable. We thus present some additional data characterizing how the various benchmarks use the stack cache. Figure 6 shows the transfer volume to and from the data and stack cache. The benchmarks that showed the best runtime improvements make heavy use of the stack cache. For instance, 79% of the memory accesses of the `bitcnts` go to the stack cache. With regard to data transfer volume, these numbers even increase. While roughly 65% of the accesses of the `patricia` benchmark target the stack cache, almost 75% of the transfer volume is serviced by the stack cache.

Benchmarks with little or no gain rarely make use of the stack cache, as shown by Figure 5. The main reason is that these benchmarks are dominated by a single loop without any spill code or function calls. This is confirmed by the numbers in Figure 7, which shows the amount of dynamically allocated data on the stack cache.

Indeed, the current algorithm to allocate data on the stack cache only leverages compiler-generated spill slots, which are often linked to function calls (saving/restoring registers before and after calls). However, we expect that more powerful allocation algorithms will be able to overcome this limitation. For instance, the `rawcaudio` and `rawdaudio` benchmarks mostly operate on small buffers, which are potential candidates for stack cache allocation.

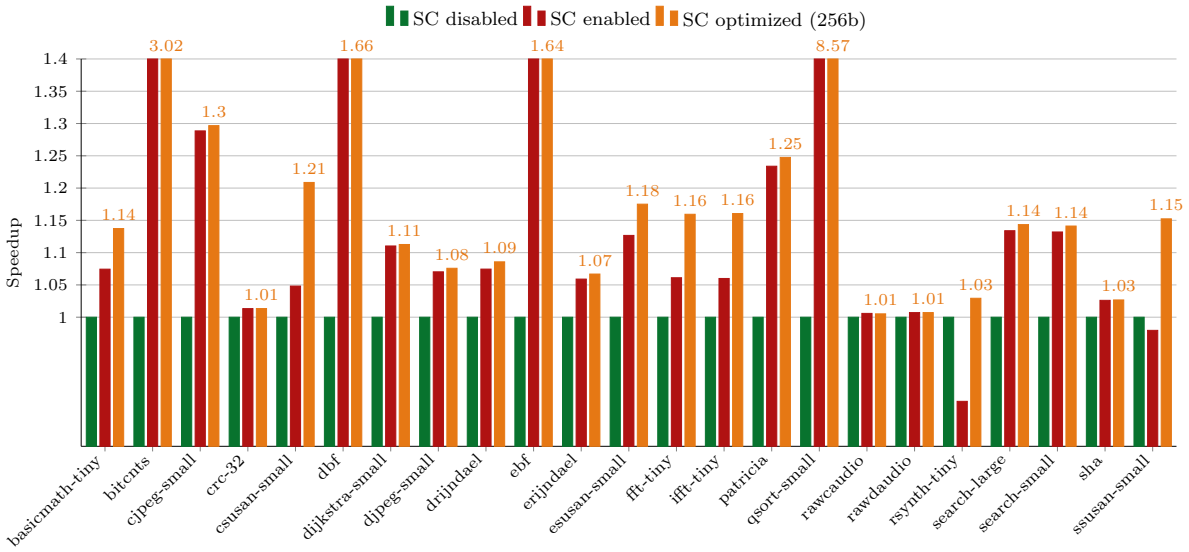

Fig. 5: Speedup when running benchmarks with stack cache support disabled, enabled, and enabled with optimization (cache size 256 bytes, larger is better).

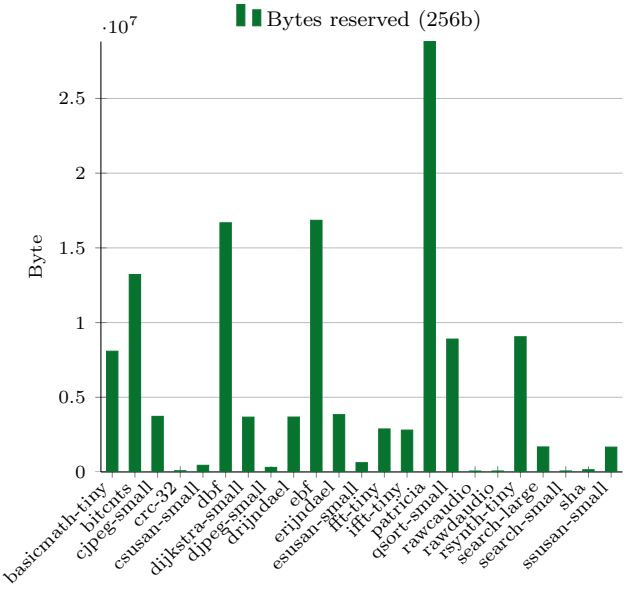

Fig. 7: Data dynamically allocated on the stack cache. The stack cache usage correlates with the number of executed function calls (cache size 256 bytes).

VI. Discussion

From the evaluation we have seen that even a very small stack cache (1 KB and less) serves well to cache the register spill and fill slots. As next step we would like to evaluate the stack cache with respect to the WCET. As AbsInt already provides a stack size analyzer tool StackAnalyzer, [3] we assume the that this is relative easy.

[3]see http://www.absint.com/stackanalyzer/index.htm

A. WCET Analysis

On a standard cache each load or store instruction can result in a cache miss and needs to be considered by program analysis. The explicit management of the stack with reserve and ensure by the compiler defines the programs points where an interaction with the main memory (spill or fill) *might* occur. All other accesses to the stack allocated data are guaranteed hits.

A simple, but very conservative, WCET estimate for the stack cache is to assume a spill and fill on every reserve and ensure instruction. Although this is conservative, all stack cache accesses within a function are guaranteed hits.

A simple improvement is to statically analyze the call tree and cut out all functions that are within a stack cache size from the call tree leaves. Such a part of the call tree needs to spill only one stack cache size, the rest of the stack reserve and ensure instructions need no spill or fill operations.

A more advanced analysis may perform data flow analysis of the two pointers m_top and sc_top to find the exact spill and fill points and sizes.

B. Stack Cache in Software

Our presented stack cache is one design point between two extremes: (a) a conventional cache, where the hardware decides and performs replacement of individual cache lines and (b) a scratchpad memory SPM that is used for stack allocated data. With a SPM the spill and fill of stack frames is performed in software. A compiler can implement the same algorithm that we have presented in this paper for the SPM. The main difference is that the stack pointer as well needs to be manipulated to always point into the SPM. One issue is, when a stack frame is allocated over the SPM boundary. Either crossing this boundary needs to be avoided, or the address for each load and store needs to be masked to point into the SPM.

VII. Conclusion

Real-time systems need time-predictable architectures to support static worst-case execution time analysis. One architectural feature, the data cache, is hard to analyze when different data areas (e.g., heap allocated and stack allocated data) share the same cache.

To simplify the data cache analysis we propose to split the data cache into different caches, serving different data areas. In this paper we present a cache for stack allocated data. Stack addresses are relative easy to predict statically and therefore the stack cache is easy to analyze.

One benefit of a specialized cache for stack allocated data is that it can be optimized for stack data. The exchange between the stack cache and main memory can be restricted to dedicated program points and needs not be considered on each load or store. An unused stack frame after a return needs not to be written back to main memory.

We have implemented the stack cache in hardware in the time-predictable processor Patmos. Furthermore, benchmarking of standard embedded benchmarks showed that even a small stack cache provides a good hit rate.

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).

Source Access

We provide the code of the processor (including the stack cache), the processor simulation, and the compiler in open source at: https://github.com/t-crest

References

[1] K. D. Cooper and T. J. Harvey. Compiler-controlled memory. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems (ASPLOS-VIII), ACM SIGPLAN*, pages 2–11. ACM, 1998.

[2] D. R. Ditzel and H. R. McLellan. Register allocation for free: The c machine stack cache. In *Proceedings of the first international symposium on Architectural support for programming languages and operating systems (ASPLOS-I), ACM SIGARCH*, volume 10, pages 48–56. ACM, 1982.

[3] DTU. D 2.1 software simulator of patmos. Technical report, T-CREST: http://www.t-crest.org/page/results, 2012.

[4] C. Ferdinand and R. Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2-3):131–181, 1999.

[5] A. González, C. Aliagas, and M. Valero. A data cache with multiple caching strategies tuned to different types of locality. In *ICS '95: Proceedings of the 9th international conference on Supercomputing*, pages 338–347, New York, NY, USA, 1995. ACM.

[6] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE 4th annual Workshop on Workload Characterization*, 2001.

[7] M. Huang, J. Renau, S. M. Yoo, and J. Torrellas. L1 data cache decomposition for energy efficiency. In *Proceedings of the 2001 international symposium on Low power electronics and design (ISLPED '01), ACM SIGDA*, pages 10–15. ACM, 2001.

[8] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee. Predictable programming on a precision timed architecture. In E. R. Altman, editor, *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2008)*, pages 137–146, Atlanta, GA, USA, October 2008. ACM.

[9] I. Liu. *Precision Timed Machines*. PhD thesis, EECS Department, University of California, Berkeley, May 2012.

[10] 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.

[11] I. Liu, J. Reineke, and E. A. Lee. A PRET architecture supporting concurrent programs with composable timing properties. In *Signals, Systems and Computers, 2010 Conference Record of the Forty-Four Asilomar Conference on*, November 2010.

[12] T. Lundqvist and P. Stenström. A method to improve the estimated worst-case performance of data caching. In *Proc. 6th International Conference on Real-Time Computing Systems and Applications*, pages 255–262. IEEE Computer Society, 1999.

[13] V. Milutinovic, M. Tomasevic, B. Markovi, and M. Tremblay. A new cache architecture concept: the split temporal/spatial cache. In *Electrotechnical Conference, 1996. MELECON '96., 8th Mediterranean*, volume 2, pages 1108–1111 vol.2, May 1996.

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

[15] D. Patterson and C. Sequin. A VLSI RISC. *Computer*, 15(9):8 – 21, sep 1982.

[16] D. A. Patterson. Reduced instruction set computers. *Commun. ACM*, 28(1):8–21, 1985.

[17] D. A. Patterson and C. H. Sequin. RISC I: A reduced instruction set VLSI computer. In *Proceedings of the 8th annual symposium on Computer Architecture*, ISCA '81, pages 443–457, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[18] 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)*, volume 3292 of *LNCS*, pages 371–382, Agia Napa, Cyprus, October 2004. Springer.

[19] M. Schoeberl. Design and implementation of an efficient stack machine. In *Proceedings of the 12th IEEE Reconfigurable Architecture Workshop (RAW2005)*, Denver, Colorado, USA, April 2005. IEEE.

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

[21] M. Schoeberl. Time-predictable cache organization. In *Proceedings of the 2009 Software Technologies for Future Dependable Distributed Systems (STFSSD '09), ACM*, pages 11–16. ACM, 2009.

[22] M. Schoeberl. Time-predictable chip-multiprocessor design. In *Asilomar Conference on Signals, Systems, and Computers*, Asilomar, CA, November 2010.

[23] M. Schoeberl, B. Huber, and W. Puffitsch. Data cache organization for accurate timing analysis. *Real-Time Systems*, 49:1–28, 2012.

[24] 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)*, pages 11–20, Grenoble, France, March 2011.

[25] S. Uhrig and J. 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.

[26] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic. The risc-v instruction set manual, volume i: Base user-level isa. Technical Report UCB/EECS-2011-62, EECS Department, University of California, Berkeley, May 2011.

[27] M. Zabel, T. B. Preusser, P. Reichel, and R. G. Spallek. Secure, real-time and multi-threaded general-purpose embedded Java microarchitecture. In *Prceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*, pages 59–62, Lübeck, Germany, Aug. 2007.