Efficient Context Switching for the Stack Cache: Implementation and Analysis

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

Florian Brandner, Amine Naji
U2IS
ENSTA ParisTech
Université Paris-Saclay
{brandner, naji}@ensta-paristech

Mathieu Jan
CEA, LIST
Embedded Real Time Systems Laboratory
mathieu.jan@cea.fr

ABSTRACT

The design of tailored hardware has proven a successful strategy to reduce the timing analysis overhead for (hard) real-time systems. The stack cache is an example of such a design that has been proven to provide good average-case performance, while being easy to analyze.

So far, however, the analysis of the stack cache was limited to individual tasks, ignoring aspects related to multitasking. A major drawback of the original stack cache design is that, due to its simplicity, it cannot hold the data of multiple tasks at the same time. Consequently, the *entire* cache content needs to be *saved* and *restored* when a task is preempted.

We propose (a) an analysis exploiting the simplicity of the stack cache to bound the overhead induced by task preemption and (b) an extension of the design that allows to (partially) hide the overhead by *virtualizing* stack caches.

Categories and Subject Descriptors

F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*; C.3 [**Special-Purpose and Application-Based Systems**]: *Real-time and embedded systems*

Keywords

Program Analysis, Stack Cache, Cache-Related Preemption Delays, Real-Time Systems

1. INTRODUCTION

With the rising complexity of the underlying computer hardware, the analysis of the timing behavior of real-time software is becoming more and more complex and imprecise. The design of tailored computer architectures [20, 17, 14], and here in particular designs focusing on the memory hierarchy [13, 10, 1], thus gained considerable attention.

The stack cache of the Patmos processor [1, 16, 15] exploits the regular structure in the access patterns to stack data. Functions often operate exclusively on their local variables, resulting in spacial and temporal locality of stack accesses following the nesting of function calls. The cache can be implemented using a circular buffer using two pointers: the memory top pointer `MT` and the stack top pointer `ST`. The `ST` points to the top element of the stack and data between `ST` and `MT` is present only in the cache. The remaining data above[1] `MT` is available only in main memory. In contrast to traditional caches, memory accesses are guaranteed hits. The time to access stack data thus is constant, simplifying Worst-Case Execution Time (WCET) analysis. The compiler (programmer) is responsible to enforce that all stack data is present in the cache when needed using three stack control instructions: reserve (`sres`), free (`sfree`), and ensure (`sens`). The worst-case (timing) behavior of these instructions only depends on the worst-case spilling and filling of `sres` and `sens` respectively, which can be bounded by computing the maximum and minimum cache occupancy [8], i.e., the value of $\mathtt{MT} - \mathtt{ST}$. The cache's simple design thus reduces the analysis complexity considerably (see Section 2).

However, the simple structure of the stack cache also has drawbacks. One problem arises when multiple tasks are executed using preemptive scheduling. The two pointers only capture the cache state of the currently running task, the state of other (preempted) tasks is *lost* once `ST` and `MT` are overwritten. The data of preempted tasks might still be in the cache. However, the hardware *cannot* ensure that this data remains unmodified. Even worse, it cannot ensure that modified cache data, not yet written back to main memory, remains coherent. As a consequence the entire stack cache content has to be *saved* to main memory when a task is preempted. In addition, the stack cache content has to be *restored* before that task is resumed. This may induce considerable overhead that has to be accounted for during the analysis of a real-time system equipped with a stack cache.

The contributions of this work are: (1) a Stack Cache Analysis (SCA) technique to bound the overhead induced by the stack cache during preemption, i.e., *cache-related preemption delays* [9], and (2) a hardware extension to *virtualize* several stack caches in a shared memory, which allows us to quickly switch between these virtual caches. The preemption overhead can partially be hidden through this extension, which is profitable for the Worst-Case Response Time (WCRT) of the preempting task. This furthermore opens promising opportunities to save/restore virtual caches of preempted tasks during the execution of other tasks.

This paper is structured as follows: Section 2 provides background related to the stack cache as well as static program analysis. In Section 3, we then present our approach to analyze the cache-related preemption delays induced by the

[1]We assume that the stack grows towards lower addresses.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

RTNS 2015, November 04-06, 2015, Lille, France
© 2015 ACM. ISBN 978-1-4503-3591-1/15/11…$15.00
DOI: http://dx.doi.org/10.1145/2834848.2834861

stack cache. Section 4 is dedicated to virtual stack caches, their design and the possible scheduling opportunities that they open. The analysis and the hardware extension are evaluated in Section 5 before concluding.

2. BACKGROUND

The stack cache is implemented as a ring buffer with two pointers [1]: *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 $\mathtt{MT} - \mathtt{ST}$ (*occupancy*) represents the amount of occupied space in the stack cache, which cannot exceed the total size of the cache's memory $|SC|$, thus $0 \leq \mathtt{MT} - \mathtt{ST} \leq |SC|$. The stack control instructions manipulate the two stack pointers and initiate memory transfers to/from the cache to main memory, while preserving this equation. A brief summary is given below, details are available in [1]:

`sres` k:	Subtract k from `ST`. If this violates the equation, i.e., the cache size is exceeded, a memory *spill* is initiated to decrement `MT` until $\mathtt{MT} - \mathtt{ST} \leq |SC|$.
`sfree` k:	Add k to `ST`. If this would result in $\mathtt{MT} < \mathtt{ST}$, `MT` is set to `ST`. Memory is not accessed.
`sens` k:	Ensure that the occupancy is larger than k. If this is not the case, a memory *fill* is initiated to increment `MT` until $\mathtt{MT} - \mathtt{ST} \geq k$.

A lazy pointer (`LP`) [16] can be added as an extension to track coherent cache data, i.e., data between `MT` and `LP` is known to have the same value in the cache and in main memory. Coherent data can then be excluded from memory spill operations. Since `LP` only refers to cached data it respects the invariant $\mathtt{ST} \leq \mathtt{LP} \leq \mathtt{MT}$, which has to be preserved by the stack control instructions and requires minor modifications to stack store instructions (`sts`) [16].

The compiler manages the stack frames of functions quite similar to other architectures with exception of the ensure instructions. For brevity, we assume a simplified placement of these instructions. Stack frames are 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. We also restrict functions to only access their own stack frames.[2]

2.1 Stack Cache Analysis

As all memory accesses (`lds`/`sts`) through the stack cache are guaranteed hits, the timing behavior of the stack cache only depends on the amount of data spilled/filled by `sres` and `sens` instructions respectively. In the case of the standard stack cache this amount can be bounded by analyzing the cache's maximum/minimum occupancy ($\mathtt{MT} - \mathtt{ST}$) [8], while for lazy spilling the *effective occupancy* ($\mathtt{LP} - \mathtt{ST}$) needs to be considered [16]. The analyses rely on these definitions:

Control-Flow Graph (CFG): The CFG of a function is a directed graph $G = (N, E, r, t)$. Nodes in N represent instructions and edges in E the potential execution flow. Nodes r and $t \in N$ denote the function's unique entry and exit points respectively. Additionally, we define $Succs(n) = \{m \mid (n, m) \in E\}$, the set of immediate successors of n.

[2]Data that is larger than the stack cache or that is shared can be allocated on a *shadow stack* outside the stack cache.

(a) Code of A	(b) Code of B	(c) Code of C
(i_1^A) `func A()`	(i_1^B) `func B()`	(i_1^C) `func C()`
(i_2^A) `sres 2` $\langle 0\rangle$	(i_2^B) `sres 2` $\langle 0\rangle$	(i_2^C) `sres 3` $\langle 3\rangle$
(i_3^A) `B()`	(i_3^B) `nop` ↯	(i_3^C) `sfree 3`
(i_4^A) `sens 2` $\langle 2\rangle$	(i_4^B) `C()`	
(i_5^A) `sfree 2`	(i_5^B) `sens 2` $\langle 1\rangle$	
	(i_6^B) `sfree 2`	

Figure 1: Program consisting of 3 functions, reserving, freeing and ensuring space on the stack cache (cache size: 4).

Call Graph (CG): The CG of a program $C = (F, A, s)$ is a directed graph. Nodes in F represent functions and edges in A call sites. Node $s \in F$ is the program's entry point.

Data-flow analysis (DFA): A DFA is defined by a tuple $A = (\mathcal{D}, T, \sqcap)$, where $\mathcal{D}$ is an abstract domain (e.g., values of stack pointers), transfer functions $T_i : \mathcal{D} \to \mathcal{D}$ in T model the impact of individual instructions i on the domain, and $\sqcap : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ is a join operator. Together with a CFG an instance of an (*intra-procedural*) DFA can be formed, yielding a set of data-flow equations. For simplicity, we specify these equations though functions $\mathrm{IN}(i)$ and $\mathrm{OUT}(i)$, which are associated with an instruction i and return values over $\mathcal{D}$. The equations are finally solved by iteratively applying these functions until a fixed-point is reached [2].

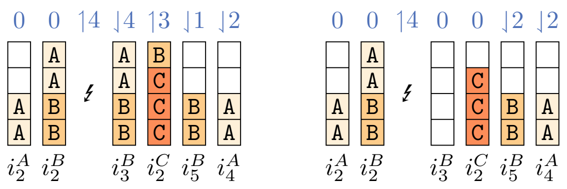
Inter-procedural analyses can be defined by additionally considering the call relations captured by the CG. In this case, additional data-flow equations are constructed modeling function calls end returns [2]. Often these analyses are *context-sensitive*, i.e., the analyses distinguish between (bounded) chains of functions calls to define calling contexts.

Example 1. Consider functions `A`, `B`, and `C` shown in Figure 1 without preemption and a stack cache whose size is 4 blocks. The stack cache analysis (SCA) derives the worst-case filling and spilling bounds at the `sres` instructions (i_2^A, i_2^B, i_2^C) and the `sens` instructions (i_4^A, i_5^B). For instance, the minimum occupancy before the `sens` i_5^B is known to be at least 1, since the preceding call to `C` spilled (evicted) at most 1 of `B`'s cache blocks. The worst-case filling at this point thus is bounded by 1 (indicated by $\langle 1\rangle$). Similarly, the maximum occupancy at the `sres` i_2^C is known to be at most 4, since `A` and `B` allocated 2 blocks respectively. The worst-case spilling thus is 3 ($\langle 3\rangle$).

3. ANALYSIS OF PREEMPTION DELAYS

Preemptive multitasking provides better schedulability for real-time systems by allowing a running task to be preempted by another task having more critical timing requirements. Task preemption involves a *context switch*, which, w.r.t. the preempted task, consists of three steps: (1) *saving* the task's execution context (registers, address space, device configurations, ...), (2) running another task, and finally (3) *restoring* the task's context. Since the traditional stack cache hardware cannot be shared by several tasks, the content of the stack cache has to be considered a part of the execution context and thus needs to be saved/restored as well. This may induce some overhead that has to be accounted for during schedulability analysis.

For traditional caches this overhead is known as Cache-Related Preemption Delays (CRPD) [9]. We will later formally define a static program analysis that allows us to bound this overhead for every program point where a pre-


0 0 ↿4 ⇃4 ↿3 ⇃1 ⇃2
A A B B A A B B A A B C C C B B A A
i_2^A i_2^B i_3^B i_2^C i_5^B i_4^A
0 0 ↿4 0 0 ⇃2 ⇃2
A A B B A A C C C B B A A
i_2^A i_2^B i_3^B i_2^C i_5^B i_4^A

(a) Simple approach. (b) Improved approach.

Figure 2: Cache states after executing the indicated instructions (below) and number of blocks transferred (above).

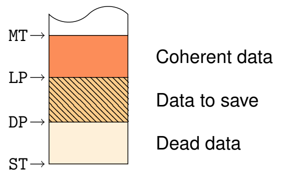


Figure 3: Partitioning of the stack: (1) Coherent data above LP (🟧), (2) data to save (▨), and (3) dead data below DP (⬜).

emption might occur. However, we will start out first with a motivating example, illustrating the underlying problem:

Example 2. Assume that a preemption occurs after instruction i_3^B (↯) of the code in Figure 1. The stack cache content then has to be *saved* and *restored* to/from main memory. A simple bound of the number of blocks that have to be transferred back and forth is given by the maximum occupancy provided by the SCA. In this example four blocks (of `A` and `B`) need to be transferred, as illustrated by Figure 2a.

This overhead can be reduced as illustrated by Figure 2b. Actually, no cache block needs to be restored here. The blocks of `A` are only accessed after returning from `B`. The **`sens`** i_4^A will automatically restore both blocks. According to our initial SCA this instruction fills 2 blocks in the worst-case, i.e., the blocks are restored without additional overhead. Similarly, the blocks of `B` are not accessed before the call and are automatically restored by the **`sens`** i_5^B, after returning from `C`. The SCA determined that at most 1 block is filled here. The overhead of transferring an additional block thus has to be added to the preemption costs. At the same time, the occupancy before **`sres`** i_2^C is now 0 instead of 4, i.e., spilling is reduced and the program thus runs faster. For this example this almost amortizes the preemption overhead.

This example illustrated that the number of cache blocks to save/restore can be reduced depending on the future use of the cached data. Our analysis, explained in the following subsections thus, is based on the notion of *liveness* – very similar to the concept of Useful Cache-Blocks (UCB) [9].

Context Saving Analysis (CSA): Clearly, data that is present in the cache, but known to be coherent with the main memory (captured by the lazy pointer `LP` [16]), can be excluded from context saving and thus reduce the preemption cost. Furthermore, some data might be excluded from saving depending on liveness, i.e., data that is not used in the future can be excluded. We will show how the analysis of dead and coherent data can be combined to reduce the number of blocks that need to be saved on a context switch.

Context Restoring Analysis (CRA): As for CSA, dead data can be excluded from context restoration. However, in many cases also live data can be excluded, e.g., when the data is spilled by an **`sres`** instruction before it is actually used or when an **`sens`** instruction would refill the data anyways. We will show that the underlying analysis problem is very similar to the liveness analysis required for CSA and, in particular, that the placement of **`sens`** instructions after calls simplifies the analysis problem.

3.1 Context Saving Analysis

The worst-case timing of saving the stack cache's context depends on the number of cache blocks that have to be transferred to the main memory. In the simplest case, all blocks potentially holding data need to be transferred, i.e., the maximum occupancy provided by SCA is a safe bound. However, not all the data present in the stack cache actually needs to be considered. Inspired by the lazy pointer (`LP`) [16], we define a pointer that tracks dead data. Due to the structure of typical stack frames, dead data usually resides at the bottom of the stack, i.e., right above `ST`. We thus introduce a virtual marker tracked by the analysis (but not realized as a hardware register), the dead pointer (`DP`):

Definition 1. The dead pointer (`DP`) is a virtual marker tracking dead data, such that $\texttt{ST} \leq \texttt{DP} \leq \texttt{MT}$. Data below `DP` is considered dead, while date above `DP` is potentially live.

The `LP` and `DP` define a partitioning of the stack cache's content into three distinct regions shown in Figure 3. Data above `LP` is coherent and thus can be ignored during context saving. Similarly, data below `DP` is known to be dead and can safely be ignored. Only the remaining data, between `DP` and `LP`, actually needs to be transferred to main memory.

As for the traditional SCA, only the pointers' relative positions w.r.t. `ST` need to be known. The results of the `LP` analysis, which requires minor modifications to the original SCA, can directly be reused for the context saving analysis. It is thus not further explained (refer to [16]).

The analysis of the `DP` is based on typical liveness analysis, i.e., a value is said to be live when it is used by a subsequent load (**`lds`**) and is considered dead immediately before a store (**`sts`**). Our analysis is a function-local, backward data-flow analysis, conservatively tracking the lowest possible position of the `DP` relative to `ST`, i.e., $\min(\texttt{DP} - \texttt{ST})$. Whenever a **`lds`** is encountered, it must be ensured that `DP` is below its effective address relative to `ST` (`EA`). A **`sts`**, on the other hand, might push the `DP` upward as the overwritten data is dead immediately before the store, since the analysis proceeds backward. Finally, with regard to a function, all its data is dead immediately before its **`sfree`**. The `DP` then is at its highest possible position, i.e., the stack frame's size `k`. The following data-flow equations specify how the `DP` changes with regard to the stack frame of a function:

$$\mathrm{OUT}(i) = \begin{cases} \texttt{k} & \text{if } i = \textbf{\texttt{sfree}}\ \texttt{k} \\ \min(\mathrm{IN}(i), \texttt{EA}) & \text{if } i = \textbf{\texttt{lds}}\ \texttt{EA} \\ \mathrm{IN}(i) + 1 & \text{if } i = \textbf{\texttt{sts}}\ \texttt{EA} \wedge \texttt{EA} = \mathrm{IN}(i) \\ \mathrm{IN}(i) & \text{otherwise} \end{cases} \tag{1}$$

$$\mathrm{IN}(i) = \begin{cases} 0 & \text{if } i = t \\ \min_{s \in Succs(i)}(\mathrm{OUT}(s)) & \text{otherwise} \end{cases} \tag{2}$$

Assuming a unit cost $\widehat{c}_s$ to transfer a cache block to main memory, the overhead induced by context saving before an instruction i depends on the size of the coherent area $\mathrm{CA}(i)$ (derived from the `LP` [16]), the size of the dead area $\mathrm{DA}(i)$ (given by Equation 2), and the maximum occupancy $\mathrm{Occ}(i)$:

$$savingCost(i) = \widehat{c}_s \max(0, \mathrm{Occ}(i) - \mathrm{CA}(i) - \mathrm{DA}(i)) \quad (3)$$

Note that the size of the coherent data as well as the maximum occupancy are potentially calling-context dependent, i.e., might change with the nesting of surrounding function calls. This can easily be considered in the above equations. The costs would then, of course, also be context-dependent.

It would be possible to consider the calling-context when analyzing the dead area ($DP(i)$). Whenever all data in a function's stack frame is dead, the size of its caller's dead area can be added to $DP(i)$. However, this is rarely beneficial in practice, since all functions, except leaf functions not calling other functions, store the return address on the stack. Details on inter-procedural analysis are thus omitted.

3.2 Context Restoring Analysis

Similar to context saving, the time required to restore a task's stack cache context depends on the number of cache blocks that need to be transferred back from main memory to the cache. As before, a simple solution would be to transfer all the blocks potentially holding data, which is again bounded by the maximum occupancy.

However, as shown in Example 2, not all cache blocks have to be restored. We can distinguish the following cases, as illustrated by Figure 4: (1) cache blocks containing dead data only (given by Equations 1 and 2), (2) blocks potentially containing live data that have to be restored, and (3) blocks that are restored by a subsequent `sens`. Since only subset of the cache blocks are restored the occupancy after a preemption is usually reduced. This may reduce the spill costs of subsequent `sres` instructions. The analysis thus has to consider another case: (4) potential gains due to reduced spilling. Case (1) and (2) can be handled by function-local analyses explained in Section 3.2.1, while case (3) and (4) require inter-procedural analyses covered in Section 3.2.2.

3.2.1 Local Restore Analysis

Dead data can simply be excluded form the memory transfer as explained before. However, in contrast to context saving, space has to be allocated on the stack cache in order to guarantee that subsequent memory accesses succeed. The allocation is only needed when dead data exists, i.e., $\mathrm{DA}(i)$ is non-zero. Even then, the operation only requires an update of `MT`, which can be performed in constant time ($\widehat{c}_a$):

$$allocationCost(i) = \begin{cases} \widehat{c}_a & \text{if } \mathrm{DA}(i) \neq 0 \\ 0 & otherwise \end{cases} \quad (4)$$

Blocks containing live data have to be restored and thus transferred from main memory. Note that this is only needed when the data is *not* restored by an `sens` before the data is access. We thus have to keeps track of live data, while accounting for the impact of `sens` instructions:

Definition 2. The restore pointer (`RP`) is a virtual marker tracking potentially live data in the stack cache, i.e., `ST` $\leq$ `RP` $\leq$ `MT`. Data below the `RP` is potentially live and not guaranteed to be restored by a subsequent `sens` instruction.

An interesting observation is that `sens` instructions are placed after every function call and that functions are assumed to only access their own stack frames. This simplifies context restoration, since only stack data of the function where the preemption occurred has to be restored. The stack frames of the calling functions are then automatically restored by their respective `sens`. The computation of the associated overhead is explained in Section 3.2.2.

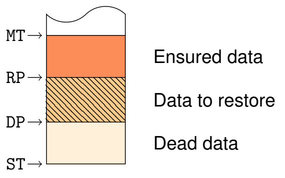


Figure 4: Partitioning of the stack: (1) data restored by `sens` (🟧), (2) data to restore (▨), and (3) dead data (⬜).

The analysis of the `RP` is a function-local, backward analysis that tracks the highest possible position of the pointer relative to `ST` (`RP` $-$ `ST`). In order to simplify the handling of dead data and `sens` instructions, we assume that both `lds` and `sts` instructions push the `RP` upwards, i.e., increase the amount of data that has to be restored, even if the data overwritten by an `sts` might be dead at this point, e.g., when `DP` $<$ `RP`. Note, however, that there is no strict ordering between the `DP` and the `RP`, i.e., it might happen that `DP` $>$ `RP`. In particular, this is the case when an `sens` is encountered. In this case the `RP` is set to its lowest possible position (0) This simply means that no data needs to be restored in case of a preemption that occurs immediately before that ensure. The `sens` will automatically reload the required data when the task gets resumed. The following equations capture the relative position of the `RP` relative to `ST`:

$$\mathrm{OUT}_{RP}(i) = \begin{cases} 0 & \text{if } i = \texttt{sens k} \\ \max(\mathrm{IN}_{RP}(i), \texttt{EA}) & \text{if } i = \texttt{lds EA} \\ \max(\mathrm{IN}_{RP}(i), \texttt{EA}) & \text{if } i = \texttt{sts EA} \\ \mathrm{IN}_{RP}(i) & \text{otherwise} \end{cases} \quad (5)$$

$$\mathrm{IN}_{RP}(i) = \begin{cases} 0 & \text{if } i = t, \\ \max_{s \in Succs(i)}(\mathrm{OUT}_{RP}(s)) & \text{otherwise} \end{cases} \quad (6)$$

Assuming unit costs $\widehat{c}_r$ to transfer a cache block from main memory, the cost of restoring the live data of the stack cache depends on the size of the dead area ($\mathrm{DA}(i)$, Equation 2) and the size of the restore area ($\mathrm{RA}(i)$, Equation 5):

$$transferCost(i) = \widehat{c}_r \max(0, \mathrm{RA}(i) - \mathrm{DA}(i)) \quad (7)$$

It remains to account for the additional transfer costs at `sens` instructions in the current function. This can be expressed by the maximum number of blocks that are *not* filled by a subsequent `sens`, which are derived from the `sens`'s argument `k` and the filling bounds $\langle b \rangle$ from the original SCA. These values are propagated upwards through the CFG:

$$\mathrm{OUT}_{FL}(i) = \begin{cases} \texttt{k} - \texttt{b} & \text{if } i = \texttt{sens k } \langle \texttt{b} \rangle \\ \mathrm{IN}_{FL}(i) & \text{otherwise} \end{cases} \quad (8)$$

$$\mathrm{IN}_{FL}(i) = \begin{cases} 0 & \text{if } i = t, \\ \max_{s \in Succs(i)}(\mathrm{OUT}_{FL}(s)) & \text{otherwise} \end{cases} \quad (9)$$

The additional overhead induced by the next `sens` within the current function can then be computed from the number of cache blocks that are not filled ($\mathrm{FL}(i)$, Equation 8) and the number of the blocks that were explicitly restored, i.e., the size of the restore area ($\mathrm{RA}(i)$, Equation 5):

$$ensureCostLocal(i) = \widehat{c}_r \max(0, \mathrm{FL}(i) - \mathrm{RA}(i)) \quad (10)$$

3.2.2 Global Restore Analysis

The additional costs caused by **sens** instructions of other functions can be derived from the longest path in a weighted CG from the program's entry node to the current function. The edge weights in the graph are the number of blocks that are *not* filled by the **sens** associated with the corresponding call site, which is given by FL(i) (Equation 9) of the site's `call` instruction. Note that this problem is very similar to the computation of the maximum displacement of the original SCA [1]. However, the length of the path is bounded: (1) by the size of the stack cache and (2) by the minimum amount of stack data remaining in the stack cache after returning from the function, i.e., $\max(0, |SC| - D(i))$ where $|SC|$ denotes the stack cache size and $D(i)$ the function's maximum displacement. The latter case is particularly interesting, since no computation is required when the function's displacement is larger than the stack cache size. The length of the path and the restoration costs then simply become 0. Given the length of such a path FLG(f) for function f, the costs induced at other functions is:

$$\mathit{ensureCost}(f) = \widehat{c}_r \mathrm{FLG}(f) \tag{11}$$

Due to the lazy restoration of the stack cache context, the worst-case occupancy after a preemption is generally lower than during regular execution. This, not only reduces the time necessary to restore the stack cache, it might also reduce spilling at subsequent **sres** instructions. The reduced spilling, in turn, partially amortizes the preemption costs. The cost gain can be determined by combining the minimum occupancy at call sites and the minimum displacement of the called function(s). Both are readily provided by the standard SCA. Assuming a stack cache size $|SC|$, the minimum spilling without preemption at a call instruction i with minimum occupancy mOcc(i) and an associated minimum displacement $d(i)$ (note: $d(i) \leq D(i)$ from above) then is:

$$\mathit{minSpill}(i) = \max(0, \mathrm{mOcc}(i) + d(i) - |SC|) \tag{12}$$

The minimum spilling with preemption is computed in a very similar way. However, the minimum occupancy is lower due to the lazy restoration of the stack cache's content. A simple bound of the minimum occupancy, that is sufficiently precise in practice, is the size of the current function's stack frame, i.e., the argument of the stack control instructions $\mathtt{k}$:

$$\mathit{minSpillPr}(i) = \max(0, \mathtt{k} + d(i) - |SC|) \tag{13}$$

The minimum gain from the reduced spilling at a call site (i) is then given by:

$$\mathit{siteGain}(i) = \max(0, \mathit{minSpill}(i) - \mathit{minSpillPr}(i)) \tag{14}$$

Finally, this minimum gain is propagated upwards through the CFG from call sites using the following equations:

$$\mathrm{OUT}_{GN}(i) = \begin{cases} \max(\mathrm{IN}_{GN}(i), \mathit{siteGain}(i)) & \text{if } i = \mathtt{call} \\ \mathrm{IN}_{GN}(i) & \text{otherwise} \end{cases} \tag{15}$$

$$\mathrm{IN}_{GN}(i) = \begin{cases} 0 & \text{if } i = t, \\ \min_{s \in Succs(i)}(\mathrm{OUT}_{GN}(s)) & \text{otherwise} \end{cases} \tag{16}$$

3.2.3 Context Restore Costs

The total context restoration costs are then bounded by accumulating the individual costs for space allocation, the explicitly transfer of cache blocks, and the implicitly transfer of cache blocks at **sens** instructions (locally and globally). In addition, the preemption costs are partially amortized by the reduced spilling at **sres** instruction ($\mathit{reserveGain}(i)$ given by Equation 15). Note that $f(i)$ denotes the function containing instruction i:

$$\begin{aligned} \mathit{restoreCost}(i) = \; & \mathit{allocationCost}(i) + \mathit{transferCost}(i) \\ & + \mathit{ensureCostLocal}(i) \\ & + \mathit{ensureCost}(f(i)) \\ & - \mathit{reserveGain}(i) \end{aligned} \tag{17}$$

Example 3. Consider again the preemption point at instruction i_3^B in the code shown in Figure 1. The context restore analysis first determines the minimal/maximal offset of the `DP` and `RP` with regard to `ST` respectively (Equations 2 and 6). Both offsets are 0 (due to the absence of **lds** and **sts** instructions in the code, which are omitted for brevity, i.e. $\mathit{allocationCost}(i_3^B) = 0$). Likewise, the number of cache blocks that have to be transferred to the cache can be bounded by 0 ($\mathit{transferCost}(i_3^B) = 0$). However, the function's cache blocks are later restored by an **sens** (i_5^B). The instruction restores one block for free, as indicated by the bound $\langle 1 \rangle$. The analysis thus propagates the number of blocks that are *not* restored ($2-1=1$) upwards through the CFG (Equation 9). The implicit transfer of a cache block thus has to be accounted for in the context restoration costs ($\mathit{ensureCostLocal}(i_3^B) = 1$).

It remains to perform the inter-procedural analyses. First, a weighted call graph is constructed as shown in Figure 5, where the edge weights are derived from Equation 9 at the corresponding call sites. The longest path to function `B` from the program's entry pointer (`A`) has length 0, i.e., all cache blocks of calling functions are restored for free. The same result could have been computed from `B`'s maximum displacement (5), which exceeds the cache size (4). Thus, we have $\mathit{ensureCost}(i_3^B) = 0$.

Second, the gain due to the reduced spilling at the function call to `C` (i_4^B) has to be analyzed. The minimum occupancy before the call, provided by the standard SCA, is 4, while the maximum displacement is 3. The minimal spill is 3 blocks during a regular execution (Equation 12), while in the case of a preemption only a single block is spilled (Equation 13). The minimum gain propagated to the preemption point (Equation 16) thus evaluates to 2 (cf. Equation 14 and $\mathit{reserveGain}(i_3^B) = 2$). Thus, the restore cost is negative, preempting the program at this point makes it running faster, as informally presented in Example 2.

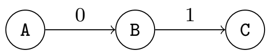

$$\begin{tikzcd} \mathtt{A} \arrow[r, "0"] & \mathtt{B} \arrow[r, "1"] & \mathtt{C} \end{tikzcd}$$

Figure 5: Weighted CG of the code in Figure 1 used to bound the additional transfer costs at **sens** instructions of other functions.

3.3 Discussion

Similar to the original SCA, which determines the worst-case filling and spilling of the stack cache control instructions, the analysis proposed here mostly operates locally on individual functions. This reduces the computational complexity (context-sensitivity is avoided) and simplifies the efficient analysis of large programs (e.g., through parallel analysis). Inter-procedural information is modeled through longest path problems on the CG, which is much smaller than a corresponding inter-procedural CFG. As real-time software usually avoids recursion, these computations are very efficient (linear in the size of the CG). Also note that the function-local data-flow analyses usually ignore `sres` and `sfree` instruction. Consequently, the computed results do not apply for code before an `sres` as well as after an `sfree`. This is not an issue, since the correct information can be derived from the calling functions, i.e., code before/after the first/last stack control instruction in a function is logically considered to be part of the immediate caller.

The presented analysis sometimes trades efficiency and simplicity for precision. This is particularly the case w.r.t. the gain stemming from reduced spilling. Currently, only the gain from a single call in the current function is considered. This is overly pessimistic, since several of the called functions and also functions called after returning from the current function might profit. However, the accumulation of gains from successive function calls cannot be modeled using traditional data-flow analysis. More precise bounds would thus require alternative techniques.

The information provided by the analysis is very rich and, as such, probably is not immediately suited to be exploited by a task scheduler (e.g., in the form of look-up tables). We consider techniques that make use or enable the use of this information to be out of scope for this work.

4. VIRTUAL STACK CACHES

The previous section presented the timing analysis of context switching assuming a single stack cache. In this section, we show an issue when integrating this timing analysis into a schedulability analysis and propose virtual stack caches to solve this issue. Finally, we show how to integrate them within a processor and present preliminary discussion regarding possible opportunities for task scheduling.

4.1 Schedulability Analysis Issue

When a preemption occurs, the content of classical data caches will be updated as the preempting task perform memory accesses, i.e. when misses occur. When the preempted task is resumed, an additional CRPD must be accounted for in its WCET due to data blocks that were evicted by the preempting task. A response time analysis integrating CRPDs can then be performed, as in [4] for instance. When considering a stack cache, the stack data of the preempted task must be saved before the stack cache of the preempting task can be restored. Only then, the preempting task can start execution. The stack cache's CRA, presented in the previous section, bounds the restoration costs, which have to be added to the preempted task's WCET similar to standard CRPDs. However, the costs determined by CSA have an immediate impact on the preempting task (similar to effects of a standard cache with a write-back policy [20]). Let us illustrate this through an example.

Example 4. Figure 6 shows a high-priority task τ_1 preempting a low-priority task τ_2. Before τ_1 can start execution, the content of τ_2's stack cache is saved to main memory. Thus, τ_1 has to wait until the memory transfer of the low-priority task (represented by the red block on the left in Figure 6) is completed. When τ_1 finishes, τ_2's stack cache content is brought back from main memory (represented by red block on the right) causing another delay when τ_2 is resumed. τ_1 therefore suffers from an additional delay that depends on the amount of data of τ_2 computed by CSA to be transfered.

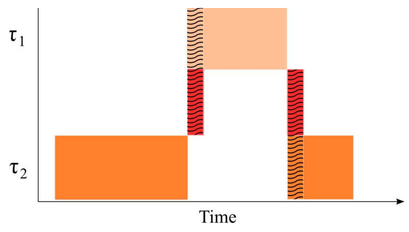


Figure 6: Context switch overhead caused by preemption.

Apart from an increased WCRT of high-priority tasks, this delay can also vary heavily and cause undesirable jitter, depending on the preempted tasks and their respective CSA results. A similar issue only exist in caches with a write-back policy, which are not recommended for real-time systems [20]. While the stack cache simplifies the WCET analysis of a single task, this additional CRPD, that depends on the preempted tasks, complicates the WCRT analysis when using preemptive schedulers.

4.2 Virtual Stack Cache Design

To mitigate this problem, we propose to allocate a Virtual Stack Cache (VSC) to each task, i.e. each task has its own dedicated VSC. These caches are then mapped to a fast local scratchpad memory, shared among all these tasks (i.e., running on the same physical core). For now, let us assume that all the VSCs of a system fit into the underlying memory. The context saving and restoration costs are then completely eliminated. It suffices to retrieve the location where the VSC of the preempting task is mapped, which, in the simplest case, means fetching two pointers. The scheduling issue pointed out above, simply disappears along with the preemption overhead.

The hardware, naturally, has to keep track of the VSC locations at the processor-level given by an offset (`vscOffset`) and size (`vscSize`) pointer. Each task's stack area is then located in the range $[\texttt{vscOffset}, \texttt{vscOffset} + \texttt{vscSize}]$ in the underlying memory. On a context switch, only the `vscOffset` and `vscSize` pointers have to be stored, while no transfer of stack data is required.

Scratchpad memories are typically small and expensive, which limits the number of VSCs that can be stored simultaneously under a static partitioning. It also appears to be a waste of resources to keep inactive stack data in the scratchpad. Clearly, a more efficient solution is needed that allows to off-load VSCs to off-chip memory when the stack data is not needed. VSCs lend themselves for such a (semi-)dynamic scheme, since their mapping can freely be updated (even more, the size of VSCs could be updated dynamically). We thus envision that VSCs are combined with an arbitration mechanism that allows the system's task scheduler to dynamically save and restore the VSCs of inactive tasks to/from main memory. Since the task scheduler is controlling this process, we believe that a time-predictable solution is feasible, which as a side-effect improves the system's resource utilization.

4.3 Scheduling Opportunities

Managing the stack cache using a dynamic partitioning strategy clearly requires to implement prefetching techniques, for memory transfers to/from the stack caches, based on future decisions taken by the underlying scheduler. A timing constraint exists for these memory transfers: they must be completed before the tasks are executed so that load and store accesses to the stack cache are guaranteed hits.

To illustrate this, let us assume a static scheduling and that at most two VSCs can be mapped in the stack cache. While some task τ_i is running, the VSC of another task τ_j could be filled, so that when τ_j is scheduled, its stack data is available. Then, while τ_j is running, the VSC of τ_i can be spilled before the stack cache of the next task to be scheduled is filled. Using this approach, the CRPD issues for the stack cache can be avoided as memory transfers are performed in the background, while the processor is running a task. The results of the CSA and CRA analyses can be used to bound the time needed to perform memory transfers and ensure that VSCs are available before tasks are executed. The analysis results can furthermore be used to determine those program points where preemption overhead is the smallest.

This approach depends on the memory bandwidth that is available for performing such memory transfers. In a single-core system this usually depends on the number of free bus slots. Multi-core systems, on the other hand, often use Time Division Multiplexing (TDM), where each processor accesses main memory during a dedicated time slot (via a network or bus). VSCs then can be transferred to/from main memory in unused TDM slots. Section 5.4 presents an evaluation of the average-case memory bandwidth that is available when using a TDM approach. However, we leave a formal timing analysis and its integration within extensions to the task model for the schedulability analysis as future work.

Note that a dynamic scheduling can be used in conjunction with a dynamically partitioned stack cache. However, this would probably result in wasting memory bandwidth. Indeed, each time the scheduling structures are updated, a different task τ_j than the one whose VSC is currently being prefetched (τ_i) may be identified as being the next task to be executed. Thus, the VSC of τ_j must be transferred, making the previously speculatively prefetched stack cache of τ_i useless. The challenge is then to provide off-line guarantees for the memory transfers related to the stack caches. Finally, if the memory transfers associated to the VSC of a task cannot be guaranteed to be finished before the task activation, the order of memory transfers could be optimized to start with data that is first used by the task. However, this requires a model of patterns of memory accesses performed by the tasks. The bet is that sufficient time intervals are available to finish memory transfers of data before they are actually used by the tasks.

4.4 Optimized Stack Caches Partitioning

Let us now discuss some optimizations related to the partitioning of the virtual stack caches. When more than two VSCs can be mapped in the memory underlying the stack cache, it would be interesting to study how to allocate, in the time and space domains, the VSCs in order to minimize their memory bandwidth consumption. For instance, the VSC of most often executed task could be statically allocated, while dynamically allocating the VSC of other less frequent tasks.

It would also be interesting to tune the partitioning of the stack cache based on the priority levels of tasks, e.g., in the presence of permanent hardware faults. For instance, when a fault is detected in a memory block holding a VSC of a given task τ_i, three options would be possible. First, the content of the VSC of τ_i could be remapped to a different location of the memory. If not possible, the second option is then to reduce the size of the VSC of tasks whose priorities are lower than τ_i. Finally, the last option is to remove one or several VSCs of tasks whose priority is lower than τ_i. This may have an impact on the WCET of these lower priority tasks depending on whether stack cache filling/spilling can still be hidden or not. However, the same performance is guaranteed for the higher priority tasks.

5. PRELIMINARY EXPERIMENTS

This section presents an evaluation of the preemption costs associated with the stack cache, covering both results from the static analysis (Section 3) and measurements related to the proposed hardware extension to virtualize the stack cache (Section 4).

The benchmarks are taken from the MiBench benchmark suite [6], which covers a large variety of small- and medium-sized programs typically found in embedded systems. The programs were compiled with optimizations enabled (`-O2`) using the LLVM[3] compiler for the Patmos processor [17]. The hardware of the platform is configured with a 64KB, 4-way set-associative data cache using LRU replacement, and a write-through policy (recommended for real-time systems [20]). Code is cached by a 64KB method cache [17] with LRU replacement and 32 code block entries. The stack cache is 256b small and uses a lazy pointer [16]. Note that varying the stack cache size between 256b and 1KB showed little impact on the results obtained. The global memory is assumed to have a moderate latency of 21 cycles. Memory transfers are performed in bursts of 32b. The cache line size of all caches matches the memory's burst size. Note, the stack control instructions still operate in words, i.e., stack frame sizes have to be word-aligned, while memory transfers are performed in bursts.

The analysis is implemented in the Patmos backend of the LLVM compiler, and operates on the final machine-level code, right before code emission. The reported numbers represent a simplified cost model, consisting of the number of bytes that have to be saved or restored during context switching at the beginning of basic blocks. Measurement are performed using the cycle-accurate Patmos simulator assuming a multi-core platform with either 2, 9 or 16 processors. In this case, the memory accesses are arbitrated using *Time-Division Multiplexing* (TDM). Each core then has a fixed, pre-assigned TDM slot at its disposal.

5.1 Context Restore Analysis

The context restore analysis shows remarkable results over all benchmark programs considered. The main benefit stems from the fact that the `sens` instructions are placed after each function call. Many of these instructions restore a part of the stack cache context for free, leading to considerable reductions in comparison to a full restoration based on maximum occupancy. In total, the benchmarks consist of 114257 basic blocks of which, 113596 (99.4%) show an improvement. In the mean, over all benchmarks, the improvement is 4 fold (min. 3x, max. 7x per benchmark). Figure 7 nicely illustrates these improvements. An unoptimized, full restoration

[3]http://www.llvm.org/

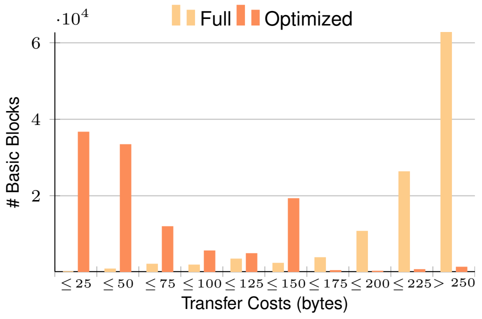


Figure 7: Histogram of transfer sizes (in bytes) for context restoration at basic blocks using max. occupancy (Full) and our approach (Optimized). Lower is better.

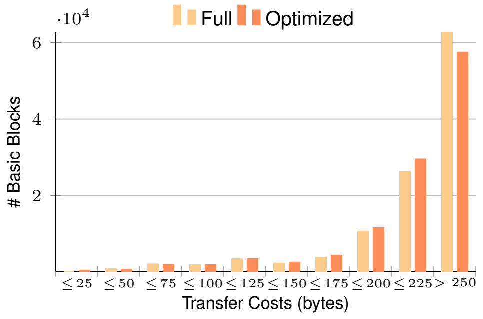


Figure 8: Histogram of transfer sizes (in bytes) for context saving at basic blocks using max. occupancy (Full) and our approach (Optimized) from Section 3. Lower is better.

typically reloads (250b or more), while our optimized approach typically only reloads up to 50b (with another peak between 125b and 150b). In many cases no explicit memory transfer is needed at all (49627 or 43.4%). Even a few cases can be observed where the restoration cost becomes negative (1968 or 1.7%), i.e., the program runs faster since the total transfer size to restore the cache content is smaller than the gain due to reduced spilling (cf. Equation 15). For 4204 basic blocks a non-zero gain due to reduced spilling was found (3.7%). Recall that this number, and the magnitude of the gains, are most likely grossly under estimated by our analysis as noted in Section 3.3.

Due to the fact that the analysis operates on a very simple domain (integers) and usually only considers individual functions, the analysis time itself is negligible. Also the inter-procedural aspects of the analysis appear to scale well. This particularly applies to the longest path search on the CG required to determine the worst-case restoration cost of `sens` instructions of other functions (Section 3.2.2). Over all benchmarks only 7 functions out of 1428 require a potentially time-consuming longest path search in a strongly connected component of the CG. For 771 the length of the path is known to be 0 due to the maximum displacement provided by the standard SCA. All other functions are in non-cyclic regions of the CG, which allows us to apply dynamic programming in order to determine the longest path.

5.2 Context Saving Analysis

Despite the fact that the context saving analysis does not account for inter-procedural effects, it shows consistent improvements over all benchmark programs considered. From 114257 basic blocks in the benchmarks 18941 (16.6%) show a reduction in the context saving overhead. However, the reductions are moderate, as can be seen in the histogram of Figure 8. The transfer size is improved by 8.2% in the mean over all benchmarks (min. 4%, max. 17% per benchmark), resulting in a moderate shift in the histogram (from the right to the left). These results are hardly surprising, since the data of all functions currently holding data in the stack cache has to be saved. The reductions are thus much smaller than for the context restore analysis. It is evidently much harder to eliminate the context saving overhead, which unfortunately can have an immediate impact on the WCRT of other tasks. The VSCs, introduced in Section 4, thus appear to be an important instrument to limit this impact.

5.3 Hardware Implementation

The virtual stack cache extension, which aims at partially hiding the preemption cost associated with the stack cache, is also evaluated within the Patmos platform. For this we adapted the platform's hardware model by introducing two new special registers, by modifying the processor's bus connections, and by updating the 5-stage processor pipeline.

Figure 9a shows Patmos' original memory subsystem, integrating a single stack cache (S$), an instruction cache (I$), a data cache (D$) and a local scratchpad memory (LM). Separate buses are used to communicate between the processor core (CPU) and the caches on one side and the local memory on the other side. All caches are connected to a shared global memory (MEM), which is arbitrated using TDM in case of a multi-core platform. To implement virtual stack caches, we propose to restructure the bus infrastructure, and to merge the stack cache's memory with the existing local memory. For this, we need to remove the stack cache from the bus between the CPU and the data/instruction caches 9b. However, the bus between the caches and the global memory remains untouched.

By default, the stack cache registers (`ST`, `MT`) are read in the third stage (`EX`). Therefore, we also read the two new `vscOffset` and `vscSize` registers in the `EX` stage. In addition, the address calculation of stack-cache-related loads (`lds`) and stores (`sts`) has to be updated according to Equation 18. The computations are done in the 4th stage (`MW`) within the processor. Accesses to a virtual stack cache thus do not differ from regular accesses to the local memory.

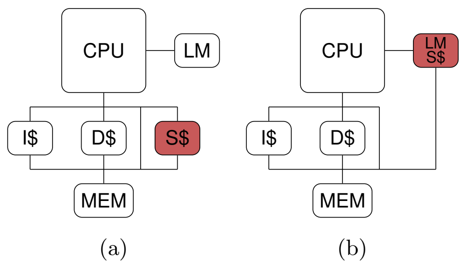


Figure 9: Block diagram of the memory subsystem components of a processor, (a) using a normal stack cache, (b) using a stack cache merged with a scratchpad memory.

Benchmark	2 cores	9 cores	16 cores	Benchmark	2 cores	9 cores	16 cores
basicmath-tiny	24.58	4.33	1.86	bitcnts	99.63	98.21	96.58
cjpeg-small	32.49	4.18	2.10	crc-32	19.75	0.00	0.00
csusan-small	54.12	15.42	8.73	dbf	15.67	0.07	0.00
dijkstra-small	18.42	0.29	0.13	djpeg-small	14.14	0.89	0.01
drijndael	16.63	0.18	0.00	ebf	15.70	0.07	0.01
erijndael	14.36	0.14	0.00	esusan-small	67.24	27.04	15.93
fft-tiny	41.89	10.89	5.41	ifft-tiny	41.55	10.84	5.30
patricia	26.68	4.58	2.09	qsort-small	12.44	0.35	0.11
rawcaudio	59.27	1.30	0.71	rawdaudio	58.20	8.10	4.74
say-tiny	66.47	25.06	11.92	search-large	10.55	0.81	0.02
search-small	9.59	0.72	0.00	sha	34.87	4.05	2.05
ssusan-small	86.96	58.74	44.23	–	–	–	–

Table 1: Percentage of free slots w.r.t. the total number of TDM slots required to execute the benchmarks on multi-cores.

$$\texttt{addr} = ((\texttt{ST} + \texttt{EA})\,\%\,\texttt{vscSize}) + \texttt{vscOffset} \qquad (18)$$

The impact of these modifications[4] on the hardware was evaluated using Altera Quartus II 13.1 tool suite targeting an Altera DE2-115 board. The results show an area overhead of only 1.8% (which do not reflect the gains due to the merged memories). Moreover, the processor frequency drops from 82 MHz to 81 MHz, due to the longer combinatorial path for the address computation. Our design at this stage is not optimized and distributing the combinatorial logic for the address calculation to different stages of the pipeline can improve the frequency. We thus conclude that the virtual stack caches cause very little overhead.

5.4 Unused TDM slots

Using the cycle-accurate Patmos simulator we collected statistics on the number of unused TDM (or bus) slots during the execution of a benchmark in order to perform context saving/restoration of preempted tasks along with the execution of another task. We compare three multi-core configurations with 2, 9 and 16 cores shown in Table 1.

The measurements are encouraging, in particular when the number of cores is small. For a given fixed interval of time, when the number of cores increases, the number of pre-assigned TDM slots per core decreases. This explains the drop in the percentage of free TDM slots for the 9 and 16 cores configuration. However, if we consider their number a large amount of slots are available, as the total execution time increases. Most of the benchmarks would therefore easily allow to transfer several virtual stack caches to and from main memory during their execution. Benchmarks with a low share of free slots, in particular `erijndael` and `drijndael`, can be considered outliers. These benchmarks suffer from particularly bad cache miss rates of more than 35% even with the large 64KB data cache. We do not expect such bad miss rates in a realistic system and thus conclude from these average-case measurements that it is feasible to hide context switch costs by transparently using TDM slots left unused by the currently running task. Further work is needed to judge whether these measurements actually can be translated to bandwidth guarantees and used within the schedulability analysis, as discussed in Section 4.3.

6. RELATED WORK

We briefly present analysis techniques to compute CRPD over classical caches, based on Useful Cache Blocks (UCB). UCBs are similar to the notion of liveness we use to optimize the context switch analysis of the stack cache. In [9], UCB are used to tighten the WCET estimations. First, the number of UCBs in all execution points are calculated using data-flow analysis. Then for all tasks, a preemption cost table is constructed that defines the preemption cost at each point, which depends on the number of UCBs and on the worst case visit count of each point. Based on this table and using *Integer Linear Programming* (ILP), the worst-case preemption delay of a task is calculated. The notion of definitely-cached UCB (DC-UCB) [3] was later introduced to detect cache misses that are included in both, the CRPD bound and the WCET bound. It was shown that this approach gives safe CRPDs, when combined with an upper bound of the WCET. The results show significant improvements over the original approach based on UCBs [9].

We now review some existing work using hardware support to optimize context switching. [19] and [11] introduce hardware support to optimize the context switching in real-time systems, but at the register file level. For instance, [19] uses dedicated hardware for scheduling threads in an SMT-based processor. The hardware scheduler is also able to save/restore the registers of a thread to a special on-chip memory, the *Thread Control Block* (TCB). The TCB requires two separate ports, in order to eliminate any interference from parallel accesses to the TCB from the running program and the hardware scheduler. Our work is orthogonal, as we optimize context switching at the cache-level. The use of virtual stack caches furthermore opens new opportunities such as context saving to off-chip memory, which, according to our experiments, appears to be feasible without additional hardware costs. Others, such as [18], optimize the average cost of context switching, but due to lacking predictability these methods are unsuited for real-time systems.

Treating data from the program's stack differently than the non-stack data was already proposed in [12], but for reducing dynamic energy. They introduce an implicit and an explicit stack data cache. The implicit stack data cache limits the stack data to reside in specific locations (ways) of the regular data cache. In the case of the explicit stack data cache a separate data cache is reserved for stack data. The use of standard caches makes this approach amenable to standard CRPD analysis techniques. However, the worst-case behavior for such a design was, so far, not evaluated.

7. CONCLUSION

The stack cache exploits the access patterns to stack data, which results in simpler hardware and analysis. Due to its simplicity, the stack cache cannot cache the stack data of

[4]The modulo operator is in practice a simple bit mask.

different tasks at the same time. The stack cache content thus becomes part of the task's execution context, which has to be saved/restored during context switching.

We presented a static program analysis to determine the worst-case preemption costs associated with the stack cache during context switching. The analysis is composed of several smaller, function-local data-flow analyses. Inter-procedural effects are handled through variants of the longest path problem. Experiments showed that the analysis complexity is low and that the restoration costs can be reduced heavily, since ensure instructions (`sens`), placed after function calls, often restore the cache context for free. However, the context saving costs appear difficult to eliminate.

To mitigate this problem, we proposed to *virtualize* the stack cache. Several virtual caches of different tasks can then be stored in a large local scratchpad memory, which allows us to quickly switch from one virtual cache to another. The virtual caches of preempted tasks can, furthermore, be saved/restored in parallel with the execution of another task by using unused TDM/bus slots to access the off-chip main memory. Measurements indicate that typical programs easily allow the transfer of several stack caches.

Future work includes using the results of the CRPD of the stack cache to bound the time needed to perform the corresponding memory transfers and integrate this amount of time, through the use of an additional task, in a schedulability analysis. Using industrial benchmarks [7, 5], we also plan to evaluate the behavior of the stack cache, in particular the dynamic partitioning of the virtual cache and the associated prefetching technique.

Acknowledgments

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).

8. 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] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2nd edition, 2006.

[3] S. Altmeyer and C. Burguiere. A new notion of useful cache block to improve the bounds of cache-related preemption delay. In *Euromicro Conf. on Real-Time Systems, 2009*, ECRTS '09, pages 109–118.

[4] S. Altmeyer, R. Davis, and C. Maiza. Improved cache related pre-emption delay aware response time analysis for fixed priority pre emptive systems. *Real-Time Systems*, 48(5):499–526, 2012.

[5] D. Chabrol, D. Roux, V. David, M. Jan, M. A. Hmid, P. Oudin, and G. Zeppa. Time- and angle-triggered real-time kernel. In *Design, Automation and Test in Europe*, DATE'13, pages 1060–1062, 2013.

[6] 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.

[7] M. Jan, V. David, J. Lalande, and M. Pitel. Usage of the safety-oriented real-time oasis approach to build deterministic protection relays. In *Symp. on Industrial Embedded Systems*, SIES'10, pages 128–135, 2010.

[8] 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.

[9] C.-G. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Trans. Comput.*, 47(6):700–713, 1998.

[10] S. Metzlaff, I. Guliashvili, S. Uhrig, and T. Ungerer. A dynamic instruction scratchpad memory for embedded processors managed by hardware. In *Proc. of the Architecture of Computing Systems Conf.*, pages 122–134. Springer, 2011.

[11] J. Mische, S. Uhrig, F. Kluge, and T. Ungerer. Using smt to hide context switch times of large real-time tasksets. In *Proc. of Conf. on Embedded and Real-Time Computing Systems and Applications*, RTCSA'10, pages 255–264, 2010.

[12] L. E. Olson, Y. Eckert, S. Manne, and M. D. Hill. Revisiting stack caches for energy efficiency. Technical Report TR1813, University of Wisconsin, 2014.

[13] 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.

[14] C. Rochange, S. Uhrig, and P. Sainrat. *Time-Predictable Architectures*. ISTE Wiley, 2014.

[15] S.Abbaspour and F. Brandner. Alignment of memory transfers of a time-predictable stack cache. In *Proc. of the Junior Researcher Workshop on Real-Time Computing*. 2014.

[16] 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.

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

[18] V. Soundararajan and A. Agarwal. Dribbling registers: A mechanism for reducing context switch latency in large-scale multiprocessors. Technical report, 1992.

[19] E. Tune, R. Kumar, D. M. Tullsen, and B. Calder. Balanced multithreading: Increasing throughput via a low cost multithreading hierarchy. In *Proc. of the Symp. on Microarchitecture*, MICRO'04, pages 183–194, 2004.

[20] 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.