Eager Stack Cache Memory Transfers

Amine Naji[1] and Florian Brandner[2]

1 **U2IS, ENSTA ParisTech, Université Paris-Saclay**
amine.naji@ensta-paristech.fr
2 **LTCI, CNRS, Télécom ParisTech, Université Paris-Saclay**
florian.brandner@telecom-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. The stack cache, for instance, allows the compiler to efficiently cache a program's stack, while static analysis of its behavior remains easy.

This work introduces an optimization of the stack cache that allows to *anticipate* memory transfers that might be initiated by future stack cache control instructions. These eager memory transfers thus allow to reduce the average-case latency of those control instructions, very similar to "prefetching" techniques known from conventional caches. However, the mechanism proposed here is guaranteed to have no impact on the worst- case execution time estimates computed by static analysis. Measurements on a dual-core platform using the Patmos processor and time-division-multiplexing-based memory arbitration, show that our technique can eliminate up to 62% (7%) of the memory transfers from (respectively to) the stack cache on average over all programs of the MiBench benchmark suite.

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

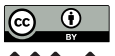
Keywords and phrases Predictability, Eager Memory Transfers, Stack Cache, Real-Time Systems, Prefetching, Eager Eviction

1 Introduction

The design of modern computer architectures has become more and more complex over the last decades in order to optimize performance and efficiency. In the vast majority of the cases modern architectures try to improve the average-case performance[1] by introducing instruction and data caches, branch predictors, instruction pipelines, and out-of-order execution. The optimizations usually follow the popular design principle: "Make the common case fast." [8]. A downside of this approach is that rare corner cases are often slowed down, which leads to a considerable gap between the best-case and worst-case performance that can be observed. This, in turn, complicates the precise analysis of the timing behavior of real-time programs running on such computer architecture and often results in considerable overestimation.

Time-predictable computer architectures thus gained considerable traction in recent years [15, 9, 12, 10]. In these designs the focus is on predictable and analyzable behavior, while retaining acceptable average-case performance. The stack cache [2] is an example of a predictable cache design that was shown to be analyzable [5, 1], while efficiently handling memory accesses to stack data [11] at low (hardware) cost. Stack data is cached using a sliding *window* that follows the top of the stack across function calls. The cache is implemented

[1] Energy consumption recently gained considerable importance as a secondary design goal.



using a ring buffer, following a FIFO policy. Data accesses are, by definition, guaranteed cache hits, the content of the cache thus has to be managed explicitly using three stack cache control instructions: (1) `sres` k allows to reserve k words on the stack, (2) `sfree` k can be used to free previously reserved stack space, and (3) `sens` k, finally, can be used to make sure that at least k words are available in the cache. Only the reserve (`sres`) and ensure (`sens`) operations may initiate time-consuming memory transfers and thus need to be considered during timing analysis [5, 1]. In the case of the `sres` instruction, content might be evicted, or *spilled*, from the cache in order to make space for the k newly reserved words. The `sens` instruction on the other hand might require to *fill* data from main memory when less than k words are available in the cache. The remaining stack cache operations (notably `sfree`) have constant timing and are thus trivial to analyze.

In order to improve predictability and ensure composability, the original stack cache design [2] stalls the processor while performing spilling or filling, even when the stack cache would not be used by any of the subsequent instructions. This allows to analyze the stack cache's timing behavior in isolation from other components of the Patmos computer architecture [12] at the expense of average-case performance. In this work, we explore the use of eager – or anticipatory – memory transfers in order to alleviate this shortcoming. The goal is to improve average-case performance by performing memory transfers in the background alongside with other instructions that are executed by the processor. The eager transfers are, however, not allowed to interfere with the worst-case behavior of the stack cache (or any other hardware component in the system). Most notably, the timing bounds computed for a regular stack cache without our optimizations, should not be invalidated in the presence of our optimizations. This is ensured by exploiting features of a recently proposed stack cache extension [11] to track data that are coherent between the cache and main memory.

2 Background

The stack cache is implemented as a ring buffer with two hardware registers holding pointers [2]: *stack top* (`ST`) and *memory top* (`MT`). The top of the stack is represented by `ST`, which points to the address of all stack data either stored in the cache or in main memory. `MT` points to the top element that is stored only in main memory. The stack grows towards lower addresses. The difference $\texttt{MT} - \texttt{ST}$ represents the amount of occupied space in the stack cache. This notion of *occupancy* is crucial for the effective analysis of the stack cache behavior [5]. In this work we will use an extension of the original stack cache design that allows to track coherent data [11]. This extension introduces a third pointer `LP`, which divides the stack cache into two parts: (1) cache data between `ST` and `LP` is potentially incoherent with the corresponding addresses in main memory, while (2) data between `LP` and `MT` is known to have the same value in the cache and in main memory – the data is coherent. The knowledge about coherent data allows to optimize the stack cache's operation. This is captured by a complementary notion of occupancy, called *effective occupancy* that is given by $\texttt{LP} - \texttt{ST}$.

Clearly, the (effective) occupancy cannot exceed the total size of the cache's memory $|SC|$. The stack cache thus has to respect the following invariants:

$$\texttt{ST} \leq \texttt{MT} \quad (1) \qquad 0 \leq \texttt{MT} - \texttt{ST} \leq |SC| \quad (2) \qquad \texttt{ST} \leq \texttt{LP} \leq \texttt{MT} \quad (3)$$

Stack Cache Operations: The stack control instructions manipulate the three stack pointers and initiate memory transfers to/from the cache from/to main memory, while preserving the above equations. A brief summary of each control instruction is given below, further details are available in [2, 11]:

sres k: Subtract k from ST. If this violates the Equations from above, data has to be evicted from the cache. In the simplest case only coherent data is discarded, i.e., $\texttt{LP} - \texttt{ST} \leq |SC|$ but $\texttt{MT} - \texttt{ST} > |SC|$. It then suffices to set $\texttt{MT} = \texttt{ST} + |SC|$. Otherwise, a memory *spill* of incoherent data has to be performed by a transfer covering the address range $[\texttt{ST} + |SC|, \texttt{LP}]$ to main memory. When spilling is completed, the LP and MT pointers are updated $\texttt{LP} = \texttt{MT} = \texttt{ST} + |SC|$.

sfree k: Add k to ST. If this violates Equation 1 or 3, MT and/or LP are simply set to ST. Main memory is not accessed.

sens k: Ensure that the occupancy is larger than k. If this is not the case, a *fill* from main memory is initiated covering the address range $[\texttt{MT}, \texttt{ST} + k]$. MT is subsequently incremented to $\texttt{MT} = \texttt{ST} + k$. LP does not change.

Data in the cache is accessed using dedicated stack load and store instructions. These instructions only access the stack cache's ring buffer and thus exhibit constant execution times. This is particularly true for stack store instructions, which only modify the cached value but not the backing main memory. Modified values are potentially transferred to main memory only by sres instructions. This policy resembles traditional *write back* caches. Also note that the LP might be updated by stack store instructions, i.e., when previously coherent data is modified. This has no impact on the constant instruction latency [11].

Compiler Support: The compiler manages the stack frames of functions quite similar to other architectures with exception of the ensure instructions. Stack frames are typically allocated upon entering a function (sres) and freed immediately before returning (sfree). A function's stack frame might be (partially) evicted from the cache during calls. Ensure instructions (sens) are thus placed immediately after each call. Evicted data is then reloaded into the cache. Functions may only access their own stack frames. Data that is larger than the stack cache or that is shared is allocated on a *shadow stack* outside the stack cache.

3 Eager Memory Transfers

Prefetching is a well-known technique used in conventional caches, which aims to hide memory access latencies caused by cache misses. Instead of waiting for a cache miss to initiate a memory transfer, prefetching anticipates such misses and fetches data from memory in advance of the actual memory reference. The idea, though simple, raises two important problems: (1) the addresses of future memory references need to be predicted and (2) side effects may arise due to the eviction of data from the cache in order to make space. Both of these issues are difficult to solve in general settings and pose even more problems in the context of real-time systems requiring predictability.

We explore the use of *eager memory transfers* – combining prefetching and eager eviction [7] – in order to reduce the latency of the stack cache control instructions. We introduce two kinds of eager memory transfers: (1) *eager spilling* transfers data from the stack cache to main memory, while (2) *eager filling* transfers data from main memory to the stack cache. The stack cache, in contrast to conventional caches, tracks its content using simple pointers and thus can only cache a contiguous memory region between the ST and MT pointers. In the following we will exploit this feature in order to realize "prefetching-like" functionality for the stack cache and address the two aforementioned problems faced in standard caches.

Address Prediction: Due to the use of pointers to track the stack cache content, it is trivial to predict the address of any future memory transfers that might be initiated by any stack cache control instruction. Data is either read from memory at the address starting at MT or written to memory at the address up to LP, depending on whether the (effective)

occupancy will grow too large (`sres` spilling up to `LP`) or will become too small (`sens` filling from `MT`). It thus suffices to predict whether data needs to be spilled or filled with regard to the future stack cache control instructions.

Side effects: We rely on a recently proposed stack cache extension [11] that allows to track coherent data between the stack cache and main memory in order to avoid side effects when performing eager memory transfers. A first observation is that eager spilling only needs to consider incoherent data (just like regular spilling). The eagerly spilled data is, however, not evicted from the stack cache. Instead, it simply becomes coherent. Since no data was evicted from the cache, side effects on future `sens` instructions are excluded. Similarly, since the amount of incoherent data was reduced, the spilling at future `sres` instructions is potentially reduced. A second observation is that eagerly filled data is known to be coherent. Side effects on future `sres` instructions are consequently excluded after eager filling since the amount of incoherent data did not change. The filling at future `sens` instructions, on the other hand, is reduced due to the newly loaded data.

The eager memory transfers are guaranteed to have no side effects on the stack cache itself. However, side effects on other hardware components, and here in particular the bus and main memory, may arise. For instance, a cache for regular data might be blocked by an eager memory transfer upon a cache miss. Such interferences may, of course, impact the program's worst-case performance and compromise predictability as well as composability.

An elegant solution is to exploit the arbitration scheme that mitigates between competing memory accesses [3, 1]. In the context of this work, we use the Patmos multi-core architecture, which relies on time-division multiplexing (TDM) to arbitrate main memory accesses. In the following we assume that each processor core may transfer a single memory burst from/to main memory in a dedicated *TDM slot*. Transfers may only be initiated at the beginning of a TDM slot, which are periodically scheduled in a *TDM period*. The duration of a period then depends on the number of cores n and the duration of a TDM slot k and is given by $n \cdot k$ cycles. We assume that the memory controller is able to process transfers with arbitrary start addresses and lengths. The actual memory transfer is, however, performed at the granularity of bursts, i.e., the start address and length are aligned accordingly to the burst size (excess data is either masked or discarded). In such a setting it is easy to detect TDM slots that are not used by any other hardware component. It suffices to check that no other memory request is pending at the beginning of the processor core's TDM slot. The free TDM slots of a processor can then be used to perform the eager memory transfers and avoid any side effects on either the stack cache itself nor any other hardware component.

3.1 Eager Fill

The *eager fill* operation aims to reduce the latency of a future ensure instruction `sens` k. Recall that filling is required only when the occupancy is too small, i.e., $\texttt{MT} - \texttt{ST} < k$. The occupancy has to be increased in order to reduce the latency. This can be achieved by loading, i.e. filling, data from main memory such that `MT` can be pushed upwards until the occupancy reaches the stack cache size. The corresponding memory transfer, however, has to be limited to a single burst transfer in order to guarantee that only a single TDM slot is occupied. Assuming a burst size `BS`, an eager fill operation thus proceeds as depicted by the algorithm in Figure 1a. The eager fill operation can be initiated whenever a TDM slot is free and is then guaranteed to be free of any interference with other hardware components that might wish to access main memory. It remains to show that the worst-case timing of subsequent stack cache operations is not affected. Three cases have to be considered, depending on the kind of the next stack cache control instruction:

```
if (MT - ST < |SC|) {
  start = MT;
  end = ⌊(MT+BS)/BS⌋ × BS;
  fill(start, end);
  MT = end;
}
```

(a) The eager fill operation.

```
if (LP − ST < k) {
  end = LP;
  start = ⌈(LP−BS)/BS⌉ × BS;
  spill(start, end);
  LP = start;
}
```

(b) The eager spill operation.

Figure 1 Pseudo code illustrating the operation of the eager filling and eager spilling.

sres k: May only initiate a memory transfer when incoherent data has to be evicted from the cache. The address range of the transfer ($[\texttt{ST} + |SC|, \texttt{LP}]$) only depends on the position of ST and LP. Eager filling does not modify either of those pointers (effective occupancy) and thus cannot impact spill costs.

sfree k: Free instructions do not access memory and exhibit constant latency.

sens k: May only initiate a memory transfer when the occupancy is too low. The address range of the transfer ($[\texttt{MT}, \texttt{ST} + k]$) only depends on ST and MT. The former is not impacted by eager filling, while the address of MT is incremented, i.e., the occupancy was previously increased. Fill costs thus may only be reduced.

Eager fill operations, consequently, may only improve the latency of future sens instructions. Note, however, that some side effects may still arise. This may appear when all filling of an sens instruction is eliminated. In this case, the sens instruction no longer synchronizes with the TDM period and may change the alignment of subsequent memory accesses. This may incidentally increase the number of stall cycles of these memory accesses. The number of additional stall cycles can, however, never exceed the gain induced by eager filling. WCET estimates computed without considering eager filling thus remain valid.

3.2 Eager Spill

The aim of the eager spill operation is to anticipate and reduce future spill costs associated with subsequent sres instructions. A spill is initiated by an sres if the effective occupancy would exceed the size of the stack cache, i.e., $\texttt{LP} - \texttt{ST} > |SC|$. The effective occupancy thus has to be lowered in order to reduce the spill latency. One possible solution is to copy incoherent stack data to main memory without evicting them from the cache. This allows to decrement LP and thus reduce the effective occupancy.

As for eager filling, the corresponding memory transfer size must not exceed the burst size so that at most one TDM slot is used. Assuming a burst size BS, an eager spill operation then proceeds as depicted by Figure 1b. The eager spill operation can be performed during free TDM slots as soon as the effective occupancy is non null. We will, nonetheless, prevent the spilling of data from the stack frame of the current function. This is because it may happen that data about to be eagerly spilled is modified by a stack store instruction. This would require additional checks to ensure that incoherent data is correctly tracked and increase hardware costs as well as complexity. As before, only free TDM slots are used, which guarantees that eager spill operations cannot interfere with other memory accesses. The worst-case timing of subsequent stack cache control operations is also not affected:

sres k: May only initiate a memory transfer when the effective occupancy becomes too large. The covered address range ($[\texttt{ST} + |SC|, \texttt{LP}]$) only involves the ST and LP

pointers. The latter is lowered by eager spilling, while the former is not modified, i.e., effective occupancy was previously decreased. The spill costs experienced by an `sres` instruction thus may only be reduced.

`sfree` k: Free instructions do not access memory and exhibit constant latency.

`sens` k: May only initiate a memory transfer when the occupancy is too low. The address range of the transfer ($[\texttt{MT}, \texttt{ST} + k]$) only depends on `ST` and `MT`. Both are not impacted by eager spilling. Fill costs thus cannot be impacted by eager spilling.

Eager spill operations, consequently, may only improve the latency of future `sres` instructions. Similarly to eager filling, the alignment of memory accesses with regard to the TDM period may change. The worst-case timing behavior of the program is not impacted.

3.3 Spill/Fill Arbitration

The eager fill and spill operations can be executed asynchronously alongside other instructions that are executed by the processor whenever a free TDM slot is encountered and the respective conditions necessary to perform a transfer are met. The two operations naturally compete for the available TDM slots, we thus defined several simple arbitration policies.

Spill/Fill-Only: As the names indicate, in these two configuration schemes only one of the two eager operations is performed throughout program execution, subject to the respective conditions as described above. This allows to quantify the attainable profit of either operation, ignoring the potential overhead induced by unprofitable eager transfers.

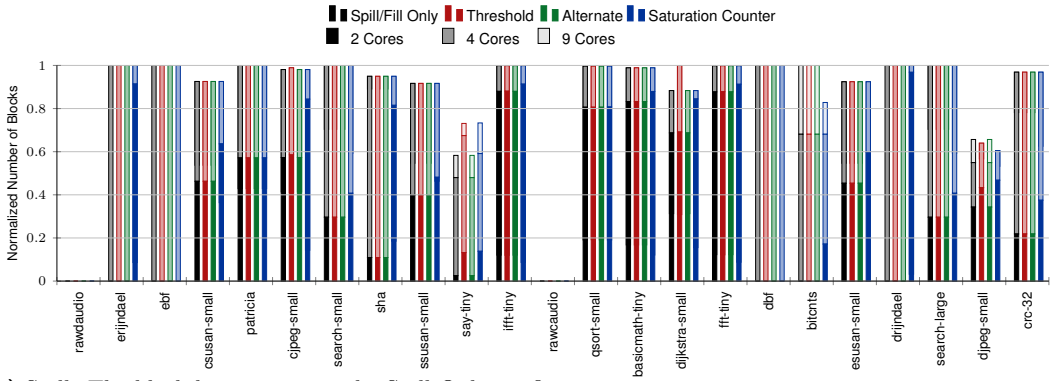
Alternate: Eager spill and fill are performed alternatingly in order to attain the maximum profit by applying both operations whenever this is possible on a fair arbitration policy.

Threshold: This approaches aims to reduce the amount of unprofitable eager operations, e.g., eagerly spilling data that is never evicted. Eager operations are performed alternatingly until a preset (effective) occupancy level (threshold) is reached. In the experiments, eager spilling stops when the effective occupancy is half of the stack cache size. Likewise, eager filling stops when the occupancy reaches half of the stack cache size.

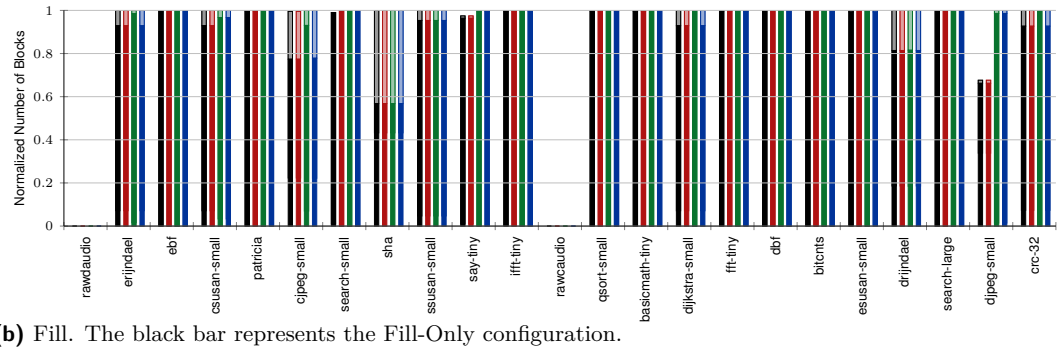
Saturation Counter: In this approach, the kind of the next stack control instruction is predicted and eager operations chosen such that its transfer costs are reduced. The hypothesis is that `sres` and `sens` instructions are performed in sequences when descending/ascending the call chain. The prediction uses a saturation counter, similar to branch prediction [8], that is in-/decremented up to prespecified maximum levels whenever an `sres`/`sens` instruction is encountered. The eager spill/fill operations are then only permitted when the counter value lies within predefined ranges. We use a simple 1-bit saturation counter in the experiments.

4 Experiments

We evaluated eager memory transfers using the cycle-accurate simulator of the Patmos processor [12], which implements a stack cache and its associated control instructions. It also allows to simulate several processor cores in parallel that access a shared main memory using bursts of 32 B. Memory arbitration is then performed using a TDM policy. We furthermore extended the stack cache implementation to support eager memory transfers using the arbitration strategies described above. Benchmarks of the MiBench benchmark suite [4] were compiled using optimizations (`-O2`) and subsequently executed on multi-core configurations with 2, 4 (2×2), and 9 (3×3) cores. Each core is equipped with a 256 byte stack cache, a 64 KB, 4-way set-associative data cache using a *least-recently used* replacement and write-through policy, as well as a 64 KB, 64-entry method cache using *first in, first out*



(a) Spill. The black bar represents the Spill-Only configuration.



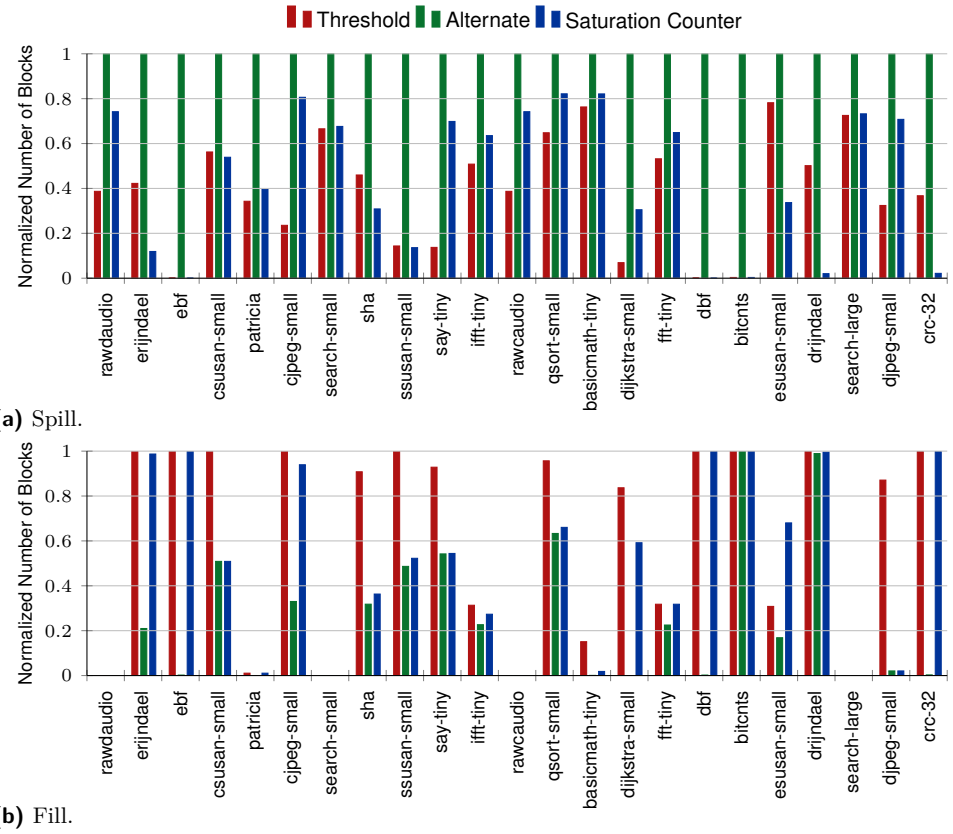
(b) Fill. The black bar represents the Fill-Only configuration.

Figure 2 Normalized number of total cache blocks regularly spilled/filled with respect to standard stack cache implementation supporting lazy pointer. (Lower is better)

replacement. The stack cache operates on 4 byte blocks, while the block size of the other caches matches the burst size of the main memory. Memory accesses take 21 cycles.

Figure 2 shows the normalized reduction in the number of blocks spilled and filled by `sres` and `sens` instructions in comparison to regular program execution without eager memory transfers. For eager spilling, results show a considerable reduction of spill costs by 62% over all benchmarks for the dual-core platform. For several benchmarks all spilling is performed by the eager operation (`erijndael`, `ebf`, `dbf`, `bitcnts`, `drijndael`). The total stack size of `rawcaudio` and `rawdaudio` fits into the stack cache. So, no spilling is ever performed for these benchmarks. The results for 4 and 9 cores are very close and give reductions of 6% and 1% respectively. Notable differences can be observed for `say-tiny`, `bitcnts`, and `djpeg-small`. This can be explained by the increased TDM period, which reduces the number of free TDM slots and the potential to perform eager memory transfers. All arbitration strategies were able to reduce the number of blocks spilled by `sres` instructions. The Alternate and Threshold configurations performed best and almost always reached the best possible result represented by the Spill-Only strategy.

The results for eager filling are less pronounced, resulting in reductions of only 7.4%, 1.7%, and 0.1% for the platforms with 2, 4, and 9 cores respectively. The large difference with eager spilling is surprising. Investigations showed that our hypothesis that `sres`/`sens` instructions often appear in sequences appears to hold. However, the average distance between `sres` instructions is typically much larger than the distance between `sens` instructions. The probability to encounter free TDM slots thus is much smaller between consecutive `sens` instructions, thus reducing the amount of eager filling that can be performed. Again, all strategies are able to achieve reductions. However, the Threshold configuration clearly performs best. This is once more surprising, since the theoretical bandwidth available for



(a) Spill.

(b) Fill.

Figure 3 Efficiency of the various eager spill/fill arbitration policies relative to the Spill- and Fill-Only configurations on a dual-core platform (Lower is better).

filling in the Alternate approach should at least reach 50% of the bandwidth of the Threshold configuration. It appears that the limited number of TDM slots available in between `sens` instructions aggravates the competition with eager spilling, explaining this bias. Further investigations are, however, needed to confirm this hypothesis.

We also performed measurements on a single-core configuration, where the processor performs memory accesses using a private bus (without TDM). Eager operations were initiated following the Alternate arbitration scheme immediately when no other bus requests were pending. Note that in this case interferences with other memory accesses frequently occur. We observed that spilling *and* filling of the stack control instructions was completely eliminated for almost all benchmarks, i.e., all cache transfers were carried out by eager operations. This indicates that eager transfers are effectively limited by the number of free TDM slots. An interesting idea would thus be to investigate means to explicitly allocate non-free TDM slots to eager operations. This could allow to entirely eliminate stalls at the stack control instructions in an analyzable and predictable manner.

In addition to the effective reduction by the various configurations in the number of memory transfers suffered by `sres` and `sens` instructions, we also compared the relative efficiency of the approaches. Figure 3 shows the normalized number of blocks eagerly spilled/filled with respect to the aggressive Spill-Only and Fill-Only configurations respectively. The Threshold configuration appears to provide the best trade-off between efficiency and the actual reduction of memory transfers by the stack control instructions. On the dual-core platform and over all benchmarks, it eagerly spills 60% and eagerly fills 30% fewer cache blocks than the Spill-/Fill-Only configurations respectively. Still the amount of excess spilling

(and to a lesser degree filling) is considerable. On average, over all benchmarks 75 times the number of cache blocks are spilled compared to the number of cache blocks spilled by the program when eager memory transfers are deactivated.

However, excess spilling is not necessarily a waste. The reduced effective occupancy may reduce the cost of context switching [1]. The Threshold configuration on a dual-core platform decreases the average effective occupancy over the benchmarks' entire execution time by about 25%. For `ssusan_small`, for instance, the reduction amounts to 68%, thus considerably reducing the context switch cost related to the stack cache.

5 Related Work

Prefetching data before it is needed is a common concept in computer science and particular in computer architecture design [13]. However, the vast majority of prefetching mechanisms are *only* designed to improve the average-case and thus are not suited for the use in real-time systems. The notable exception is the WCET-preserving stream prefetcher proposed by Garside and Audsley [3]. The approach avoids side effects on the content of the cache by introducing separate prefetch buffers – similar to the initial work on stream prefetching by Jouppi [6]. In addition, properties of the bus arbitration scheme are exploited to schedule "prefetch slots". The authors observe that the interference between multiple cores in the system is typically overestimated. A prefetch can thus be scheduled whenever an interference is *overdue*, while respecting the worst-case execution time of the program. The actual implementation is based on a fixed-priority scheme with a predefined blocking factor to avoid starvation. The approach provides excellent average-case improvements. However, it appears difficult to improve the WCET estimation by considering the prefetching, due to the potential interaction with all other cores in the system. Our approach does not require a separate memory structure and directly operates on the stack cache. An address prediction mechanism is also not required since addresses are a priori kown (`MT` or `LP`). We thus expect a much simpler hardware design. Instead of a fixed-priority scheme we rely on free TDM slots that are *left over* by the program. Interference from other programs or processor cores with regard to the eager memory transfers are consequently excluded. It thus appears feasible to actually improve WCET estimates by taking the eager memory transfers into consideration.

In addition to prefetching, data is also transferred from the stack cache to main memory by eager spill operations. To the best of our knowledge such a mechanism was not yet proposed in the context of time-predictable cache design. Similar ideas were, however, explored for conventional write-back cache designs and termed *eager write-back* [7].

An alternative approach is to allocate code as well as data to scratchpad memories [14]. However, scratchpad memories typically complement caches instead of replacing them. The stack cache mixes properties of both, conventional caches and scratchpads, and thus is situated in between those concepts. Due to space considerations we do not elaborate these techniques in more depth here.

6 Conclusion

We presented an elegant and simple extension of the stack cache that allows to perform memory transfers eagerly in order to reduce the latency of future stack cache control instructions. We exploit the capability to track coherent data in the stack cache using the lazy pointer (`LP`), which allows us to distinguish between the effective occupancy and the total cache occupancy. Eager filling increases the occupancy and thus may profit future

`sens` instructions, while eager spilling decreases the effective occupancy and thus may profit `sres` instructions. The interplay between effective occupancy and occupancy guarantees that the worst-case timing is not impacted. In addition, we propose to perform these eager operations in free TDM slots to avoid any interference with concurrent memory accesses.

Acknowledgements This work was supported by a grant (2014-0741D) from Digiteo France: "Profiling Metrics and Techniques for the Optimization of Real-Time Programs" (PM-TOP).

References

1 S. Abbaspour, F. Brandner, A. Naji, and M. Jan. Efficient context switching for the stack cache: Implementation and analysis. In *Proc. of the Int. Conf. on Real Time and Networks Systems*, RTNS '15, pages 119–128. ACM, 2015.

2 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.

3 J. Garside and N. C. Audsley. WCET preserving hardware prefetch for many-core real-time systems. In *Proc. of the Int. Conf. on Real-Time Networks and Systems*, RTNS '14. ACM, 2014.

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*, RTNS'13, pages 55–64, 2013.

6 N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proc. of the Int. Symp. on Computer Architecture*, ISCA '90, pages 364–373. ACM, 1990.

7 H.-H. S. Lee, G. S. Tyson, and M. K. Farrens. Eager writeback - a technique for improving bandwidth utilization. In *Proc. of the Int. Symp. on Microarchitecture*, MICRO 33, pages 11–21. ACM, 2000.

8 D. A. Patterson and J. L. Hennessy. *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann, 4rd edition, 2012.

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 Conf. on Hardware/Software Codesign and System Synthesis*, pages 99–108, 2011.

10 C. Rochange, S. Uhrig, and P. Sainrat. *Time-Predictable Architectures*. Wiley, 2014.

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

12 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 *Proc. of Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, volume 18, pages 11–21. OASICS, 2011.

13 A. J. Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, September 1982.

14 V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. WCET centric data allocation to scratchpad memory. In *Proc. of the Int. Real-Time Systems Symp.*, RTSS '05, pages 223–232. IEEE, 2005.

15 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.