A Software Managed Stack Cache for Real-Time Systems

Alexander Jordan
Department of Applied Mathematics and Computer Science
Technical University of Denmark
alejo@dtu.dk

Sahar Abbaspour
Department of Applied Mathematics and Computer Science
Technical University of Denmark
sabb@dtu.dk

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

ABSTRACT

In a real-time system, the use of a scratchpad memory can mitigate the difficulties related to analyzing data caches, whose behavior is inherently hard to predict. We propose to use a scratchpad memory for stack allocated data. While statically allocating stack frames for individual functions to scratchpad memory regions aids predictability, it is limited to non-recursive programs and static allocation has to take different calling contexts into account. Using a stack cache that dynamically spills data to and fills data from external memory avoids these problems, while its simple design allows for efficiently deriving worst-case bounds through static analysis.

In this paper we present the design and implementation of software managed caching of stack allocated data in a scratchpad memory. We demonstrate a compiler-aided implementation of a stack cache using the LLVM compiler framework and report on its efficiency. Our evaluation encompasses stack management overhead and impact on worst-case execution time analysis. The state-of-the-art worst-case execution time analysis tool aiT is able to correctly classify all stack cache accesses as accesses to the scratchpad memory.

CCS Concepts

•Computer systems organization → Real-time system architecture;

Keywords

Real-time systems, stack caching, WCET analysis, scratchpad memory

1. INTRODUCTION

To meet the timing constraints in a system with hard deadlines, the worst-case execution time (WCET) of each of its tasks needs to be bounded. Static analysis and longest path search compute a safe WCET bound, i.e., one that never underestimates the actual execution time [25]. Standard processors contain features that are optimized to improve the average-case execution time. Some of those features, e.g., random replacement in a cache, are hard, or even impossible, to be statically analyzed for their WCET contribution. Our research aims at developing computer architectures that are optimized for their provable WCET instead of their average-case execution time [17]. One example is time-predictable organization of caches.

A cache is an example of such a feature that is optimized to improve average-case execution time. Caches decrease the latency when data needs to be fetched from slow main memory. A cache stores recently used items, as the probability is high that those items are used in the near future. This is called temporal locality. Furthermore, as main memory has a high latency for the first word, but can provide several words in a burst, a cache stores those blocks of memory in cache lines. A cache line typically caches 16 or 32 bytes. Chances are high that these prefetched data is used in the near future, e.g., when accessing instructions in a basic block or when accessing a vector. This is called spatial locality. First level caches are most commonly organized as one cache for instructions and one cache for data.

While caches benefit performance on average, they also introduce additional state that needs to be considered in the WCET analysis. When static WCET analysis has to consider a data cache, statically unknown addresses, e.g., of heap allocated data, are an issue. Those unknown addresses prevent predicting whether a cache access is a hit or a miss and further destroys abstract cache state of the analysis. Splitting the data cache for different data areas can be used to improve the WCET estimation [19].

One solution in embedded real-time systems to avoid caches is to use a scratchpad memory (SPM). An SPM is a small memory that resides within the processor, similar to a first level cache. As the SPM is on-chip, the access time is usually a single clock cycle. This SPM is mapped into some address space distinct from the address space of the main memory. To benefit from the SPM, a programmer (or tool) needs to allocate data structures or instructions manually in the SPM. The main benefit of an SPM, compared to a cache, is that the accesses to data (or instructions) that reside in the SPM are guaranteed single cycle operations.

One candidate of data structures that can be allocated in an SPM is data allocated on the stack. The stack is a data area where stack frames of functions are allocated. These frames may contain the return address, local variables of a function, saved registers, and program allocated data. With optimizing compilers local variables are allocated in registers

RTNS '16, October 19-21, 2016, Brest, France

DOI: http://dx.doi.org/10.1145/2997465.2997488

and the stack contains register spill slots.

This paper presents software-managed caching of stack content in an SPM. The software stack cache is designed to be time-predictable. On function entry, space is reserved in the SPM for the stack frame and on function return this reservation is returned. Stack accesses are redirected to the SPM and are single cycle operations. Movement of stack data between the main memory and the SPM is performed in software. In contrast to hardware solutions, this proposal can be applied to any standard embedded processor that has an SPM.

The contributions of this paper are: (1) we describe a stack cache implemented in software using a standard SPM, (2) we evaluate the dynamic utilization of the SPM, and (3) we examine how static WCET analysis handles the stack-cache related memory accesses and spill/fill operations using aiT [7], a state-of-the-art WCET analysis tool.

This paper is organized in six sections: The following section presents related work. Section 3 provides details on the implementation of a software managed stack cache. Section 4 discusses its WCET analysis aspects. Section 5 evaluates the stack cache with regard to its behavior dynamically and during static analysis. Section 6 concludes.

2. RELATED WORK

Lee et al. [12] propose to divert all references to the stack data to a stack value file (SVF) instead of L1 data cache. Each memory location maps to a register in the SVF based on the lowest address bits. When the stack pointer value changes, data from the first level cache moves to the SVF.

For a stack machine, such as the Java virtual machine, the two top elements of the stack cache can be implemented as dedicated registers, which can be directly accessed during the ALU operations [16].

Bai et al. [2] propose a technique that can run any application on the limited local memory. In this work, a pointer is set to a global address rather than a local address. A function *s2p* converts the address by finding which function the pointer belongs to. Then the *s2p* function computes the offset of the pointer variable from the start of the frame in the local memory. Finally *s2p* returns the global address of the pointer.

Park et al. [14] use the memory management unit for a dynamic address mapping to the SPM. Their method requires no architectural modification or complier assistance and generates permission faults when access to the stack region is outside the SPM. Dominguez et al. [4] introduce a method to allocate stack data of recursive functions in an SPM. Profiling information helps to place the most commonly occurring stack depths in the SPM.

Circular stack management [10] is a software technique to keep the active stack data in the SPM. This work introduces a software SPM manager to check if there is enough space in the SPM. Lu et al. optimize the circular stack management approach in [13]. Their heuristic places stack data using a (profile-based) weighted call graph and thus aims to reduce the overhead of SPM management by increasing the granularity of accessing the main memory and eliminating calling the special functions that move data.

Kim [11] proposes a source code modification technique to fit the stack data into the SPM. When the stack data does not fit, the memory is divided into multiple same-sized blocks. Only the address of one block is dynamically allocated, and this block is managed as a single line direct mapped cache. However, the proposed technique cannot handle recursive function calls.

The allocation of data in the SPM can be optimized for the WCET [3, 22, 23]. Suhendra et al. use Integer Linear Programming to find an optimal allocation of data objects into the SPM. This work is further extended by search-based approaches to also consider infeasible program paths. The allocation of data in the SPM is static, i.e., fixed at compile time, where our software stack cache exchanges data with main memory in a time-predictable way.

Abbaspour et al. [1] propose a hardware stack cache. The stack cache is managed using three special instructions exposed in the instruction set: reserve, free, and ensure. Typed load and store instructions are used to access stack data. The additional stack cache space reduces the number of loads from the data cache and decreases the number of slow main memory accesses. Additionally, the stack cache eliminates the long latency stores to the write-through data cache. Worst-case spill and fill memory traffic can be statically analyzed [9]. This hardware stack cache provides the inspiration for the presented software stack cache with an SPM. While a hardware managed stack cache requires modifications to the instruction set to implement the stack cache semantics, we propose to implement the stack cache in software, with assistance of the compiler.

Schoeberl and Nielsen [20] explore a hardware managed stack cache that does not need any compiler support. This approach is the opposite of our approach, where we use standard hardware and implement stack caching with compiler support in software.

In our approach we provide compiler support to dynamically reuse the SPM for stack frames of different functions. An SPM can also be dynamically reused for different tasks in a preemptive multitasking system [24].

3. SOFTWARE MANAGED STACK CACHE

Each function in a program typically allocates space on the stack, i.e., its stack frame. A stack frame associated with a function contains information on the return address, saved register values, as well as function-local variables and data structures. This type of data is frequently accessed and thus benefits from caching. Moreover, compared to data allocated on the heap, it is easy to statically determine the addresses of stack allocated data during WCET analysis. Therefore, we propose a software managed stack cache using a processor-local SPM.

Implementing the software managed stack cache requires that the stack cache specific operations are available in software and support from the compiler. The compiler is responsible for placing calls to those management functions at certain program points and for performing address translation for stack loads and stores.

3.1 Stack Cache Management

The software managed stack cache is an SPM organized as a ring buffer, thus following a FIFO strategy. For manipulating this ring buffer, we define two pointers: *stack top* (`sc_top`) and *memory top* (`m_top`). The stack memory between these pointers is the memory that is currently cached in the SPM. The `sc_top` pointer refers to the address of the top of the stack data, which is cached in the SPM. The `m_top` pointer is the address of the first element that is not anymore cached

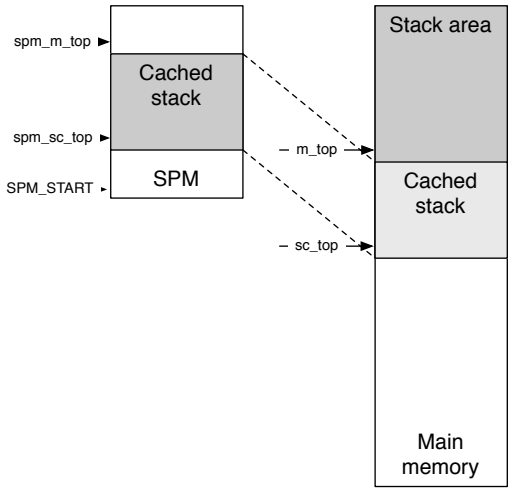

Figure 1: Stack caching in the SPM.

in the SPM, therefore the top element of the part of the stack that resides in the main memory. Both pointers point into the stack area in the main memory. To redirect stack accesses to the SPM, the addresses are *translated* to point into the address area where the SPM is mapped. The SPM can also be seen as a sliding window into the main memory.

Figure 1 shows the mapping of the stack area and the relation of the different pointers. The upper part of the main memory is reserved for the stack. From there the stack grows downward. The area between the largest address and `m_top` in the main memory, shown in dark grey, contains stack data currently stored in main memory. The area between `m_top` and `sc_top` in the main memory, shown in light grey, marks the address range that is currently cached in the SPM.

The SPM starts at address `SPM_START`. The area between the two pointers, shown in dark grey, contains the actually cached stack data. The pointers for the SPM are derived from `m_top` and `sc_top` as follows:

```
spm_x_top = (x_top % SPM_SIZE) + SPM_START
```

The modulo operation computes the SPM relative address and allows the stack cache area to wrap around the SPM. Note that the SPM size is usually a power of 2 and therefore the modulo operation is performed with a simple bit masking `AND` operation.

In the following we show the stack manipulation functions in pseudo code. Note that these functions are called by compiler-inserted code and not by application code. We assume that the stack grows downwards. The array `M[]` represents the memory, both the main memory and the SPM. For simplicity of the pseudo code we assume that the memory is organized in 32-bit words. The SPM is `SPM_SIZE` words large and starts at address `SPM_START`; it is represented as `M[SPM_START...SPM_START+SPM_SIZE-1]`.

Two processor registers are used for the stack cache: (1) the stack pointer `sc_top` pointing to the top of the stack, as index into the main memory and (2) the memory pointer `m_top` pointing to the last spilled word in main memory. At program start both pointers point to the same address, the word above the memory area reserved for the stack. The difference between these two pointers is the number of words from the stack that is cached in the SPM. In the following

```
def reserve(n):
  sc_top -= n
  n_spill = (m_top - sc_top - SPM_SIZE)
  for i in range(n_spill):
    m_top -= 1
    M[m_top] = M[SPM_START + (m_top %
        ↪ SPM_SIZE)]
```

Figure 2: The `reserve` function reserves `n` free words in the stack cache. It may spill data into main memory.

```
def free(n):
  sc_top += n
```

Figure 3: The `free` function drops `n` elements from the stack cache.

we describe the three stack manipulation functions: `reserve`, `free`, and `ensure`.

The `reserve` function is shown in Figure 2. This function is called on function entry to reserve the stack frame for the function. It reserves `n` words in the SPM for the called function by decrementing the stack pointer `sc_top` by `n`. If the new stack frame overlaps with older stack content in the SPM, that older stack data is spilled to the main memory.

The `free` function, as shown in Figure 3, frees the stack frame of a function. It is called before function return. No data is exchanged between the SPM and main memory. Only the stack pointer `sc_top` is incremented.

The `ensure` function, as shown in Figure 4, ensures that the actual stack frame is resident in the SPM. The caller calls `ensure` after function return from the callee. The caller's stack frame is `n` words large. If the content of the stack in the SPM is less than `n` words, the missing words are filled from main memory.

Load and store instructions access the stack area with a displacement `disp` relative to the stack pointer `sc_top`. As the stack data is always in the SPM, those loads and stores are single cycle latency, similar to a hit in a data cache. The implementation of stack load and store functions is shown in Figure 5. The load and store addresses need a translation from the stack area in the main memory to the address area where the SPM is mapped. In practice the size of the SPM (or the part that is used for stack caching in the SPM) is a power of 2. In that case the modulo operation is reduced to a simple bit masking `AND` instruction.

3.2 Compiler Support

An efficient implementation of the software stack cache

```
def ensure(n):
  n_fill = n - (m_top - sc_top)
  for i in range(n_fill):
    M[SPM_START + (m_top % SPM_SIZE)] =
        ↪ M[m_top]
    m_top += 1
```

Figure 4: The `ensure` function ensures that at least `n` elements are valid in the stack cache. It may fill data from main memory.

```
def load(disp):
  return M[SPM_START + ((sc_top + disp) %
      ↪ SPM_SIZE)]

def store(disp, val):
  M[SPM_START + ((sc_top + disp) %
      ↪ SPM_SIZE)] = val
```

Figure 5: Pseudo code for the stack accessing load and store instructions.

requires assistance of the compiler. By augmenting the calling conventions, we can reserve two registers for the `sc_top` and `m_top` pointers. Address translation from the stack address space of external memory to that of the SPM needs to be performed for every load from a source address and every store to a destination address. The compiler emits the arithmetic instructions for this purpose with the code that interacts with values on the stack.

Furthermore, to manage the stack cache, the compiler emits calls to the `reserve` and `free` functions within the function prologue and epilogue, respectively. A call to the `ensure` function is inserted after each call to another function in order to ensure that the stack frame of the caller is in the stack cache. However, it is not necessary to a call `ensure` after a call to a (leaf) function, which does not set up a stack frame of its own. This is a common technique employed by compilers and known as leaf optimization.

A stack cache cannot directly support certain objects that C-like languages allocate on the program stack. These are objects that are: (1) dynamically sized, (2) too large, or (3) escape the scope of the current function. The stack reservation (and freeing) is performed with a constant. Therefore, dynamically sized objects, created with `alloca()`, cannot be allocated on the stack cache. Objects that exceed the SPM size cannot be allocated in the SPM. If pointers to an object escape, to a callee, it cannot be guaranteed that the object is still cached in the SPM when the pointer is dereferenced in the callee. The stack area where this object is allocated might already be spilled to main memory. For all three cases, the compiler allocates these objects on a second stack outside the SPM cache.

At the compiler level, several optimizations can be applied to reduce the overhead of the stack cache: inlining of stack management functions reduces the overhead caused by calls. Given results from static analysis, further `ensure` operations can be avoided, when they are guaranteed to never reload data. Furthermore, for statically known pointer values, address translation for loads and stores can be simplified.

4. WORST-CASE EXECUTION TIME

Our motivation behind proposing a software stack cache is to develop computer architectures that are optimized for their provable WCET instead of their average-case execution time [17]. Therefore, we are interested how the software stack cache impacts static WCET analysis. Essentially, two classes of operations need to be considered by static analysis: (1) loads and stores that access the SPM allocated stack data and (2) the spill and fill operations.

4.1 Stack Access

An access to SPM allocated data uses a register as a stack pointer and performs address translation. The result of the address translation always points to data within the address range of the SPM. A WCET analysis tool can be configured with memory access latencies depending on address ranges. Therefore, as long as static analysis can determine that the effective address of a stack load or store points into the SPM, these instructions will have single cycle latency. Since translated addresses cannot escape (note the modulo operations in Figure 5), a local address (value) analysis is sufficient here.

As an alternative, the compiler, which necessarily knows about accesses to the stack cache, can share this information with the analysis tool. This could be achieved by providing per-instruction annotations.

4.2 Stack Spill and Fill

The second question with regard to static analysis is, whether the maximum number of fills and spills can be efficiently bounded through static analysis. Without any analysis, the conservative bound for each `reserve` and `ensure` operation would be based on its argument `n` and drastically overestimate the actual (worst-case) behavior. Tracking the worst-case state of the stack cache has previously been shown to be an inter-procedural analysis problem [9].

With the stack cache implemented in software and its internal state exposed by the `sc_top` and `m_top` registers, an alternative to a custom-built analysis is to rely on an existing value/loop-bound analysis to analyze worst-case spilling and filling. For the results to be precise enough to make this approach viable, the value analysis must be inter-procedural, context-sensitive, and has to support intervals. In Section 5.3, we show that AbsInt's aiT analysis based on abstract interpretation fulfills these requirements.

5. EVALUATION

To evaluate the efficiency of a software managed stack cache and its impact on WCET analysis, we consider our implementation for the Patmos processor [21], which is a RISC processor. Patmos and the compiler are available in open source. Furthermore, the aiT WCET analyzer supports the instruction set of Patmos. We changed the compiler as follows:

- Two (out of 32) general-purpose registers are reserved for the `sc_top` and `m_top` pointers.
- The stack cache management operations (`reserve`, `free`, and `ensure`) are implemented as functions. The small functions may be inlined by the compiler.
- The compiler calls the stack cache management functions at the relevant program points and emits instructions to perform address translation for stack loads and stores.

5.1 Setup

We measure the dynamic usage of different data areas with the MiBench benchmark suite [6]. All programs are compiled with the Patmos LLVM compiler (version 3.4) [15] with full optimizations (-O3). Global, link-time inlining is disabled. Measurements are performed using the cycle accurate platform simulator `pasim`. Our simulations assume a standard data cache alongside the software managed stack

cache. The data cache is 4-way set-associative with a write-through strategy, 32-byte cache lines, and least-recently-used (LRU) replacement. The data cache size is 2 KB. We explore the effect of different stack cache sizes on average-case performance of loads and stores. Since the result of this measurement is the number of loads and stores to different memory types, the instruction cache configuration has no influence on the results.

The small sizes for the caches (data cache and stack cache) have been chosen, as the embedded benchmarks from MiBench have a rather small memory footprint. Furthermore, as shown in Figure 6, already a small stack cache of 256 bytes caches most stack accesses.

The MiBench benchmark suite includes medium sized programs that give a good workload for the stack cache utilization measurement. However, as loops are often unbound in those benchmarks and system functions called, they are not a good fit for WCET analysis.

Therefore, for the WCET analysis we use the WCET benchmark program Debie1 [8]. Debie1 is based on the on-board software of a satellite instrument. We use the definition of the analysis problem from the 2011 edition of the WCET tool challenge.[1] Another option for WCET analysis would be the Malardalen benchmark collection [5]. However, those benchmarks are so small that most of the time local variables are allocated in registers and there is no activity on the stack. Compared to those benchmarks, Debie1 is a real application.

For WCET analysis, we use aiT version 14.04i [7], the state-of-the-art static analysis tool from AbsInt, which is part of `a3`. aiT has been adapted to support Patmos within the T-CREST research project [18]. The programs are compiled with full optimization (-O3). Global, link-time inlining is disabled. aiT is used with the default setting, the context sensitivity settings are: `interproc flexible, max-length = inf, max-unroll = inf, default-unroll = 2;`.

The data cache is configured as follows: 2 KB, 32-byte cache lines, 4-way set-associative, LRU replacement, and write-through. We configure aiT with a memory latency of 42 clock cycles for a burst of 32 bytes for a cache miss. This corresponds to the system's behavior on the DE2-115 Altera FPGA board. The SPM for the stack cache is 2 KB. Accesses to data in the address range of the SPM are configured as single cycle accesses. We perform WCET analysis with an ideal instruction cache and with an instruction cache of 4 KB, 32 byte cache lines, 4-way set-associative, and with LRU replacement.

5.2 Dynamic Stack Cache Utilization

To evaluate the effect of the software stack cache, we explore different scenarios with different stack cache sizes and a fixed data cache size of 2 KB. The results presented here are data derived from measurements with the cycle accurate Patmos simulator. We used the simulator to get detailed statistics of dynamic instruction counts, which is not possible in the current hardware implementation of Patmos.

Figure 6 shows loads and stores to different memory areas per 100 instructions executed. We present the numbers for three configurations: with no stack cache, with a stack cache of 256 bytes, and one with a stack cache of 2 KB.

The green areas represent accesses to the SPM (software stack cache), accesses that are guaranteed single cycle latency. These accesses are only present in the two bars that include a stack cache. These accesses can be quantified as single cycle accesses by a static WCET analysis tool. A large portion of those accesses improves the (predictable) performance.

The orange areas represent data cache accesses. In the bars that include a stack cache these accesses are to: (1) statically allocated objects, (2) heap allocated objects, and (3) objects allocated on the shadow stack. These accesses are hard to predict in static WCET analysis and are the target to be minimized.

Patmos supports data cache bypassing loads and stores. We use those instructions for the spill and fill operations between the stack cache and main memory. These uncached accesses are shown in blue at the top of the stacked bar chart. These accesses are practically not present in the configuration with a data cache only and when the stack cache is large enough.

Figure 6 shows that most of the benchmarks can benefit even from a small stack cache (256 B). With the small cache configuration we see some uncached loads and stores indicating some spilling and filling. With the 2 KB configuration of the stack cache we observe no spilling or filling in all benchmarks. In that case the stack cache works perfectly.

There are also benchmarks that do not benefit from stack caching. E.g., `rawcaudio` and `rawdaudio` are using mostly static data and the only function called uses few local variables that can be allocated in registers. Similarly, the benchmark `sha` uses only very few local variables. `crc32` is a single loop in the `main` function without any function call.

It should however be noted that using the software stack cache increases the number of instructions in total. Therefore, the cumulative accesses to the stack and data caches are different in the three configurations.

5.3 WCET Analysis

We explore the WCET analyzability of our stack cache with the state-of-the-art WCET analysis tool aiT from AbsInt [7]. We evaluate aiT's capability with regard to static analysis of stack cache accesses and spill/fill operations using a selection of small, medium, and large analysis problems for the Debie1 benchmark program. To focus on the effects of the data cache and the stack cache we assume an ideal instruction cache in the first experiment. As the additional instructions have an influence on the instruction cache we further provide results with an instruction cache.

To benefit from the stack cache for WCET analysis, we explore if a static WCET analysis tool can correctly determine that the accesses to the stack cache are single cycle operations and are not considered as accesses to main memory that are cached by the normal data cache.

The simple pointer arithmetic involved during the translation of addresses to the SPM space proves to be no problem for aiT's value analysis. With the correct access time for the SPM in the configuration, aiT assumes single cycle load and store latencies for all stack cache accesses during WCET calculation. This satisfies our requirement from Section 4.1.

In order to also properly bound the spill and fill activity (see Section 4.2), we must take care that (1) the use of the two reserved registers does not contradict any assumptions that the analysis tool has about the platform's calling conventions; and (2) that during analysis no assumed side-effects invalidate the machine state so that the stack cache state (i.e., the values of its pointer registers) is also affected.

With our implementation, we did encounter the latter

[1] http://www.mrtc.mdh.se/projects/WCC/2011/

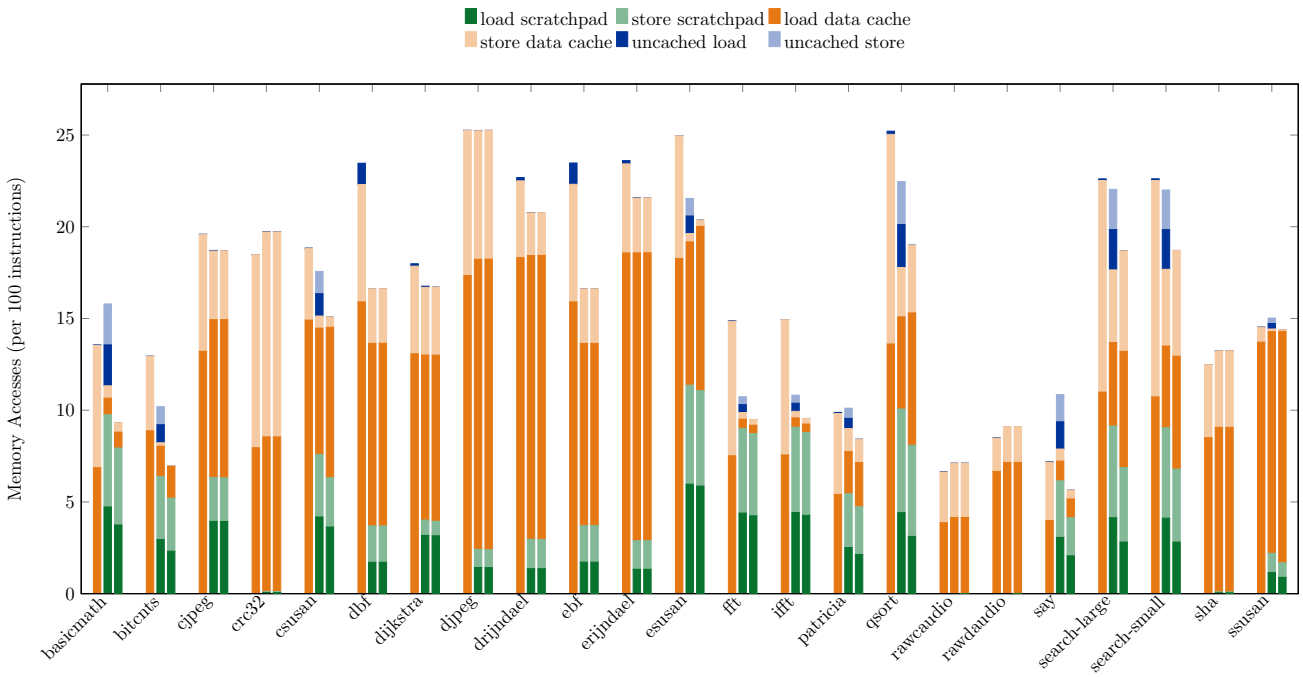

Figure 6: Memory accesses by memory type per 100 instructions (within each group: no stack cache, stack cache of 256 bytes, and stack cache of 2 KB).

Table 1: Statistics from static WCET analysis of the Debie1 benchmark (2 KB stack cache)

Problem	Operations		Worst-case		WCET no I$		WCET w. I$	
	Res/Free	Ensures	Spills	Fills	Original	SWSC	Original	SWSC
Debie1-1	1	0	0	0	778	746	2,373	2,559
Debie1-4a	10	20	0	0	5,204	5,208	6,660	6,826
Debie1-4b	7	20	0	0	619	565	1,443	1,869
Debie1-4c	3	4	0	0	483	459	1,050	1,188
Debie1-4d	4	4	0	0	535	523	1,213	1,554
Debie1-6a	16	63	0	0	13,187	13,254	22,770	25,538
Debie1-6c	16	63	0	0	27,121	24,433	43,186	42,249
Debie1-6d	16	63	0	0	15,066	14,937	27,726	29,653

problem. The registers that hold `sc_top` and `m_top` are being set to their initial value during C runtime initialization, which cannot be fully analyzed, before execution enters the `main` function. Knowing the semantics of the reserved registers, we can solve this problem by providing an annotation to the WCET analysis that initializes the stack cache pointers right before entering `main`. In our concrete case, using the aiT analyzer, the annotation for this purpose is:

```
ais2 {
  routine "main" {
    enter with:
      reg("r19") = 0x5000000,   # sc_top
      reg("r20") = 0x5000000;   # m_top
  }
}
```

No further annotations are required for aiT to derive tight bounds on the loops for all `reserve` and `ensure` invocations. aiT even proves that no stack cache spills or fills occur in any of the benchmarks for a stack cache size of 2 KB, as shown in Table 1. This is expected since Debie1 does not exhibit a deep call stack. Table 1 shows the static number of *reserve-free* pairs ('Res/Free'), as well as 'Ensure' operations in the code under analysis.

Compared to the 'Original' WCET bound, where the stack is backed by a data cache, the software stack cache ('SWSC') can yield a lower WCET bound due to fewer cache misses or improved analysis precision. This is represented by the columns under 'WCET', which assume a perfect instruction cache (i.e., every fetch is a cache hit). The impact of reserving two registers for the software stack cache is included in the WCET results. In general, we see in Table 1 that WCET bounds tend to improve and that problem Debie-6c benefits most.

When the worst-case latencies from an instruction cache are included (see 'WCET w. I$' columns), the stack cache improvement in our benchmarks is mostly outweighed by the impact of the increased code size on the instruction cache. For benchmark problem Debie-6c though, the result is a net benefit.

5.4 Discussion

From the WCET analysis we can see that the number of

executed instructions increases. For the WCET this increase in executed instructions offsets the improvement we gain with the stack cache in most cases. The reason for this increase is mainly caused by the address translation, which we perform for each stack load or store access. We consider optimizing this address translation code as future work.

The main issue in the address translation is that for each access the modulo operation has to be performed. If a stack frame is guaranteed not to cross the boundary of the SPM, a local stack pointer can be computed just once at function entry and used for all stack accesses. To enable this optimization, one option is to reuse the static analysis results about stack addresses during program compilation and insert the more efficient code when the stack frame is known to not cross the SPM address boundary.

Another option is to allocate stack frames in multiples of fixed block sizes, e.g., multiple of 8 words. In that case it is guaranteed that the first block address is within the SPM address range and the slots allocated in this first block need no modulo operation. The most accessed stack slots shall then be paced in the first block.

6. CONCLUSION

In this paper we present a mechanism to dynamically allocate stack frames in an on-chip scratchpad memory (SPM). This software stack cache is updated on function entry and exit to ensure that the stack frame of the active function is in the SPM. Using the SPM for stack data guarantees single cycle execution of loads and stores, equivalent to cache hits. This property can simplify worst-case execution time (WCET) analysis and can also reduce the WCET bound. To find an upper bound of the number of SPM memory exchanges on function entry and return, a WCET analysis tool needs to track the stack pointer values. We explored WCET analysis with the industry-standard tool aiT. Its integrated value analysis was able to: (1) detect that the stack accesses are in the SPM address range and are therefore counted as single cycle accesses and (2) track the two stack-related pointers and provide a tight bound on the loops for the stack spill and fill traffic.

Acknowledgment

The authors would like to thank Wolfgang Puffitsch for his helpful advice regarding the Patmos implementation. 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) and partially funded by the Danish Council for Independent Research | Technology and Production Sciences under the project RTEMP, contract no. 12-127600. Alexander Jordan was supported by the COST Action IC1202: Timing Analysis On Code-Level (TACLe).

Source Access

The implementation of the software stack cache is open-source and available from
https://github.com/t-crest/patmos-llvm
and
https://github.com/t-crest/patmos-newlib
in the branch `swscache`.

7. 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] Ke Bai, A. Shrivastava, and S. Kudchadker. Stack data management for limited local memory (LLM) multi-core processors. In *Proc. of the International Conference on Application-Specific Systems, Architectures and Processors*, ASAP '11, pages 231–234. IEEE, 2011.

[3] Jean-Francois Deverge and Isabelle Puaut. Wcet-directed dynamic scratchpad memory allocation of data. In *ECRTS '07: Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 179–190, Washington, DC, USA, 2007. IEEE Computer Society.

[4] 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, pages 65–74, New York, NY, USA, 2007. ACM.

[5] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The mälardalen wcet benchmarks - past, present and future. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, July 2010.

[6] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. of the Workshop on Workload Characterization*, WWC '01, 2001.

[7] Reinhold Heckmann and Christian Ferdinand. Worst-case execution time prediction by static program analysis. Technical report, AbsInt Angewandte Informatik GmbH. [Online, last accessed November 2013].

[8] Niklas Holsti, Thomas Långbacka, and Sami Saarinen. Using a worst-case execution-time tool for real-time verification of the DEBIE software. In *Proc. of the Data Systems in Aerospace Conference*, page 307. ESA, 2000.

[9] 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.

[10] A. Kannan, A. Shrivastava, A. Pabalkar, and Jong eun Lee. A software solution for dynamic stack management on scratch pad memory. In *Proc. of the Asia and South Pacific Design Automation Conference*, ASP-DAC '09, pages 612–617. IEEE, 2009.

[11] Sungjun Kim. Using scratchpad memory for stack data in hard real-time embedded systems, 2011.

[12] Hsien-Hsin S. Lee, Mikhail Smelyanskiy, Gary S. Tyson, and Chris J. Newburn. Stack value file: Custom microarchitecture for the stack. In *Proc. of the International Symposium on High-Performance Computer Architecture*, HPCA '01, pages 5–14. IEEE, 2001.

[13] Jing Lu, Ke Bai, and A. Shrivastava. SSDM: Smart

stack data management for software managed multicores (smms). In *Proc. of the Design Automation Conference*, DAC '13, pages 1–8. IEEE, 2013.

[14] Soyoung Park, Hae-woo Park, and Soonhoi Ha. A novel technique to use scratch-pad memory for stack management. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '07, pages 1478–1483, San Jose, CA, USA, 2007. EDA Consortium.

[15] Peter Puschner, Daniel Prokesch, Benedikt Huber, Jens Knoop, Stefan Hepp, and Gernot 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)*, pages 33–40, 2013.

[16] Martin 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.

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

[18] Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, Stefan Hepp, Benedikt Huber, Alexander Jordan, Evangelia Kasapaki, Jens Knoop, Yonghui Li, Daniel Prokesch, Wolfgang Puffitsch, Peter Puschner, André Rocha, Cláudio Silva, Jens Sparsø, and Alessandro Tocchi. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471, 2015.

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

[20] Martin Schoeberl and Carsten Nielsen. A stack cache for real-time systems. In *Proceedings of the 18th IEEE Symposium on Real-time Distributed Computing (ISORC 2016)*, York, United Kingdom, May 2016. IEEE.

[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] Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. WCET centric data allocation to scratchpad memory. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS)*, pages 223–232. IEEE Computer Society, 2005.

[23] Lars Wehmeyer and Peter Marwedel. Influence of memory hierarchies on predictability for time constrained embedded software. In *Proceedings of Design, Automation and Test in Europe (DATE2005).*, pages 600–605 Vol. 1, March 2005.

[24] J. Whitham, R.I. Davis, N.C. Audsley, S. Altmeyer, and C. Maiza. Investigation of scratchpad memory for preemptive multitasking. In *Real-Time Systems Symposium (RTSS), 2012 IEEE 33rd*, pages 3–13, Dec 2012.

[25] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution time problem – overview of methods and survey of tools. *Trans. on Embedded Computing Sys.*, 7(3):1–53, 2008.