Alignment of Memory Transfers of a Time-Predictable Stack Cache

Abbaspourseyedi, Sahar; Brandner, Florian

Published in:
Proceedings of the 8th Junior Researcher Workshop on Real-Time Computing (JRWRTC 2014)

Publication date:
2014

Alignment of Memory Transfers of a Time-Predictable Stack Cache

Sahar Abbaspour
Dep. of Applied Math. and Computer Science
Technical University of Denmark
sabb@dtu.dk

Florian Brandner
Computer Science and System Eng. Dep.
ENSTA ParisTech
florian.brandner@ensta-paristech.fr

ABSTRACT

Modern computer architectures use features which often complicate the WCET analysis of real-time software. Alternative time-predictable designs, and in particular caches, thus are gaining more and more interest. A recently proposed stack cache, for instance, avoids the need for the analysis of complex cache states. Instead, only the occupancy level of the cache has to be determined.

The memory transfers generated by the standard stack cache are not generally aligned. These unaligned accesses risk to introduce complexity to the otherwise simple WCET analysis. In this work, we investigate three different approaches to handle the alignment problem in the stack cache: (1) unaligned transfers, (2) alignment through compiler-generated padding, (3) a novel hardware extension ensuring the alignment of all transfers. Simulation results show that our hardware extension offers a good compromise between average-case performance and analysis complexity.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*

General Terms

Algorithms, Measurement, Performance

Keywords

Block-Aligned Stack Cache, Alignment, Real-Time Systems

1. INTRODUCTION

In order to meet the timing constraints in systems with hard deadlines, thez worst-case execution time (WCET) of real-time software needs to be bounded. Many features of modern processor architectures, such as caches and branch predictors, improve the average performance, but have an adverse effect on WCET analysis. Time-predictable computer architectures thus propose alternative designs that are easier to analyze, focusing in particular on analyzable cache and memory designs [14, 9]. One such alternative cache design is the *stack cache* [5, 10], i.e., a cache dedicated to stack data. The stack cache is a complement to a regular data cache and thus reduces the number of accesses through the data cache. This promises improved analysis precision, since unknown access addresses can no longer interfere with stack accesses (and vice versa). Secondly, the stack cache design is simple and thus easy to analyze [5]. The WCET analysis of traditional caches requires precise knowledge about the addresses of accesses [14] and has to take the *complex* replacement policy into account. The analysis of the stack cache on the other hand is much easier and amounts to a simple analysis of the cache's fill level (*occupancy*) [5].

The original stack cache proposes three instructions to reserve space on the stack (`sres`), free space (`sfree`), and to ensure the availability of stack data in the cache (`sens`). The reserve and ensure instructions may cause memory transfers between the stack cache and main memory and thus are relevant for WCET analysis. Since these stack cache control instructions operate on words, the start addresses of memory transfers are not guaranteed to be aligned to the memory controller's burst size. Therefore, unaligned transfers incur a performance penalty that needs to be analyzed. This is in contrast to traditional caches where the cache line size is typically aligned with the burst size. This risks anew to introduce complexity to the otherwise simple WCET analysis of the stack cache. We thus compare three approaches to handle this alignment problem for the stack cache: (1) a stack cache initiating unaligned transfer, (2) compiler-generated padding to align all stack cache allocations (and thus all transfers), and (3) a novel hardware extension that guarantees that all stack cache transfers are block-aligned.

For our hardware extension a burst-sized block of the stack cache is used as an *alignment* buffer, which can be used to perform block-aligned memory transfers. The downside of this approach is that the effective stack cache size is reduced by one block. On the other hand, the block-aligned transfers simplify WCET analysis. In addition, this allows us to perform allocations at word granularity, which improves the cache's utilization. The hardware overhead of our approach is minimal: the implementation of `sres` and `sens` is simplified, while `sfree` requires some minor extensions. The free instructions may need to initiate memory transfers to preserve the alignment of the stack cache content.

Section 2 introduces the stack cache, followed by a discussion of related work. In Section 4, we explain the block-aligned stack cache and its impact on static analysis. We finally present the results from our experiments in Section 5.

2. BACKGROUND

The original stack cache [1] is an on-chip memory implemented as a special ring buffer utilizing two pointers: stack top (`ST`) and memory top (`MT`). The `ST` points to the stack data either stored in the cache or main memory. The `MT` points to the stack data in main memory. The difference

$\texttt{MT} - \texttt{ST}$ represents the occupied space in the stack cache. Since this value cannot exceed the total size of the stack cache ($|SC|$), $0 \leq \texttt{MT} - \texttt{ST} \leq |SC|$ always holds.

Three control instructions manipulate the cache (more details are available elsewhere [1]):

- **`sres` x:** Subtract x from `ST`. If this violates the equation from above, i.e., the stack cache size is exceeded, a *spill* is initiated, which lowers `MT` until the invariant is satisfied again.
- **`sens` x:** Ensure that the occupancy is larger than x. If this is not the case, a *fill* is initiated, which increments `MT` accordingly so that $\texttt{MT} - \texttt{ST} \geq x$.
- **`sfree` x:** Add x to `ST`. If this results in a violation of the invariant, `MT` is incremented accordingly. Memory is not accessed.

The analysis of the stack cache [5] is based on the observation that the concrete values of `ST` and `MT` are not relevant for the worst-case behavior. Instead the focus is on determining the *occupancy*, i.e., the value $\texttt{MT} - \texttt{ST}$. The impact of function calls is taken to account by the function's *displacement*, i.e. the number of cache blocks spilled to main memory during the function call. The analysis then consists of the following phases: (1) an analysis of the minimum/maximum displacement for call sites on the call graph, (2) a context-independent, function-local data-flow analysis bounding the filling at ensure instructions, (3) a context-independent, function-local data-flow analysis bounding the worst-case occupancy for call sites, and (4) a fully context-sensitive analysis bounding the worst-case spilling of reserves.

The spilling and filling bounds at the stack control instructions can then be taken into account during WCET analysis to compute a final timing bound. Note that the alignment is not considered here and thus handled conservatively.

3. RELATED WORK

Static analysis [13, 3] of caches typically proceeds in two phases: (1) potential addresses of memory accesses are determined, (2) the potential cache content for every program point is computed. The alignment usually is not an issue, as the size can be aligned with the main memory's burst size. Through its simpler analysis model, the stack cache does not require the precise knowledge of addresses, thus eliminating a source of complexity and imprecision. It has been previously shown that the stack cache serves up to 75% of the dynamic memory accesses [1]. An extension to avoid spilling data that is coherent between the stack cache and main memory [10] was presented.

Our approach to compute the worst-case behavior of the stack cache has some similarity to techniques used to statically analyze the maximum stack depth [2]. Also related to the concept of the stack cache, is the register-window mechanism of the SPARC architecture, for which limited WCET analysis support exists in Tidorum Ltd.'s Bound-T tool [12].

Alternative caching mechanisms for program data exist with the Stack Value File [6] and several solutions based on Scratchpad Memory (SPM) (e.g. [8]), which manage the stack in either hardware or software.

4. BLOCK-ALIGNED STACK CACHE

As explained above, the original stack cache operates on words and thus does not automatically align transfers according to the main memory's requirements. With regard to average performance, this is less of an issue and may only lead to a less optimal utilization of the main memory's bandwidth. For the WCET analysis the issue is more problematic. The alignment of the stack cache content needs to be known or otherwise all access have to be assumed unaligned. This information, however, is highly dependent on execution history and thus inevitably increases analysis complexity.

4.1 Hardware Modifications

This work proposes a hardware extension that guarantees that the stack cache initiates aligned memory transfers only, i.e., the start address as well as the length of the memory transfer are multiples of the memory's alignment requirement, i.e. the burst size. This avoids the need to track the alignment of the stack cache content during the WCET analysis, while allowing us to perform all stack cache operations at word granularity, which improves the cache's utilization.

The stack cache is organized in blocks matching the burst size. Moreover, we logically reserve a block in the stack cache as an alignment buffer. Note that this reserved block is not fixed, instead the last block pointed to by `MT` *dynamically* serves as this alignment buffer. This block is not accessible, for instance, to the compiler, and thus reduces the effective size of the stack cache by one block. The buffer allows us to align all the memory transfers to the desired block size. With regard to the original stack cache (see Section 2), this corresponds to an additional invariant that needs to be respected by the stack cache hardware given a block size `BS`:

$$\texttt{MT} \mod \texttt{BS} = 0. \tag{1}$$

In order to respect this new invariant the stack control instructions have to be adapted as follows:

- **`sres` x:** Subtract x from `ST`. If the occupancy exceeds the stack cache size, a *spill* is initiated, which lowers `MT` by multiples of `BS` until the occupancy is smaller than $|SC|$.
- **`sens` x:** If the occupancy is not larger than x, a *fill* is initiated, which increments `MT` by multiples of `BS` so that $\texttt{MT} - \texttt{ST} \geq x$.
- **`sfree` x:** Add x to `ST`. If $\texttt{MT} < \texttt{ST}$, set `MT` to the smallest multiple of `BS` larger than `ST` and fill a single block from main memory.

It is easy to see that the modifications to `sres` and `sens` are minimal. Clearly, when Eq. 1 holds, spilling and filling in multiples of `BS` ensures that the equation also holds after these instructions. The reserved block serving as an alignment buffer, in addition guarantees that sufficient space is available during filling to receive data and sufficient data is available during spilling to transmit data.

The situation is more complex for `sfree`. Whenever a number of `sfree` instructions are executed in a sequence such that the occupancy becomes zero, the `MT` pointer needs to be updated. In order to satisfy Eq. 1, two options exist: (a) set `MT` to the largest multiple of `BS` *smaller* than `ST` or (b) set `MT` to the smallest multiple of `BS` *larger* that `ST`. The former option would mean that the cache's occupancy becomes negative, which would entail non-trivial modifications to the other stack control instructions. The second option, which represents a non-negative occupancy, thus is preferable. However, in order to guarantee that the content of the stack cache reflects the occupancy ($\texttt{MT} - \texttt{ST}$) a single block has to be filled from main memory.

4.2 Static Analysis

The static analysis proposed for the standard stack cache [5] is in large parts applicable to the new block-aligned stack cache proposed as well. The main difference is that the timing of `sfree` instructions also has to be analyzed.

An `sfree` is required to perform a fill, iff, the minimal occupancy before the instruction is smaller than the instruction's argument x. The analysis problem for free instructions is thus identical to the analysis of ensure instructions [5].

In addition, the displacement of function calls has to be refined to account for data of the caller that is reloaded to the cache by an `sfree` before returning from the call. This is particularly important for the minimum displacement, needed for the analysis of `sens`-instructions, as the original displacement analysis is not safe anymore, unless lazy spilling [10] is used. This information can easily be derived and propagated on the program's call graph.

5. EXPERIMENTS

For our experiments we extended the hardware implementation of the stack cache available with the Patmos processor [11] as well as the cycle-accurate simulation infrastructure and the accompanying LLVM compiler (version 3.4). The average case performance was measured for all benchmarks of the MiBench benchmark suite [4]. The benchmaks were compiled, with optimizations (`-O2`) and stack cache support enabled, and then executed on the Patmos simulator to collect runtime statistics. The simulator was configured to simulate a 2 KB data cache (32 B blocks, 4 way set-associative, LRU replacement policy, and write-through strategy), a 2 KB method cache (32 B blocks, associativity 8), and a 128 B stack cache (with varying configurations). All caches are connected to a shared main memory, which transfers data in 32 B bursts. A moderate access latency of 14 cycles for reads and 12 cycles for writes is assumed.

The benchmarks were tested under three different scenarios: (1) the stack cache performs unaligned memory transfers (`unaligned`), (2) the compiler generates suitable *padding* to align all stack allocations and consequently all memory transfers (`padding`), and (3) the stack cache employs the block-aligned strategy from Section 4 (`block-aligned`). Stack data is usually aligned at word boundaries for Patmos, which applies to the `unaligned` and `block-aligned` configurations. The `padding` configuration, however, aligns all data with the burst size (32 B).

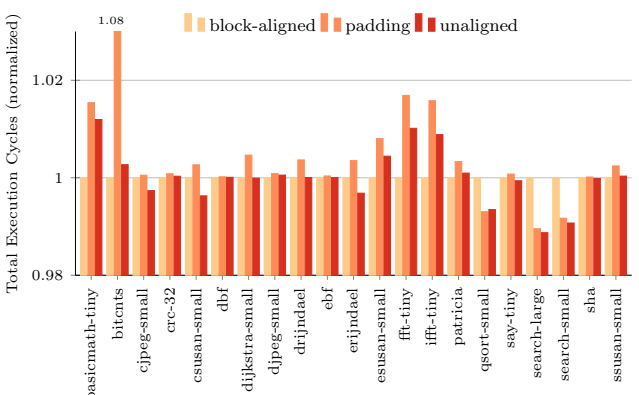

Figure 1: Total execution cycles normalized to the `block-aligned` configuration (lower is better).

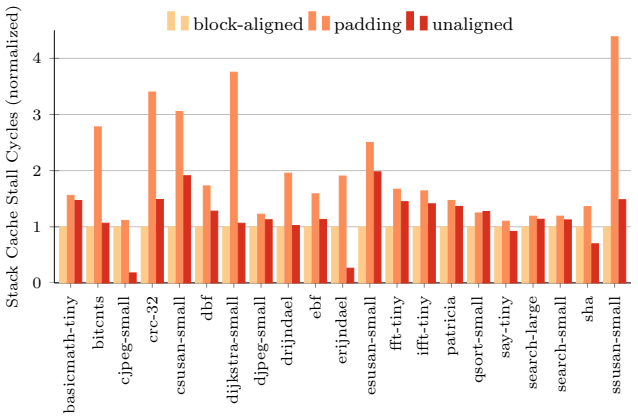

Figure 2: Total number of stall cycles induced by the stack cache normalized to the `block-aligned` configuration (lower is better).

The runtime impact of the various strategies to handle the alignment of memory transfers between the stack cache and the main memory is summarized in Figure 1. Overall, the `unaligned` and `block-aligned` configurations are very close with respect to runtime, while the `padding` configuration performs the least. In particular, the `bitcnts`, `basicmath-tiny`, `fft-tiny`, and `ifft-tiny` benchmarks here show runtime increases of 2% and more.

Note that the runtime contribution of the stack cache is relatively small, which in general precludes very large variations in the total runtime due to the stack cache. The simulator thus was extended to collect detailed statistics on the number of stall cycles induced by the stack cache as well as the spilling and filling performed. Figure 2 shows the total number of stall cycles induced by the stack cache, normalized to the `block-aligned` configuration. The `padding` configuration increases the number of stall cycles in all cases in relation to our `block-aligned` strategy (up to a factor of more than 4). The padding introduced by the compiler generally increases the stack cache's occupancy and consequently leads to additional memory transfers. Also for the `unaligned` configuration the number of stall cycles is larger than our new strategy, since the small unaligned memory transfers performed by this configuration induce some overhead. For two benchmarks, `cjpeg-small` and `erijndael`, the number of stall cycles is considerably smaller in this configuration. Our `block-aligned` stack cache here suffers additional filling and spilling due to its reduced effective size, as shown in the following Table.

The impact of the various configurations on the amount of data spilled and filled from/to the stack cache is shown in Table 1. As noted above the `padding` configuration performs additional memory transfers (spills and fills) due to the padding introduced by the compiler to ensure alignment. The `unaligned` configuration on the other hand requires the least filling and spilling as it transfers the precise amount of data needed. In addition, the reduced stack cache size available for the `block-aligned` strategy (recall that one block is reserved as an alignment buffer) plays in favor of the `unaligned` configuration.

To summarize, compiler generated padding is a simple solution to the alignment problem for the stack cache, which is easy to analyze and generally performs reasonably well,

	Block-Aligned		Padding				Unaligned			
Benchmark	Spill	Fill	Spill	rel.	Fill	rel.	Spill	rel.	Fill	rel.
basicmath-tiny	791 288	968 544	1 696 832	2.14	1 981 192	2.05	1 026 895	1.30	1 165 108	1.20
bitcnts	1 201 992	1 202 720	3 603 152	3.00	3 604 328	3.00	826 933	0.69	827 437	0.69
cjpeg-small	96 840	123 048	113 512	1.17	155 672	1.27	12 821	0.13	24 500	0.20
crc-32	2 984	3 312	11 256	3.77	38 688	11.68	2 961	0.99	3 114	0.94
csusan-small	9 184	10 000	32 136	3.50	35 936	3.59	13 403	1.46	13 619	1.36
dbf	11 984	78 136	22 456	1.87	124 672	1.60	10 395	0.87	66 056	0.85
dijkstra-small	98 056	99 520	472 944	4.82	478 152	4.80	49 047	0.50	50 518	0.51
djpeg-small	28 032	29 112	33 688	1.20	36 336	1.25	14 688	0.52	15 369	0.53
drijndael	168 584	212 800	329 600	1.96	373 952	1.76	29 693	0.18	49 823	0.23
ebf	37 536	103 624	65 040	1.73	243 872	2.35	30 618	0.82	114 959	1.11
erijndael	196 240	240 520	366 280	1.87	410 696	1.71	34 258	0.17	54 502	0.23
esusan-small	18 952	19 912	57 896	3.05	61 912	3.11	30 528	1.61	30 852	1.55
fft-tiny	215 936	217 712	448 480	2.08	470 464	2.16	277 788	1.29	279 011	1.28
ifft-tiny	206 464	207 952	425 440	2.06	446 864	2.15	264 063	1.28	264 994	1.27
patricia	3 883 936	4 590 920	6 638 160	1.71	8 089 800	1.76	3 740 659	0.96	4 514 761	0.98
qsort-small	643 296	1 283 680	1 126 664	1.75	2 411 584	1.88	772 978	1.20	1 493 122	1.16
say-tiny	261 016	358 976	303 296	1.16	460 560	1.28	98 239	0.38	118 011	0.33
search-large	216 632	322 344	323 680	1.49	534 824	1.66	207 978	0.96	366 106	1.14
search-small	8 528	14 816	12 688	1.49	24 984	1.69	7 963	0.93	16 214	1.09
sha	5 448	30 248	8 656	1.59	33 608	1.11	1 935	0.36	2 007	0.07
ssusan-small	18 280	19 096	66 736	3.65	71 336	3.74	16 247	0.89	16 463	0.86

Table 1: Words spilled and filled by the stack cache configurations `block-aligned`, `padding`, and `unaligned` (lower is better, rel. indicates the normalized value in comparison to the `block-aligned` configuration).

but may suffer from bad outliers. It generally leads to increased spilling and filling as well as a reduced utilization of the stack cache. Generating unaligned memory transfers naturally performs well for the average case. but, complicates WCET analysis since the alignment of the stack data is highly context dependent. The new solution proposed in this work, the block-aligned stack cache, offers a reasonable trade-off, which combines moderate hardware overhead with good average-case performance and simple WCET analysis.

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).

6. REFERENCES

[1] S. Abbaspour, F. Brandner, and M. Schoeberl. A time-predictable stack cache. In *Proc. of the Workshop on Software Technologies for Embedded and Ubiquitous Systems*. 2013.

[2] C. Ferdinand, R. Heckmann, and B. Franzen. Static memory and timing analysis of embedded systems code. In *Proc. of Symposium on Verification and Validation of Software Systems*, pages 153–163. Eindhoven Univ. of Techn., 2007.

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

[4] 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 *Proc. of the Workshop on Workload Characterization*, WWC '01, 2001.

[5] A. Jordan, F. Brandner, and M. Schoeberl. Static analysis of worst-case stack cache behavior. In *Proc. of the Conf. on Real-Time Networks and Systems*, pages 55–64. ACM, 2013.

[6] H.-H. S. Lee, M. Smelyanskiy, G. S. Tyson, and C. 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.

[7] L. Marchegiani. *Top-Down Attention Modelling in a Cocktail Party Scenario*. PhD thesis, Sapienza, University of Rome, 2012.

[8] S. Park, H. woo Park, and S. Ha. A novel technique to use scratch-pad memory for stack management. In *In Proc. of the Design, Automation Test in Europe Conference*, DATE '07, pages 1–6. ACM, 2007.

[9] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee. PRET DRAM controller: Bank privatization for predictability and temporal isolation. In *Proc. of the Conference on Hardware/Software Codesign and System Synthesis*, pages 99–108. ACM, 2011.

[10] S.Abbaspour, A. Jordan, and F. Brandner. Lazy spilling for a time-predictable stack cache: Implementation and analysis. In *Proc. of the International Workshop on Worst-Case Execution Time Analysis*, volume 39 of *OASICS*, pages 83–92. Schloss Dagstuhl, 2014.

[11] M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, C. Probst, S. Karlsson, and T. Thorn. *Towards a Time-predictable Dual-Issue Microprocessor: The Patmos Approach*, volume 18, pages 11–21. OASICS, 2011.

[12] BoundT time and stack analyzer - application note SPARC/ERC32 V7, V8, V8E. Technical Report TR-AN-SPARC-001, Version 7, Tidorum Ltd., 2010.

[13] R. T. White, C. A. Healy, D. B. Whalley, F. Mueller, and M. G. Harmon. Timing analysis for data caches and set-associative caches. In *Proceedings of the Real-Time Technology and Applications Symposium*, RTAS '97, pages 192–203, 1997.

[14] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling,

M. Pister, and C. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 28(7):966–978, 2009.