Lazy Spilling for a Time-Predictable Stack Cache: Implementation and Analysis

Sahar Abbaspour[1], Alexander Jordan[1], and Florian Brandner[2]

1 **Department of Applied Mathematics and Computer Science**
Technical University of Denmark {sabb,alejo}@dtu.dk
2 **Computer Science and System Engineering Department**
ENSTA ParisTech florian.brandner@ensta-paristech.fr

Abstract

The growing complexity of modern computer architectures increasingly complicates the prediction of the run-time behavior of software. For real-time systems, where a safe estimation of the program's worst-case execution time is needed, time-predictable computer architectures promise to resolve this problem. A stack cache, for instance, allows the compiler to efficiently cache a program's stack, while static analysis of its behavior remains easy. Likewise, its implementation requires little hardware overhead.

This work introduces an optimization of the standard stack cache to avoid redundant spilling of the cache content to main memory, if the content was not modified in the meantime. At first sight, this appears to be an average-case optimization. Indeed, measurements show that the number of cache blocks spilled is reduced to about 17% and 30% in the mean, depending on the stack cache size. Furthermore, we show that lazy spilling can be analyzed with little extra effort, which benefits the worst-case spilling behavior that is relevant for a real-time system.

1998 ACM Subject Classification C.3 [Special-Purpose and Application-Based Systems]: Real-time and embedded systems

Keywords and phrases Lazy Spilling, Stack Cache, Real-Time Systems, Program Analysis

Digital Object Identifier 10.4230/OASIcs.WCET.2014.83

1 Introduction

In order to meet the timing constraints in systems with hard deadlines, the worst-case execution time (WCET) of real-time software needs to be bounded. This WCET bound should never underestimate the execution time and should be as tight as possible. Many features of modern processor architectures, such as pipelining, caches, and branch predictors, improve the average performance, but have an adverse effect on WCET analysis. Time-predictable computer architectures propose alternative designs that are easier to analyze.

Memory accesses are crucial for performance. This also applies to time-predictable alternatives. Analyzable memory and cache designs as well as their analysis thus recently gained considerable attention [13, 8, 9]. One such alternative cache design is the *stack cache* [1, 5], i.e., a specialized cache dedicated to stack data, which is intended as a complement to a regular data cache. This design has several advantages. Firstly, the number of accesses going through the regular data cache is greatly reduced, promising improved analysis precision. For instance, imprecise information on access addresses can no longer interfere with the analysis of stack accesses (and vice verse). 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 [13] and has to take the (from an analysis point of view

14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014).
Editor: Heiko Falk; pp. 83–92

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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].

In this paper we propose an optimization to improve the performance of the stack cache's reserve operation based on the following observation: in many cases, the actual data spilled by a reserve has exactly the same value as the data already stored in main memory. This may happen in situations where data is repeatedly spilled, e.g., due to calls in a loop, but not modified in the meantime. Thus, the main idea is to track the amount of data that is coherent between the main memory and the cache. The cache can then avoid to needlessly spill coherent data. We show that this tracking can be realized efficiently using a single pointer, the so-called *lazy pointer*. We furthermore show that the existing analysis [5] can be easily adapted to this optimization, by simply refining the notion of occupancy, to account for the coherent space defined by the lazy pointer.

Section 2 introduces the stack cache, followed by a discussion of related work. Section 4 presents the motivation for our work. In Section 5, we explain lazily spilling and its static analysis. We finally present the results from our experiments in Section 6.

2 Background

The original stack cache [1] is implemented as a kind of ring buffer with two pointers: *stack top* (`ST`) and *memory top* (`MT`). The former points to the top of the logical stack, which consists of all the stack data that is either stored in the cache or main memory. The latter points to the top element present in main memory only. For simplicity, we herein assume a hardware implementation[1] with a stack growing towards lower addresses.

The difference $\texttt{MT} - \texttt{ST}$ represents the amount of occupied space in the stack cache. Clearly, this value cannot exceed the total size of the stack cache's memory $|SC|$, thus $0 \leq \texttt{MT} - \texttt{ST} \leq |SC|$. The stack control instructions hence manipulate the two stack pointers, while preserving this equation, and initiate the corresponding memory transfers as needed. A brief summary is given below (details are available in [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.
- **`sfree` x:** Add x to `ST`. If this results in a violation of the invariant, `MT` is incremented accordingly. Memory is not accessed.
- **`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 $\texttt{MT} - \texttt{ST} \geq x$ holds.

Using these three instructions, the compiler generates code to manage the stack frames of functions, quite similar to other architectures with exception of the ensure instruction. For brevity, we assume a simplified placement of these instructions. Stack frames are allocated upon entering a function (`sres`) and freed immediately before returning from a function (`sfree`). The content of a function's stack frame might be evicted from the cache upon function calls. The compiler thus ensures a valid stack cache state, immediately after each call site (`sens`). This simplified placement can be relaxed as discussed in Section 5.2. We furthermore restrict functions to only operate on their locally allocated stack frames. Any stack data shared between functions, or exceeding the stack cache's size, is allocated on the so-called *shadow stack* outside the stack cache.

[1] Note that the stack control instructions could also be implemented by means of software.

3 Related Work

Static analysis [12, 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. 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]. 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 [11, Section 2.2].

Alternative caching mechanisms for program data exist with the Stack Value File [6] and several solutions based on Scratchpad Memory (SPM) (e.g. [7]), which manage the stack in either hardware or software.

4 Motivating Example

Figure 1 shows a function `bar` using the stack cache allocating a stack frame of two words (l. 2) on a stack cache of size of 8 words. The stack frame is freed before returning from the function (l. 16). The loop (l. 5–14), repeatedly calls function `foo`. We assume function `foo` reserves 8 words, thus evicts the entire stack cache content and displaces 8 words.

Assuming no stack data has been allocated so far, the stack cache is entirely empty. The `MT` and `ST` pointers are pointing to the top of the stack at address 256. The `sres` instruction of function `bar` (l. 2) decrements `ST` by two, without any spilling. At this point, `MT` points to address 256 and `ST` to 248 resulting in an occupancy of 2 words.

The stack store and load instructions (l. 4, 7) do not modify the `MT` and `ST` pointers. After the function call to `foo`, an `sres` execution reserves 8 words. Decrementing `ST` by 32 with an unmodified `MT` would result in an occupancy of 10 words ($256 - 216$), exceeding the stack cache size. Two words are thus spilled and `bar's` stack frame is transferred to the main memory (addresses [248, 255]). Before returning from `foo`, its stack frame is freed, by incrementing `ST` by 32 (`ST` = `MT` = 248), meaning that the stack cache is empty. The `sens` instruction (l. 11) executed next, is required to ensure that `bar's` stack frame is present in the cache for the load of the next loop iteration. Therefore, a cache fill operation increments the `MT` pointer to 256. The current cache state is identical to the state before the function call. For subsequent loop iterations, the stack cache states change in exactly the same way. An interesting observation at this point is that the values of the respective stack slots are loop-invariant and do not change. Consequently, during every iteration the exact same values

```
1  function bar()
2     sres 2
3     // store loop-invariant stack data
4     sws [1] = ...
5  loop:
6     // load loop-invariant stack data
7     lws ... = [1]
8     // displaces entire stack cache
9     call foo
10    // reload local stack frame
11    sens 2
12    cmp ...
13    // jump to beginning of loop
14    bt loop
15    // exit function
16    sfree 2
17    ret
```

Figure 1 `foo` evicts the entire stack cache, `bar`'s stack data is thus spilled on each iteration.

are written to the main memory. Even more, the respective memory cells already hold these values for all loop iterations except for the first. A standard stack cache obviously lacks the means to avoid this redundant spilling of stack data that is known to be *coherent*.

5 Lazy Spilling

We propose to introduce another pointer, which we call *lazy pointer* (LP), that keeps track of the stack elements in the cache that are known to be coherent with main memory. The lazy pointer only refers to the data in the stack cache, thus the invariant $\texttt{ST} \leq \texttt{LP} \leq \texttt{MT}$ holds.

The LP divides the reserved space in the stack cache into two regions: (a) a region between the ST and LP and (b) a region between the LP and MT. The former region defines the *effective occupancy* of the stack cache, i.e., it contains potentially modified data. The data of the second region is coherent between the main memory and the stack cache. Whenever the ST moves up past the LP or the MT moves down below the LP, the LP needs to be adjusted accordingly. When a stack store instruction writes to an address above the LP, some data potentially becomes incoherent. Hence, the LP should move up along with the effective address of the store.

5.1 Implementation

Following the above observations, we need to adapt the stack control instructions and the stack store instructions to support lazy spilling. Note, however, that lazy spilling does not impact the ST and the MT, i.e., their respective values are identical to a standard stack cache.

`sres` x: The stack cache's `sres` instruction decrements the ST. Moreover, the reserve potentially spills some stack cache data to the main memory and thus may decrement the MT. To respect the invariant from above, this may require an adjustment of the LP to stay below the MT. We can, in addition, exploit the fact that the newly reserved cache content is uninitialized – and thus can be treated as coherent with respect to the main memory. For instance, when $\texttt{ST} = \texttt{LP}$ before the reserve, all the stack cache content is known to be coherent, which allows us to retain the equality $\texttt{ST} = \texttt{LP}$ even after the `sres` instruction. Moreover, when the space allocated by the current `sres` covers the entire stack cache, all the data in the stack cache is uninitialized. In this case it is again safe to assume $\texttt{ST} = \texttt{LP}$ after the reserve. Apart from updating the LP, the spilling mechanism itself requires modifications. Originally, the MT pointer is used to compute the amount of data to spill. However, when lazy spilling is used, the amount of data to spill does not depend on the MT anymore. Instead, the LP has to be used to account for the coherent data present in the stack cache. With respect to spilling, the LP effectively replaces the MT – hence the term effective occupancy for the region between the ST and the LP.

`sfree` x: The stack cache's `sfree` instruction increments the ST. The MT may potentially increase as well. To satisfy the invariant from above, the LP is incremented in case it is below the ST. No further action is required.

`sens` x: The stack cache's `sens` instruction does not modify the ST and may only increase the MT. The invariant is thus trivially respected. Moreover, coherency of the data loaded into the cache is guaranteed. The `sens` instruction thus requires no modification.

`store`: The stack store instruction writes to the stack cache and may thus change the value of previously coherent data. Therefore, the LP needs an adjustment whenever the effective address of the store is larger than the LP, i.e., the LP needs to be greater than the effective address of the store instruction.

Result: Construct SCA Graph G
initialize worklist W with entry context $(s, 0)$;
while *unhandled context c in W* **do**
  $f \leftarrow$ function of c;
  $e \leftarrow$ effective occupancy of c;
  foreach *call instruction i in f calling f'* **do**
    $e' \leftarrow \min(e + r, o_{\text{eff}}[i]))$;
    $c' \leftarrow (f', e')$;
    add edge $c \rightarrow c'$ to G;
    if *c' is a new context* **then**
      add new node to G;
      add context c' to W

Algorithm 1: Construct Spill Cost Analysis Graph

▶ **Example 1.** Consider again the program from Figure 1. The stack cache is initially empty and the `ST`, the `MT`, and the `LP` point to address 256. As before, the `sres` instruction moves the `ST` down to address 248. The `LP` moves along with the `ST`, since `ST` = `LP` and we assume that the newly reserved and uninitialized space is coherent with the main memory. Next, the store instruction (l. 4) modifies some data present in the stack cache. The `LP` consequently has to move upwards and now points to address 252. The first call to function `foo` causes two words to be spilled (see Section 4) and the `MT` is thus decremented to 248. As the `LP` is larger than the `MT` at this point, the `LP` would normally require an adjustment. However, since we assume that `foo` reserves the entire stack cache, it is safe to set the `LP` to the value of `ST` (216). Any stores within `foo` then automatically adjust the `LP` as needed. The stack cache is again empty after returning from function `foo`. All three pointers of the stack cache then point to address 248. The `sens` reloads the stack frame of function `bar` (l. 11), leaving both `ST` and `LP` unmodified, but incrementing `MT` to 256. The occupancy of the normal stack cache at this point is 8 (2 words), while the effective occupancy of the stack cache with lazy spilling is 0, i.e., the entire cache content is known to be coherent. The effective occupancy for the next call to `foo` as well as for subsequent iterations of the loop, stays at 0. The reserve within `foo` thus finds the entire stack cache content coherent and avoids spilling completely. Finally, when the program exits the loop, the `sfree` instruction (l. 16) increments `ST` to 256. In order to satisfy the invariant, `LP` also has to be incremented.

5.2 Static Analysis

With the restrictions from Section 2 in place, the analysis algorithms assume that reserve and free instructions appear at a function's entry and exit; this may be relaxed, given the program remains well-formed regarding its stack space. Due to space restrictions, we refer to [5], which describes the relaxation and the analysis of the original stack cache design.

The filling behavior of ensure instructions is not affected by lazy spilling, therefore, the context-insensitive *ensure analysis* remains unchanged compared to its original. The spilling behavior of a reserve instruction depends on the state of the stack cache at function entry, which in turn depends on a nesting of stack cache states of function calls reaching the current function. To fully capture this context-sensitive information, we can represent spill cost in the *spill cost analysis graph* (SCA graph). An example SCA graph is depicted in Figure 2b. The effective spill cost for an `sres` instruction in a specific calling context is computed from the occupancy of the context and the locally reserved space x (in `sres` x). This value is then multiplied with the timing overhead of the data transfer to external memory.

Without lazy spilling, the construction of the SCA graph depends on the notion of stack cache *occupancy*, i.e., the utilized stack cache region of size $\texttt{MT} - \texttt{ST}$. In order to benefit from the reduced overhead of lazy spilling, we need to consider the possibly smaller region $\texttt{LP} - \texttt{ST}$. Bounds for this effective occupancy of the stack cache are propagated through the call graph during *reserve analysis*, which constructs the SCA graph as shown in Algorithm 1. The o_{eff}-bound used by the algorithm improves the overestimation still present in context-sensitive reserve analysis. It can be computed by an intra-procedural data-flow analysis, which considers *minimum displacement* and the destinations of stack cache stores. For every point within a function, but most interestingly before calls, o_{eff} represents a local upper bound on the effective occupancy.

As a pre-requisite for the intra-procedural analysis, minimum displacements for all functions in the program are computed. This is achieved by several shortest-path searches in the program's call graph. The resulting values are bounded by the size of the stack cache $disp(i) = \min(|SC|, d_{min}(i))$ and used in the data-flow transfer functions of the analysis:

$$OUT(i) = \begin{cases} \min(IN(i), |SC| - disp(i)) & \text{if } i = call \quad (1a) \\ \max(IN(i), blocks(n)) & \text{if } i = \texttt{sws n} \quad (1b) \\ IN(i) & \text{otherwise} \quad (1c) \end{cases}$$

After returning from a call instruction i, the LP cannot have moved upwards with respect to its location before the call. In fact, a function call can only leave the LP unchanged or move it downwards. I.e., if stack contents of the current function are spilled, this potentially increases the coherent region of the stack cache and thus decreases the effective occupancy (1a). The amount of spilling is represented by the previously computed (worst-case) minimum displacement $disp(i)$.

Opposite to calls, the LP moves up when i is a store instruction (1b). Since the number of data words per stack cache block is fixed, the number of blocks that become incoherent due to a store, only depends on its destination n (in $blocks(n) = \lceil n/\text{words-per-block} \rceil$).

Note that while an `sens` instruction can impact the original analysis of occupancy bounds, it does not influence the effective occupancy and thus plays no role during reserve analysis in the presence of lazy spilling.

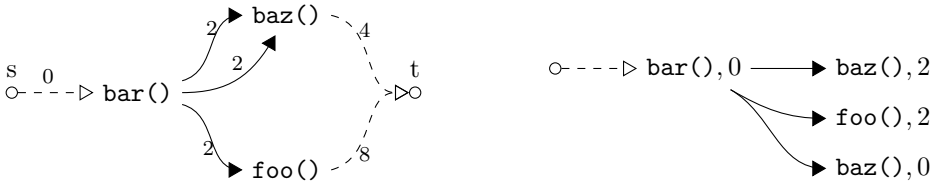
In order to safely initialize the o_{eff} bound and propagate it between instructions (i.e., from the OUT-values of the predecessors of an instruction to its IN-value (2b)), we define the following transfer functions:

$$IN(i) = \begin{cases} |SC| & \text{if } i \text{ is entry} \quad (2a) \\ \max\limits_{p \in Preds(i)}(OUT(p)) & \text{otherwise} \quad (2b) \end{cases}$$

It is further worth noting that in practice not all stack store instructions need to be analyzed. As long as the effective occupancy value exceeds the stack cache size reserved by the current function (i.e., the coherent region does not overlap with the current stack frame), a particular store has no effect on the analysis state.

▶ **Example 2.** Consider the program from Figure 1, where two additional function calls to function `baz` are performed. The first call appears right before entering the loop, while the other appears right after exiting the loop.

The call graph for this program is shown in Figure 2a. This graph also shows the size of the stack frame reserved by each function on the outgoing call edges. For functions without calls (`baz` and `foo`), call edges to an artificial sink node t are introduced. Using these annotations, the minimum and maximum displacement of each function can be determined



(a) Call graph (edges annotated with `sres` size of source node)

(b) SCA Graph (each node represents a stack cache context: name followed by the context's effective occupancy level)

Figure 2 Call graph and SCA Graph for the program of Example 2.

using a shortest and longest path search respectively. The minimum displacements for the three functions are 8, 6, and 4; for `foo`, `bar`, and `baz` respectively.

The analysis of this program results in the SCA graph shown by Figure 2b. The analysis starts with an empty stack cache and finds that `bar` allocates 2 words (8 bytes), initializes the respective stack slots and then calls `baz`. Since `bar`'s stack content was not spilled before, the effective occupancy when entering `baz` evaluates to 2. Assuming a stack cache size of 8 words, the function can be executed without spilling. The analysis thus can deduce that the `LP` was not modified with regard to `bar`'s stack frame. Propagating this information to the function call of `foo` within the loop results in a context for `foo` with the same occupancy of 2. Once the program exits the loop, another call to `baz` is executed. Due to `foo` displacing the full stack cache and the lack of relevant stores, the data-flow analysis is able to propagate the effective occupancy of 0 down to this call. This is reflected in the SCA graph by containing a node for `baz` annotated with its effective 0-occupancy.

6 Experiments

We use the Patmos processor [10] to evaluate the impact of lazy spilling on hardware costs, average program performance, and worst-case spilling bounds. The hardware model of the processor was extended and statistics were collected on the speed and resource requirements after synthesis (Altera Quartus II 13.1, for Altera DE2-115). The average performance measurements were performed using the MiBench [4] benchmarks suite. The programs were compiled using the Patmos LLVM compiler (version 3.4) with full optimizations (`-O3`) and executed on a cycle-accurate simulator. We compare five configurations utilizing (a) a standard data cache combined with a lazily spilling stack cache having a size of 128 or 256 bytes (LP_{128}, LP_{256}), (b) a standard data cache combined with a standard stack cache (SC_{128}, SC_{256}), and (c) a standard data cache alone (DC). The stack caches perform spilling and filling using 4 byte blocks. The data cache is configured to have a size of 2 KB, a 4-way set-associative organization, with 32 byte cache lines, LRU replacement, and a write-through strategy (recommended for real-time systems [13]). In addition to data caches, the simulator is configured to use a 16 KB method cache for code. The main memory is accessed in 16 byte bursts and 7 cycles latency.

6.1 Implementation Overhead

The standard stack cache of Patmos is implemented by a controller, which executes spill and fill requests, and control instructions (`sres`, `sfree`, `sens`) sending requests to that controller. The controller is independent from the extensions presented here. It thus suffices to extend the `sres`, `sfree`, and stack store instruction as indicated in Section 5.1. The Patmos processor

has a five-stage pipeline (fetch, decode, execute, memory access, write back). The stack cache reads the `ST` and the `MT` in the decode stage and immediately computes the amount to spill/fill. The `LP` is read in the decode stage as well, which allows us to perform all `LP`-related updates in the decode and execute stages. That way additional logic on the critical path in the memory stage, where the cache is actually accessed, is avoided. This applies in particular to the store instruction, whose effective address only becomes available in the execute stage. For lazy spilling, a single additional register is needed (`LP`). Updating the `LP` in the `sres` instruction adds two multiplexers to the original implementation. The `sfree` and stack store instructions each need an additional multiplexer. The area overhead is, therefore, very low. Moreover, these changes do not affect the processing frequency.

6.2 Average Performance

Table 1 (columns Spill) shows the reduction in the number of blocks spilled in comparison to the standard stack cache. Note that results for `rawcaudio` and `rawdaudio` are not shown, as they never spill due to their shallow call nesting depth. The best result for LP_{128} is achieved for `bitcnts`, where lazy spilling practically avoids spilling. In the mean, spilling is reduced to just 17%. The worst result is attained for `qsort-small`, where 62% of the blocks are spilled. For LP_{256} spilling is reduced to 30% in the mean. The best result is observed for `search-large`, where essentially all spilling is avoided. For `crc-32`, `drijndael`, `erijndael`, and `sha` only marginal improvements are possible, since these benchmarks already spill little in comparison to the other benchmarks. The worst result of those benchmarks with a relevant amount of spilling is obtained for `qsort-small`, where 76% of the blocks are spilled.

Miss rates are not suitable for comparison against standard data caches. We thus compare the number of bytes accessed by the processor through a cache in relation to the number of stall cycles it caused, i.e., $\frac{\#RD+\#WR}{\#Stalls}$. A high value in Table 1 (columns SC and DC) means that the cache is efficient, as data is frequently accessed without stalling. The data cache alone gives values up to 3.3 only. Ignoring benchmarks with little spilling, the best result for SC_{128} is achieved by `dbf` (477.4). For SC_{256}, `bitcnts` gives the best result (17054.7). Lazy

Table 1 Bytes accessed per stall cycle and reduction in spilling for the various configurations.

	SC_{128}			LP_{128}		SC_{256}			LP_{256}		DC
Benchmark	SC	DC	Spill	LP-SC	DC	SC	DC	Spill	LP-SC	DC	
basicmath-tiny	2.3	1.1	0.17	4.0	1.1	26.4	1.1	0.53	34.0	1.1	1.1
bitcnts	4.6	191.6	0.00	12.2	191.6	17054.7	193.7	0.71	19201.4	193.7	1.2
cjpeg-small	116.9	1.0	0.51	148.4	1.0	3470.7	1.0	0.09	6154.4	1.0	1.1
crc-32	9.0	0.9	0.03	21.3	0.9	814.9	0.9	1.00	814.9	0.9	0.9
csusan-small	11.3	2.2	0.16	18.6	2.2	1218.8	2.3	0.72	1430.0	2.3	1.5
dbf	477.4	1.0	0.47	623.0	1.0	–	1.0	–	–	1.0	1.0
dijkstra-small	19.5	1.4	0.20	32.8	1.4	335.2	1.4	0.54	433.7	1.4	1.4
djpeg-small	9.0	0.8	0.34	13.5	0.8	293.4	0.8	0.66	361.5	0.8	0.8
drijndael	15.8	0.9	0.20	28.7	0.9	185620.0	0.9	1.00	185620.0	0.9	0.9
ebf	172.5	1.0	0.44	224.6	1.0	–	1.0	–	–	1.0	1.0
erijndael	32.6	0.9	0.57	43.3	0.9	258340.0	0.9	1.00	258340.0	0.9	0.9
esusan-small	15.9	3.4	0.25	25.3	3.4	70.7	3.6	0.02	139.5	3.6	1.5
fft-tiny	3.1	1.1	0.08	5.8	1.1	85.0	1.1	0.56	103.4	1.1	1.1
ifft-tiny	3.1	1.2	0.08	5.9	1.2	83.1	1.1	0.56	101.0	1.1	1.1
patricia	2.5	1.0	0.27	4.2	1.0	26.4	1.0	0.55	31.9	1.0	1.0
qsort-small	3.1	1.0	0.62	3.7	1.0	7.8	1.0	0.76	8.6	1.0	1.0
rsynth-tiny	16.0	1.9	0.08	29.9	1.9	1096.1	1.9	0.48	1539.8	1.9	1.3
search-large	2.9	0.8	0.48	3.9	0.8	26.3	0.8	0.00	52.5	0.8	0.9
search-small	2.9	0.8	0.49	3.7	0.8	28.1	0.8	0.02	54.8	0.8	0.9
sha	8.3	1.6	0.20	14.1	1.6	668.7	1.6	0.91	700.6	1.6	1.6
ssusan-small	29.2	17.1	0.20	43.9	17.1	4313.5	17.1	0.80	4678.0	17.1	3.3

spilling leads to consistent improvements over all benchmarks. An interesting observation is that for most benchmarks the presence of a stack cache *improves* the performance of the data cache. The best example for this is `bitcnts`, but also `csusan` and `ssusan` profit considerably, where the data cache alone delivers 1.2 bytes per stall cycle. When a stack cache is added to the system, this value jumps up to 192.4 and 196.1 respectively.

Our measurements show that lazy spilling eliminates most spilling in comparison to a standard stack cache. Also the efficiency of the standard data cache is improved in many cases. Due to the low memory latency assumed here, this translates to run-time gains of up to 21.8% in comparison to a system with a standard stack cache. In the mean, the speedup amounts to 8% and 9.2% in comparison to a system only equipped with a data cache.

6.3 Static Analysis

We evaluate the impact of lazy spilling on the static analysis by comparing its bounds on the worst-case spilling with the actual spilling behavior observed during program execution. In order to be considered *safe*, the analysis' bounds always need to be larger or equal to the observed spilling of an execution run. Apart from a safe result, the analysis should also provide *tight* bounds. Note that the observed spilling relates to the average-case behavior of the programs, which does not necessarily trigger the worst-case stack cache behavior (the same inputs as in Section 6.2 have been used). However, we still report the estimation gap between worst case and average case, to show the effect of lazy spilling in the analysis.

We extended the Patmos simulator and first verified that the actual spilling of all `sres` instructions executed by a benchmark program is safely bounded by the analysis. We then measured the maximum difference between statically predicted and observed spilling (Max-Spilling-Δ) for each `sres` in every of its contexts (i.e., considering all nodes of the SCA graph reachable by execution). Table 2 shows the results for the 128-byte stack cache configuration and allows for a comparison between statically predicted and dynamically observed spill costs. A first look reveals that static spill cost is reduced for all programs in our benchmark set (down to 12% for `rsynth-tiny`). Furthermore, when the estimation gap is initially low, lazy spilling tends to widen the gap between static and dynamic spill cost

Table 2 Analysis precision: static worst-case compared to observations from dynamic execution (bytes spilled based on maximum spilling per stack cache context)

	SC_{128} Max-Spilling-Δ			LP-SC_{128} Max-Spilling-Δ		
Benchmark	Static	Dynamic	Gap	Static	Dynamic	Gap
basicmath-tiny	68,128	32,040	2.13×	10,052	8,080	1.24×
bitcnts	892	684	1.30×	768	320	2.40×
crc-32	844	652	1.29×	684	372	1.84×
csusan	5,404	2,592	2.08×	2,420	1,196	2.02×
dbf	684	456	1.50×	564	324	1.74×
dijkstra-small	10,220	5,796	1.76×	6,676	2,608	2.56×
drijndael	1,172	664	1.77×	1,024	488	2.10×
ebf	684	456	1.50×	564	324	1.74×
erijndael	880	400	2.20×	752	292	2.58×
esusan	4,724	1,888	2.50×	2,256	1,024	2.20×
fft-tiny	32,484	9,476	3.43×	5,804	3,712	1.56×
ifft-tiny	32,224	9,256	3.48×	5,620	3,548	1.58×
patricia	1,996	1,672	1.19×	1,804	984	1.83×
qsort-small	3,804	1,492	2.55×	2,432	840	2.90×
rsynth-tiny	109,864	15,320	7.17×	13,504	3,140	4.30×
search-large	840	740	1.14×	668	340	1.96×
search-small	828	728	1.14×	708	312	2.27×
sha	1,160	660	1.76×	1,032	448	2.30×
ssusan	6,608	1,824	3.62×	2,452	1,060	2.31×

slightly. But for those benchmarks that exhibit large estimation gaps with a standard stack cache, lazy spilling can even improve analysis precision.

7 Conclusion

From our experiments, we conclude that the benefits of lazy spilling extend from the average case performance to worst-case behavior, where it can even benefit analysis precision. In future work, we plan to introduce loop contexts and a limited notion of path-sensitivity to the analysis, to better capture occupancy states and thus spilling behavior.

Acknowledgment. This work was partially funded under the EU's 7th Framework Programme, grant agreement no. 288008: Time-predictable Multi-Core Architecture for Embedded Systems (T-CREST).

References

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

2 Christian Ferdinand, Reinhold Heckmann, and Bärbel 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 Christian Ferdinand and Reinhard Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2-3):131–181, 1999.

4 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.

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 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.

7 Soyoung Park, Hae woo Park, and Soonhoi 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.

8 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.

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

10 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*, volume 18, pages 11–21. OASICS, 2011.

11 BoundT Time and Stack Analyzer – Application Note SPARC/ERC32 V7, V8, V8E. Technical Report TR-AN-SPARC-001, Version 7, Tidorum Ltd., 2010.

12 Randall T. White, Christopher A. Healy, David B. Whalley, Frank Mueller, and Marion 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.

13 Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 28(7):966–978, 2009.