2015 IEEE 18th International Symposium on Real-Time Distributed Computing Workshops

Stack Caching Using Split Data Caches

Carsten Nielsen and Martin Schoeberl
Department of Applied Mathematics and Computer Science
Technical University of Denmark
Email: s123161@student.dtu.dk, masca@imm.dtu.dk

***Abstract*—In most embedded and general purpose architectures, stack data and non-stack data is cached together, meaning that writing to or loading from the stack may expel non-stack data from the data cache. Manipulation of the stack has a different memory access pattern than that of non-stack data, showing higher temporal and spatial locality. We propose caching stack and non-stack data separately and develop four different stack caches that allow this separation without requiring compiler support. These are the simple, window, and prefilling with and without tag stack caches. The performance of the stack cache architectures was evaluated using the SimpleScalar toolset where the window and prefilling stack cache without tag resulted in an execution speedup of up to 3.5% for the MiBench benchmarks, executed on an out-of-order processor with the ARM instruction set.**

***Keywords*-cache memory; microprocessors; stack caching;**

I. Introduction

Typical programs use a stack for allocating local variables, passing parameters to procedures, and register spilling. In a memory hierarchy, items stored on the stack benefit greatly from the advantages of caches because memory accesses to stack data exhibit both strong temporal and spatial locality, as accesses are limited to relatively small stack frames. Performance problems occur when stack data, heap, and/or static data compete for the same space in the cache hierarchy. This type of problem is normally solved by increasing associativity, but this solution is costly in terms of power, area, and speed [1].

In this paper we examine methods of separately caching stack data in a so-called stack cache by exploiting the strong spatial and temporal locality observed in stack data memory accesses. The methods are designed with the requirement that no compiler support should be necessary for an implementation to be successful. We describe four different types of stack caches: (1) the simple stack cache, (2) the window cache, and the prefilling cache without (3) and (4) with tag bits.

The simple stack cache is a conventional data cache placed in parallel with the L1 data cache in the memory hierarchy, configured to handle all memory accesses directed towards the stack. The window cache is placed between the processor and the L1 data cache and handles only memory accesses in a small address range, corresponding to its size, offset from the stack pointer. The prefilling stack cache without tag is obtained by adding a prefilling and spilling strategy to the window cache and the prefilling stack cache with tag is created by adding a prefilling or spilling strategy to the simple stack cache. These caches all exclusively contain stack data, thereby freeing this from occupying the L1 data cache and disturbing the caching of more randomly accessed non-stack data.

We evaluate their performance impact on an out-of-order processor with the ARM instruction set by simulation in the SimpleScalar [2] framework, with the criterion that they should improve average execution time.

The contributions of this paper are: describing and evaluating architectures of four types of stack caches suitable for embedded and general purpose computing, of which two are successful in reducing execution time.

This paper is organized in 7 sections: The following section presents related work. Section III describes the four different stack cache types. Section IV evaluates the performance of the stack caches with embedded benchmarks. Section V discusses further research opportunities. Section VI concludes the paper.

II. Related Work

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

Lu, Bai, and Shrivastava implemented scratch-pad memory (SPM) stack caching for software managed multicore (SMM) architectures [3]. 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

1555-0885/15 $31.00 © 2015 IEEE
DOI 10.1109/ISORCW.2015.59

15% and 1% compared to circular stack management. A similar approach optimized for recursive functions has also been explored [4].

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 [5]. 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 SPM acting as a cache that slides up and down in memory, following the stack pointer [6]. 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 to a normal cache architecture, configured to only cache stack data, and showed that the SPM stack cache was both faster and more energy efficient.

Stack caching has also been implemented on stack machine architectures such as hardware implementations of the Java virtual machine (JVM) [7] [8] [9]. In the JVM operands are saved on the top of the stack while local variables are stored deeper down. Schoeberl showed that the operands can be cached in a two-entry register file containing the two topmost elements of the stack, while single read port on-chip memory contained the lower parts of the stack [10]. The result was more efficient than approaches caching only in a small register file [11] and approaches using only three-port on-chip memory for caching [12] in both area and speed.

For real-time system it has been proposed to split the data cach [13], [14]. 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 [15] implemented a stack cache for the Patmos processor [16] which requires compiler support. Their scheme uses three additional hardware instructions: reserve, free, and ensure. These instructions are used by the compiler to make sure that the stack frame belonging to a function is cached. This allows entire stack frames to be kept in the stack cache instead of entirely in main memory 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 [17].

Lee, Smelyanskiy, Newburn, and Tyson developed the Stack Value File (SVF) microarchitecture [18]. 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 dirtly 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 [19]. 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.

III. Stack Caching

Stack caching aims to exploit access patterns to the stack, a data structure that is typically used to hold local variables, return addresses, arguments for function calls, and spilled registers. Most higher-level languages use a stack invisible to the programmer for these purposes, although some languages, such as Forth, are based on the model of a stack machine and require the programmer to manage it. The stack occupies a contiguous area in memory starting at some predefined address. For historical reasons, the stack usually grows downwards. Today, this convention is completely arbitrary and with virtual memory there is nothing inhibiting growing the stack upwards.

When used for the purposes described above, the stack is organized into stack frames. A stack frame is an area on the stack containing variables relevant to a single function or routine. When new functions are called, a new stack frame is created on top of the stack. Since stack frames may not inhabit the same memory location on every invocation of a

name	position	window type	prefilling
simple	parallel	no	no
window	elevated	yes	no
prefill w/o tag	elevated	yes	yes
prefill w/ tag	parallel	no	yes

Table I: Type overview of the four stack caches examined in this paper.

function, variables are addressed relative to the stack pointer, which always points to the topmost element on the stack.

A frame pointer, in addition to a stack pointer, supports allocation of dynamic data structures on the stack. The frame pointer points to the first element of the topmost stack frame and is fixed for the function. The stack pointer changes as data is allocated dynamically. In that case the frame pointer is used to address the local variables, and all other stack frame content, with a constant offset. Fig. 1 shows a possible snapshot of a stack, holding subroutine arguments, return address, the previous frame pointer as well as saved registers and local variables.

Stack usage varies between architectures, depending on the number of available general purpose and parameter passing registers. For example, architectures like x86 need to use the stack more than a MIPS-like architecture, in order to spill registers and pass parameters to functions.

A. Stack Cache Types

We can classify a stack cache with three parameters: (1) its position within the memory hierarchy, (2) whether or not it employs a memory window, and (3) what prefill strategy is used. The hierarchy position determines whether the stack cache is *parallel* to the L1 data cache or *elevated* (between the processor and the original L1 data cache). A parallel stack cache sources misses from the L2 data cache, while an elevated stack cache uses the L1 data cache as shown in Fig. 2. A window type cache will only be accessed when the requested memory address lies between two pointers named *bos* and *tos*, meaning bottom and top of stack respectively. Therefore, $bos - tos$ determines the size of the memory area cached in the stack cache at any point in time.

A non-window cache accepts all memory accesses directed to the stack. This is done by redirecting all memory accesses between the stack pointer and the starting address of the stack, which is determined at compile time.

Beyond this the stack cache may also employ prefilling/spilling according to a number of strategies of varying complexity. This paper deals with four stack caches, a simple stack cache, a window stack cache, a prefilling stack cache without a tag, and a prefilling stack cache with a tag.

An overview of the attributes connected to the described stack cache types is listed in Table I

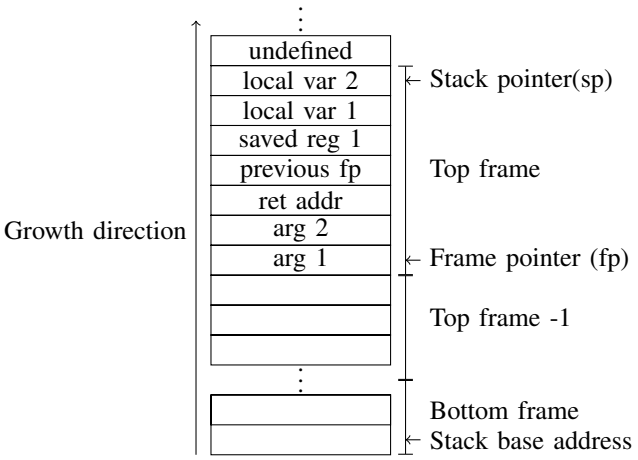

Figure 1: The stack in operation, showing a possible organization of return addresses, saved registers, and local variables.

B. Simple Stack Cache

Our base assumptions of the memory hierarchy are core-local level 1 (L1) caches for instruction and data with a single cycle access time. These caches are backed up by a larger (and slower) level 2 (L2) cache. That L2 cache is shared on a multicore processor. To this configuration we add a cache for stack allocated data.

The simplest possible version of such a cache, is a standard cache placed in parallel with the L1 data cache and configured to handle all memory accesses directed to the stack, ensuring that all loads and stores to the stack pass through it. This eliminates conflicts between stack data and non-stack data in the L1 data cache. Since the stack is a contiguous block of memory, it makes sense to choose a direct mapped cache for the stack cache.

Adding the simple stack cache will show an increased hit rate for stack data as well as overall hitrate for stack and normal data combined, provided that the stack cache is large enough to contain most stack frames. When the cache becomes too small and items are written back to the L2 data cache, performance will suffer when reloading these.

C. Window Type Stack Cache

To avoid the miss penalty incurred by the simple stack cache, the cache can be placed such that it fetches from the L1 data cache instead of the L2. Additionally, address range boundaries can be imposed on the stack cache such that only memory accesses within a specific range are handled by it. This eliminates the problem of stack frames being too large for the stack cache, since any reference to an item that would not fit in the stack cache, simply bypasses it and is handled by the L1 data cache.

These boundaries can be enforced using two pointers, bottom-of-stack *bos* and top-of-stack *tos*. In the simple

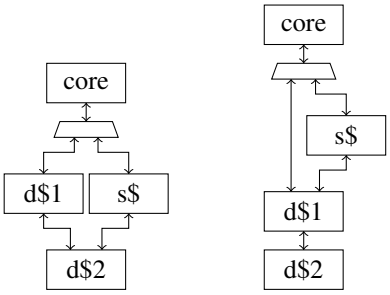

Figure 2: The two different hierarchal placement options for the stack cache. The stack cache is either parallel to the data cache, meaning misses are served by the L2 cache in the hierarchy, or elevated above the data cache, meaning misses are served by the L1 data cache.

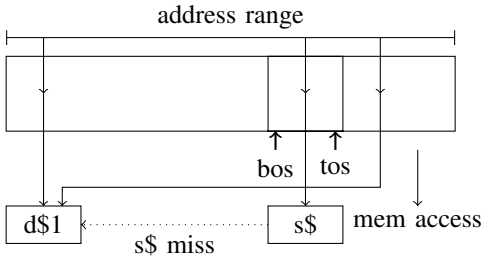

Figure 3: Window type stack cache. Only memory accesses between the bos and tos pointers are directed to the stack cache. The stack cache fetches misses from the L1 data cache. Arrows signify memory accesses and bolded arrows signify pointers.

window type stack cache, *tos* is locked to the stack pointer and *bos* is below *tos* by the size of the stack cache. To avoid coherency issues, the bypassing must be to the same cache that the window type stack cache fetches misses from. For a window type stack cache that is parallel to the L1 data cache, all accesses to the stack, that do not fall within the range between *bos* and *tos*, would need to bypass the stack cache and go directly to the L2 data cache. This would have a severe impact on performance. Fig. 3 shows the window type stack cache.

The window type stack cache can be thought of as a ring-buffer, overwriting the last elements when new data is written at the address pointed to by the stack pointer.

D. Tagless Prefilling Stack Cache

By adding a prefill/spill mechanism to the window stack cache we can remove the need for tag memory. This mechanism controls the *bos* and *tos* pointers so that they are not fixed to the stack pointer, but free to move around according to a spill/fill policy. This policy must insure that only addresses between *bos* and *tos* are represented in the stack cache in order for it to be tagless.

The simplest possible prefill/spill mechanism acts as follows: When an item is pushed to the stack, the bottom item in the stack cache is written back to the L1 data cache automatically and *bos* is decremented by the block size of the stack cache, while *tos* follows the value of the stack pointer. When an item is popped from the stack, *tos* will again follow the stack pointer, but *bos* must now be incremented and the new memory item pointed to by *bos* must be loaded from the L1 data cache.

If more than one item is popped from the stack in a single operation, the prefill mechanism will attempt to fill all the remaining slots in the stack cache. This potentially hurts performance as items at the bottom of the stack cache may be reloaded and written back to the L1 data cache several times without being used, as the stack pointer moves up and down near the top of the memory range contained in the stack cache. We therefore move away from the simple policy such that *bos* is only changed when the stack pointer has moved upwards by a significant amount of addresses, two block sizes in our implementation. Thereby limiting the amount of necessary loads from the L1 data cache.

When large stack frames are popped or initialized, the *tos* and *bos* pointers will need to traverse a large number of addresses, possibly greater than the size of the stack cache. In this case, we allow the cache to completely flush itself and jump to the new location of the stack pointer. This is done to prevent the stack cache from loading all stack items between *tos*, or *bos*, and the stack pointer at the time of the large change in the stack pointer value.

Additionally, when items are pushed to the stack, valid data located at the bottom of the address range, covered by the stack cache, must be written back to the L1 data cache. We can avoid this by setting a target fill percentage of less than one hundred percent. The prefill/spill mechanism will then spill cache lines, moving *bos* towards *tos* if the current fill percentage is too high. Inversely, it will fill cache lines, moving *bos* away from *tos*, if the current fill percentage is too low. This will free space in the stack cache where new stack data can be written without the need for writing back old data to the L1 data cache. We use a fill percentage of 50%. The described prefill/spill policy is the one that will be evaluated in this paper. However, we will continue to elaborate on optimizations that may increase performance.

In a conventional cache, data items located between the heap and stack are written back through the cache hierarchy, because the data cache cannot distinguish between stack and non-stack data. This data is by definition invalid and does not need to be written back. The stack cache allows us to make the distinction between stack and non-stack data, and thereby avoid the writeback of invalid data when items are popped from the stack. Likewise, when the processor issues a write to an address in a new cache line at the top of the stack, this cache line does not need to be fetched from a

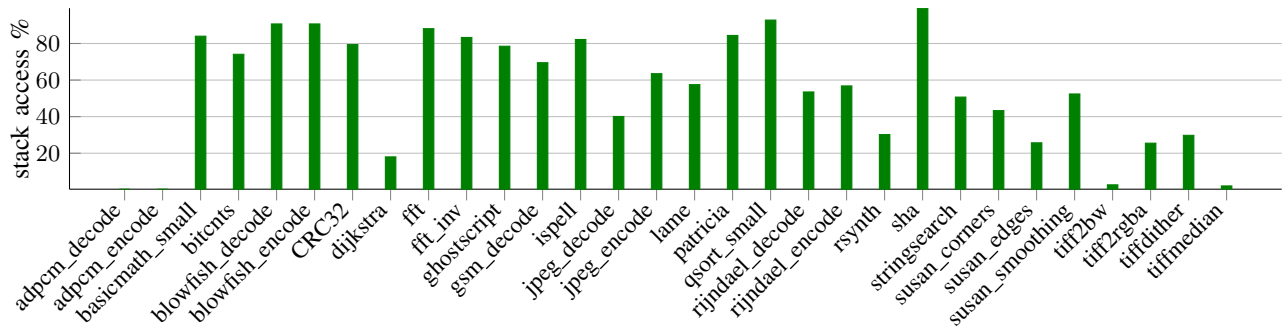

Figure 4: Percentage of memory accesses, excluding instruction fetch, directed towards the stack for each benchmark. The benchmark vary considerable in their stack use, some producing hardly any stack accesses and some more than 80 percent.

lower hierarchy level. The data can simply be written into the stack cache.

This write might move the *tos* pointer. If the stack cache becomes full it will also move *bos*, and some cache lines will need to be written into the next hierarchy level. In the other case, where *tos* moves in the other direction, the line will not be written back for the reasons explained above. Thus, some data may only ever inhabit the stack cache, further reducing the pressure on the L1 data cache. We do not implement these optimizations as we have found that stack data is sometimes written to addresses not in the current stack frame when executing the MiBench benchmarks on the SimpleScalar ARM out-of-order processor.

When operating under optimal conditions, the prefilling/spilling cache has an adequate amount of non-occupied space that allows the top of stack pointer to track the stack pointer unhindered. When new items are written to the stack near the stack pointer they will then be placed in the stack cache. When the stack pointer rises below this point on a function return, the data will be invalidated and overwritten without write-back to the L1 cache. It is unfortunately not always possible to operate under optimal conditions. When stack frames become too large for the cache, the stack pointer will grow far beyond the *bos* pointer. This causes the stack cache to flush itself and reset the bottom and top of stack pointers to the stack pointer. Afterwards it will begin filling towards the specified fill percentage. The cache controller operates in parallel with the core and only issues spills and fills to/from the L1 data cache on cycles where the memory bus between the L1 and L2 data cache is unused in an attempt to minimize interference with L1 data cache misses that would stall the core.

The described policy has the added benefit of not requiring tag memory in the cache as the memory position of entries can be inferred from the *bos* and *tos* pointers.

E. Prefilling Stack Cache with Tag Memory

We can also configure the prefilling cache to be parallel to the L1 data cache instead of elevated above it by changing the previously described tagless prefilling cache such that it sources misses from the L2 data cache. The access and fill policy must also be changed to prevent coherency issues that may arise if the same block is present in both the L1 data cache and a parallel window stack cache. Writing to the block in the parallel window stack cache before moving the window would invalidate the block in the L1 data cache. Instead of only accepting accesses between *bos* and *tos*, the prefilling stack cache with tag memory therefore accepts all accesses to the stack, like the simple stack cache. However, it still maintains *bos* and *tos* for prefilling purposes. Unlike the tagless design, this cache does not require flushing of the cache before jumping to a far-away stack pointer since it does not rely on the *bos* and *tos* pointer for determining tag bits.

IV. Evaluation

We have implemented the stack caches on the out-of-order architecture of SimpleScalar using the SimpleScalar ARM ISA and in this section we will examine their effect on execution time.

The benchmarks used for evaluation are all part of the MiBench benchmark suite [20]. These benchmarks target embedded systems and have a wide range of instruction distributions. The amount of memory operations varies between 10-65%, ALU operations vary between 25-80% and branch instructions vary between 6-20%. Program sizes vary with text segment sizes between 150-300 KB for the vast majority of the benchmarks, while data segments vary between a few kilobytes and 500 KB. Only a few benchmarks lie outside these numbers, most notably the lame benchmark, having a text segment size of 1.6 MB and a data segment size of 3.9 MB.

All simulations shown are performed with a 4 KB L1 data cache with 32 byte blocks and 4 way associativity unless explicitly stated otherwise. The 4 KB size was chosen to mimic the size of L1 data caches on small embedded systems with limited resources. This is fitting since the MiBench suite targets these systems.

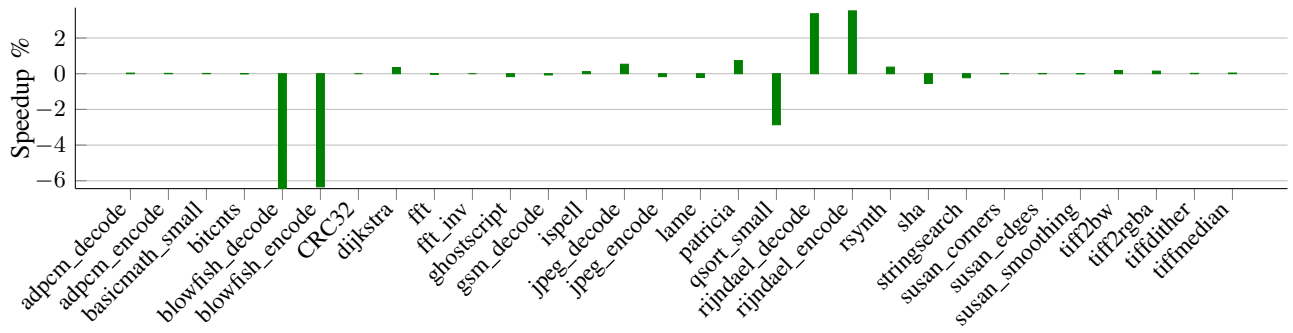

Figure 5: Speedup obtained by adding a 1 KB parallel simple stack cache compared to having no stack cache. Some benchmarks show improved execution time, but this is outweighed by the massive increase in execution time for the blowfish benchmarks. The parallel prefilling cache shows approximately the same result and is therefore not shown.

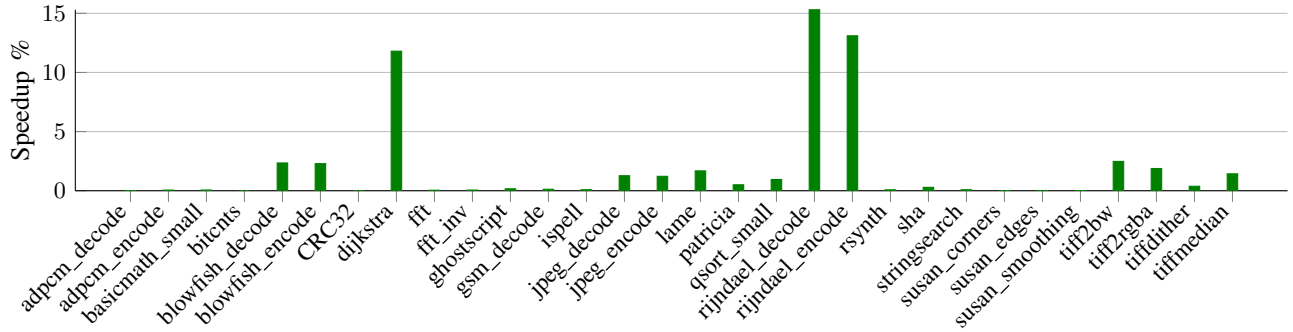

Figure 6: Speedup of an 8 KB L1 data cache compared to speedup obtained by adding a 4 KB simple stack cache to a 4 KB L1 data cache. Having a stack cache size equal to the data cache is inefficient compared to doubling the data cache size.

When the stack and data cache are in a parallel configuration, they share a bus to the L2 data cache which has a size of 256 KB, associativity level 4, block size of 64 B and access time of 6 cycles for all measurements. In the case of an elevated configuration, the stack cache has a 1 cycle access time to the L1 data cache which is the same as the latency from the core to the L1 data cache. The stack cache access time is 1 cycle in both the elevated and parallel configurations. The main memory uses burst transfer, taking 32 cycles to transfer one L2 cache block.

For the stack cache to be effective, there must be some competition between stack data and other data in the case where the system only has a data cache. This is most likely to happen if a program has a balanced mix of stack and non-stack memory accesses. It is reasonable to assume that each stack access miss corresponds to a previous eviction of non-stack data caused by a stack access and therefore also to a data miss that could have been avoided if stack data was cached separately.

Fig. 4 shows the stack usage as a percentage of the total number of data memory accesses for a selection of benchmarks. Some benchmarks show a surprisingly high ratio of stack accesses to non-stack accesses, which can happen for two reasons. Either, the benchmarks makes many references to a narrow range in memory, reusing many of the same data items. For example most of the execution time could be spent in a tight loop manipulating mostly control variables. Otherwise, the amount of stack data items relative to the amount of non-stack data items is genuinely large. Stack caching is more effective in the latter case, as stack data would not take up much space in the L1 data cache in the prior case. Most programs lie between the two extremes in their stack access pattern. Therefore, the percentage of memory accesses directed towards the stack is not a good predictor of stack cache performance.

A. *Parallel Stack Caches*

Speedup obtained by adding a 1 KB simple stack cache is show in Fig. 5. With a size of 1024 bytes, the cache is not large enough to contain a majority of the stack data which causes many conflict misses, reducing execution time for some of the benchmarks. Since it is in a parallel configuration, the L1 data cache cannot hold the extra data items and misses in the stack cache are forced to go to the L2 data cache, thereby incurring a large miss penalty.

The parallel configuration is therefore a bad choice unless the cache size is increased further. However, this will make

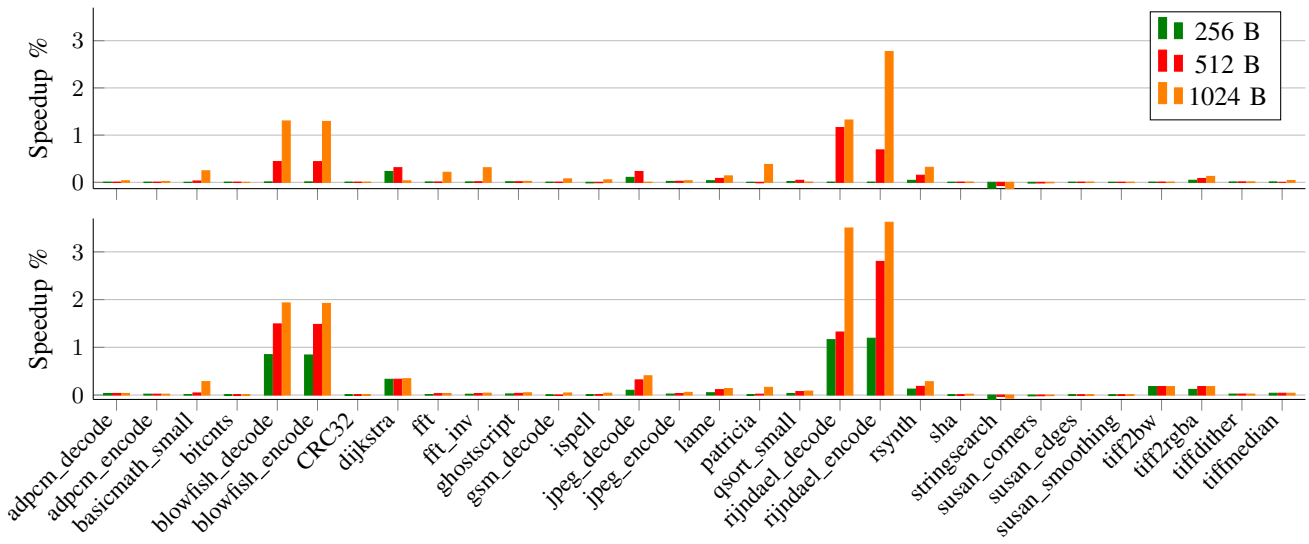

Figure 7: Speedup obtained by adding an elevated prefilling stack cache (top) and an elevated window stack cache (bottom) with the size specified in the legend.

the stack cache closer in size to the L1 data cache and at that point it is more effective to just enlarge the L1 data cache. If the two caches are of equal size, removing the stack cache and doubling the size of the data cache is vastly superior to employing the stack cache because this will also affect other than just stack data. This is shown in Fig. 6 where an 8 KB L1 data cache is compared to a 4 KB L1 data cache with a 4 KB simple stack cache.

B. Elevated Stack Caches

The two elevated stack caches are more likely to produce positive results since they do not have the same miss penalty as the parallel stack caches. Fig. 7 shows speedup over a processor with no stack cache for both the basic window cache and tagless prefilling cache with sizes of 256 B, 512 B and 1024 KB. The blowfish and rijndael benchmarks both show significantly decreased execution times of up to 3.5% using the 1 KB window stack cache, while the rest of the benchmarks are somewhat unaffected. Only the stringsearch benchmark shows a very small increase in execution time.

The small responses to the addition of a stack cache can be explained by the small size of many of the MiBench benchmarks. They are suited for embedded applications, and emphasize a somewhat low memory pressure.

The window type stack cache performs markedly better than the tagless prefilling cache. This is likely due to a suboptimal fill/spill strategy as well as the lack of the write back optimization. The implemented prefill/spill strategy aiming for a 50% filled cache jams the memory bus between the L1 and L2 data cache enough to cause some stalls and an increase in execution time. A more elaborate prefill/spill scheme, perhaps tracking the history of the stack or frame pointer, might provide better performance for stack access patterns close to the bottom of the call tree where leaf procedures may be called many times in a row, in addition to solving the problem of the jammed L1 to L2 memory bus.

Elevated prefilling and window stack caches are thus positive additions to a either an embedded or general purpose architecture, even for small sizes (1024 B), decreasing execution time by up to 3.5%.

V. Future Work

We have tested different stack caches on an out-of-order processor implementing the ARM ISA. It would be interesting to see how the stack cache performs on the x86 ISA where there are fewer general purpose registers, and on the MIPS ISA where there are more general purpose registers. We predict that the stack cache will be more effective on x86 since the larger stack frames will take up more space if placed only in the L1 data cache. Also of interest is the performance of stack caches on processors that do not employ out-of-order execution, such as Patmos [16].

While the goal of this paper was to explore stack cache architectures that do not need compiler support, it could also be interesting to examine what could be done with compiler support in the general purpose domain since the approach yielded very positive results for Abbaspour et al. [15].

The described stack caches also need to be implemented in actual hardware to provide real data on the area and power cost, although the cache controller is rather small for simple spill/fill strategies.

VI. Conclusion

Typical programs use a stack for storing the return address, argument passing, local variable storage, and register spilling. Access patterns to the stack data structure exhibit both high spatial and temporal locality. We have implemented four so-called stack caches that attempt to exploit these patterns without requiring explicit compiler support: the simple, window, prefilling with tag and prefilling without tag stack caches. These caches do not require compiler support and for the set of benchmarks used in evaluation, adding the window and tagless prefilling stack cache results in an execution time reduction of up to 3.5%, for a cache size of just 1 KB.

Acknowledgment

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.

References

[1] R. K. Megalingam, K. Deepu, I. P. Joseph, and V. Vikram, "Phased set associative cache design for reduced power consumption," in *2009 2nd IEEE International Conference on Computer Science and Information Technology*. IEEE, 2009, pp. 551–556.

[9] M. Zabel, T. B. Preusser, P. Reichel, and R. G. Spallek, "Secure, real-time and multi-threaded general-purpose embedded Java microarchitecture," *DSD 2007: 10Th Euromicro Conference on Digital System Design Architectures, Methods and Tools, Proceedings*, pp. 59 – 62, 2007.

[2] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: an infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, pp. 59–67, 2002.

[3] J. Lu, K. Bai, and S. Aviral, "SSDM: Smart Stack Data Management for software managed multicores (SMMs)," *DAC*, pp. 1 – 8, 2013.

[4] A. Dominguez, N. Nguyen, and R. 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*. New York, New York, USA: ACM Press, 2007, p. 65.

[5] A. Kannan, A. Shrivastava, A. Pabalkar, and J.-e. Lee, "A software solution for dynamic stack management on scratch pad memory," in *2009 Asia and South Pacific Design Automation Conference*. IEEE, Jan. 2009, pp. 612–617.

[6] S. Park, H.-w. Park, and S. Ha, "A Novel Technique to Use Scratch-pad Memory for Stack Management," in *2007 Design, Automation & Test in Europe Conference & Exhibition*. IEEE, Apr. 2007, pp. 1–6.

[7] M. Schoeberl, "A Java processor architecture for embedded real-time systems," *Journal of Systems Architecture*, vol. 54, no. 1-2, pp. 265–286, Jan. 2008.

[8] 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 '07*. New York, New York, USA: ACM Press, 2007, p. 230.

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

[11] D. 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*, pp. 53 – 59, 2001.

[12] S. Ito, L. Carro, and R. Jacobi, "Making Java work for microcontroller applications," *IEEE Design & Test of Computers*, vol. 18, no. 5, pp. 100–110, 2001.

[13] M. Schoeberl, "Time-predictable cache organization," in *Proceedings of the First International Workshop on Software Technologies for Future Dependable Distributed Systems (STFSSD 2009)*. Tokyo, Japan: IEEE Computer Society, March 2009, pp. 11–16.

[14] M. Schoeberl, W. Puffitsch, and B. Huber, "Towards time-predictable data caches for chip-multiprocessors," in *Proceedings of the Seventh IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2009)*, ser. LNCS, no. 5860. Springer, November 2009, pp. 180–191.

[15] S. Abbaspour, F. Brandner, and M. Schoeberl, "A time-predictable stack cache," in *Proceedings of the 9th Workshop on Software Technologies for Embedded and Ubiquitous Systems*, 2013.

[16] 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*, 2011, pp. 11–20.

[17] A. Jordan, F. Brandner, and M. Schoeberl, "Static analysis of worst-case stack cache behavior," in *Proceedings of the 21st International Conference on Real-Time Networks and Systems (RTNS 2013)*. New York, NY, USA: ACM, 2013, pp. 55–64.

[18] H. Lee, M. Smelyanskiy, C. Newburn, and G. Tyson, "Stack value file: Custom microarchitecture for the stack," *HPCA: Seventh International Symposium on High-Performance Computing Architecture, Proceedings*, pp. 5 – 14, 2001.

[19] L. E. Olson, Y. Eckert, S. Manne, and M. D. Hill, "Revisiting stack caches for energy efficiency."

[20] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*. IEEE, 2001, pp. 3–14.