Analysis of preemption costs for the stack cache

Amine Naji, Sahar Abbaspour, Florian Brandner, Mathieu Jan

Analysis of preemption costs for the stack cache

Amine Naji[1] · **Sahar Abbaspour**[2] · **Florian Brandner**[3] · **Mathieu Jan**[4]

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 was shown to provide good average-case performance, while remaining 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,

This work is an extension of the paper "Efficient Context Switching for the Stack Cache: Implementation and Analysis" by Abbaspour et al. (2015). It was partially supported by a Grant (2014-0741D) from Digiteo France: "Profiling Metrics and Techniques for the Optimization of Real-Time Programs" (PM-TOP).

✉ Amine Naji
amine.naji@ensta-paristech.fr

Sahar Abbaspour
sabb@dtu.dk

Florian Brandner
florian.brandner@telecom-paristech.fr

Mathieu Jan
mathieu.jan@cea.fr

[1] U2IS, ENSTA ParisTech, Université Paris-Saclay, Palaiseau Cedex, France

[2] Department of Applied Mathematics and Computer Science, Technical University of Denmark, Lyngby, Denmark

[3] LTCI, Télécom ParisTech, Université Paris-Saclay, Paris, France

[4] Embedded Real Time Systems Laboratory, CEA, LIST, Saclay, France

(b) preemption mechanisms for the stack cache exploiting the previous analysis and, finally, (c) an extension of the design that allows to (partially) hide the overhead by *virtualizing* stack caches.

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. Tailored computer architectures thus have been proposed based on existing hardware designs (Wilhelm et al. 2009) as well as newly designed hardware components (Schoeberl et al. 2011; Rochange et al. 2014). Due to its impact on performance the memory hierarchy received considerable attention, as shown by recent work of Reineke et al. (2011) and Metzla et al. (2011).

The stack cache of the Patmos processor exploits the regular structure in the access patterns to stack data (Abbaspour et al. 2013, 2014; Abbaspour and Brandner 2014). Functions often operate exclusively on their local variables, resulting in spatial 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 cache 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 (Jordan et al. 2013), i.e., the value of **`MT`**$-$**`ST`**. The cache's simple design thus reduces the analysis complexity considerably.

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.

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

The two main 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* (Lee et al. 1998), and (2) three preemption mechanisms that allow to efficiently exploit the analysis information during context switching. In addition, we propose 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. 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: Sect. 2 provides background related to the stack cache as well as static program analysis. In Sect. 3, we present our approach to analyze the cache-related preemption delays (CRPD) induced by the stack cache. Three different preemption mechanisms that are able to exploit the analysis information during context switching are proposed in Sect. 4. Section 5 is dedicated to virtual stack caches, their design and the possible scheduling opportunities that they open. The analysis, its relation to the proposed preemption mechanisms, and the virtual stack cache extension are evaluated in Sect. 6 before concluding.

2 Background

The stack cache is implemented as a ring buffer with two hardware registers holding pointers (Abbaspour et al. 2013): *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 $\mathbf{MT} - \mathbf{ST}$ 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. Clearly, the occupancy cannot exceed the total size of the cache's memory $|SC|$. The stack cache thus has to respect the following invariants:

$$\mathtt{ST} \leq \mathbf{MT} \tag{1}$$

$$0 \leq \mathbf{MT} - \mathbf{ST} \leq |SC| \tag{2}$$

Data that is present in the cache is accessed using dedicated stack load (`lds`) and stack store (`sts`) instructions. The frame-relative address (`FA`) of such a memory access is added to `ST` and the sum is used to index into the ring buffer, i.e., the address within the ring buffer is given by $(\mathbf{FA} + \mathbf{ST}) \mod |SC|$. Recall that the stack load and store instructions are always cache hits. The compiler (or programmer) thus has to ensure that accessed data actually is available in the cache using dedicated stack cache control instructions. More formally, it has to be ensured that $\mathbf{FA} \leq (\mathbf{MT} - \mathbf{ST}) \leq |SC|$ before every stack cache access. Note that this can easily be realized in a compiler, as explained later.

Stack cache operations The stack cache control instructions manipulate the two stack pointers and initiate memory transfers to/from the cache from/to main memory, while preserving Eqs. 1 and 2. Memory transfers, and thus also the updates of

the various pointers, are performed at the granularity of cache blocks, which can be parameterized in size. Depending on the configured block size, the memory transfers might be misaligned with the transfer size of the underlying memory system (e.g., the burst size of DRAMs). For brevity we do not cover this issue here and refer to previous work (Abbaspour and Brandner 2014) covering techniques to handle alignment issues.

A brief summary of the memory transfers associated with each control instruction is given below, further details are available in Abbaspour et al. (2013):

sres k Subtract k from ST. If this violates Eq. 2, i.e., the cache size is exceeded, a memory *spill* is initiated to decrement MT until $\mathbf{MT} - \mathbf{ST} \leq |SC|$. Cache blocks are then transferred to main memory.

sfree k Add k to ST. If this violates Eq. 1, MT is 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 memory *fill* is initiated to increment MT until $\mathbf{MT} - \mathbf{ST} \geq \mathbf{k}$. Cache blocks are then transferred from main memory.

The stack load and store 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. Modifications are not immediately propagated to the backing main memory. The stack cache's policy to handle stack store instructions thus resembles traditional *write back* caches.

Lazy pointer (LP) An extension of the original stack cache allows to track coherent cache data (Abbaspour et al. 2014). Similar to MT and ST, LP is a pointer (realized as a hardware register) that satisfies the following equation:
$\mathbf{ST} \leq \mathbf{LP} \leq \mathbf{MT}$. The additional pointer divides the stack cache content 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 known to be coherent. Coherent data can simply be excluded from memory spill operations, i.e., it can be treated as if the data were not in the cache. We thus can refine the notion of occupancy: $\mathbf{LP} - \mathbf{ST}$ denotes the *effective occupancy* of a stack cache with a lazy pointer. Accounting for the effective occupancy allows to improve the sres instruction, with only slight modifications. The sfree instruction also requires minor modifications to correctly update the LP, while the sens instruction remains unchanged. In addition, the stack store instruction (sts) has to update LP whenever coherent date may be modified (Abbaspour et al. 2014), i.e., LP is pushed upwards to ensure $\mathbf{FA} + \mathbf{ST} \leq \mathbf{LP}$.

Compiler support 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. The evicted data is consequently reloaded into the cache if needed after each call. We also restrict functions to only access their own stack frames. Data that is shared or too large can be allocated on a *shadow stack* outside the stack cache.

The restricted placement of the stack cache control instructions can be relaxed, which allows for varying frame sizes within functions in order to pass function arguments or for optimizations. The placement merely needs to be *well-formed* (Jordan et al. 2013), which means that each `sres` has to be followed by matching `sfree` instructions on all execution paths (similar to well-formed braces).

2.1 Static analysis

The worst-case behavior of the stack cache control instructions is determined using static program analysis techniques. Before going into the details of the analysis, we briefly summarize some basic notions of static analysis:

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 execution flow. Nodes r and $t \in N$ denote unique entry and exit points respectively. Additionally, we define $Succs(n) = \{m \mid (n, m) \in E\}$, the set of immediate successors of n.

Call graph (CG) The CG $C = (F, A, s)$ is a directed graph, where nodes in F represent functions and $s \in F$ the program's entry point. Edges in A are call sites, i.e., a call instruction in the CFG of the calling function. Each call is represented by a separate edge, calls via function pointers even by multiple edges.

Weighted graph A control-flow or call graph might be associated with a function $\mathcal{W}$ that assigns a weight in $\mathbb{N}$ to each node and/or edge in the graph.

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 through 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 (Aho et al. 2006).

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 (Aho et al. 2006). Often these analyses are *context-sensitive*, i.e., the analyses distinguish between (bounded) chains of functions calls to define calling contexts.

2.2 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 or 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 (Jordan et al. 2013), i.e., $\mathbf{MT} - \mathbf{ST}$, while for lazy spilling the effective occupancy (Abbaspour et al. 2014), i.e., $\mathbf{LP} - \mathbf{ST}$, needs to be considered. We will only present the former approach in the following paragraphs. We refer to this as the *standard stack cache analysis* (SCA), which proceeds in three phases:

First, the maximum/minimum displacement is computed for each function. These values indicate the largest/smallest number of cache blocks reserved during the execution of a function (including nested calls). The displacement can be used to efficiently compute the occupancy across function calls, since it allows to bound the number of blocks evicted from the stack cache. Due to the placement of stack cache control instructions,[2] the additional amount of stack space reserved at a given point in a function, w.r.t. the function entry, is constant. In our case, it simply corresponds to the value of the parameter `k` of the `sres` instruction of the enclosing function. The problem thus can be modeled as a longest/shortest path search on a weighted call graph, where the edge weight of each call site is given by the amount of stack space allocated by the *calling* function. The minimum displacement is then lower-bounded by the shortest path from a node to an artificial sink node. Likewise, the maximum displacement is upper-bounded by the longest path.

Next, the maximum filling at `sens` instructions is bounded using a function-local data-flow analysis that propagates the *maximum* displacement from call sites to the succeeding ensure instructions. In our case every `call` is immediately followed by an `sens`, rendering this analysis trivial. The maximum filling at an `sens` instruction can be bounded by computing the minimum number of cache blocks in the cache after the corresponding `call` instruction, i.e., the minimum occupancy. The minimum occupancy *after* the call has to be smaller than the occupancy before that call, since the called functions may only evict blocks from the cache. It cannot be lower than $\max(0, |SC| - D(f))$, where $D(f)$ is the maximum displacement of the called function f and $|SC|$ the stack cache size. If this bound is smaller than `k`, the argument of the `sens`, filling may occur. The maximum amount of filling can then be computed by subtracting the computed bound from `k`.

Finally, the worst-case occupancy is computed for each call site within a function using a function-local data-flow analysis. This is done by assuming a full stack cache at function entry. Subsequently, an upper bound of the occupancy is propagated to all call sites in the function, while considering the effect of other function calls and `sens` instructions. Function calls may evict stack data and thus lower the occupancy bound, depending on the *minimum* displacement of the called function (since the maximum occupancy after the call needs to be computed). The worst-case occupancy after a call cannot exceed $\max(0, |SC| - d(f))$, where $d(f)$ indicates the minimum displacement of the called function f and $|SC|$ the stack cache size. Ensure instructions on the other hand may increase the bound through filling, i.e., the worst-case occupancy after an `sens` has to be larger or equal to `k`, the ensure's argument. The maximum spilling at `sres` instructions is finally computed by propagating occupancy values through the CG, such that the maximum occupancy at the entry of a function is derived from the minimum of either (1) the maximum occupancy at the entry of its callers plus the respective sizes of the callers' stack frames (`k` of their `sres`) or (2) the worst-case occupancy bound computed by the local data-flow analysis. The latter case allows us to consider spilling of other `sres` instructions that may reduce the occupancy before

[2] This applies to the restricted placement from above as well as *well-formed* programs.

Table 1 Summary of concepts used by the traditional stack cache analysis (SCA)

Concept	Description	Analysis
Occupancy	Number of cache blocks occupied in the stack cache	–
Min. Displacement	Min. number of blocks evicted during function call	Shortest CG path
Max. Displacement	Max. number of blocks evicted during function call	Longest CG path
Worst-case Occ.	Local bound of max. Occ. assuming full stack cache	DFA+min. Disp.
Max. Filling	Min. occupancy before `sens` instructions	DFA+max. Disp.
Max. Spilling	Max. occupancy before `sres` instructions	CG+w.-c. Occ.

reaching a call site. Since the analysis operates on the call graph, fully context-sensitive spilling bounds can be computed efficiently for all functions in a program.

Table 1 summarizes the various concepts used by the traditional SCA in order to efficiently bound the maximum filling/spilling at `sens` and `sres` respectively.

Example 1 Consider functions `A`, `B`, `C`, and `D` shown in Fig. 1 without preemption and a stack cache whose size is 4 blocks. First, the displacement computation is performed on the weighted call graph shown in Fig. 2. Function `B`, for instance, may call `C` or `D`. The maximum displacement thus has to account for the stack space reserved by `B` and by these two functions, which evaluates to either $2 = 1 + 1$ (`C`) or $5 = 1 + 4$ (`D`). For functions `A`, `B`, `C`, and `D` respectively the minimum/maximum displacement values evaluate to: 4/7, 2/5, 1/1, and 4/4. Then, the maximum filling of `sens` instructions is computed. Consider, for instance, the call from `A` to `B` (A_3). Since the maximum displacement of `B` is 5, the minimum occupancy after the call evaluates to $0 = \max(0, 4 - 5)$. The corresponding `sens` instruction (A_4) consequently has to fill both of `A`'s cache blocks $(2 - 0)$, which is indicated by the bound in angle brackets $\langle 2 \rangle$. The displacement of `C` is only 1, which yields a minimum occupancy of $3 = \max(0, 4 - 1)$ after instruction B_3. The stack cache is thus large enough to hold both stack frames of `B` and `C` and no filling is needed as indicated by the bound $\langle 0 \rangle$ at instruction B_4. Next, the worst-case occupancy before call instructions is computed using a function-local data-flow analysis. The DFA determines that the worst-case occupancy before the call from `B` to `D` (B_5) is $3 = 4 - 1$, due to the call from `B` to `C`. Before all other call instructions the worst-case occupancy is 4, since no other call may lower the maximum occupancy before reaching them. Finally, the maximum occupancy is propagated through the call graph, starting at the program's entry function `A`. The maximum occupancy at the entry of `A` consequently is 0. For the call from `A` to `B` (A_3) a maximum occupancy of 2 is computed as the minimum of the call's worst-case occupancy (4) and the maximum occupancy at the entry of `A` plus the size of `A`'s stack frame $(0 + 2)$. The maximum occupancy at the entry of `D` is similarly computed from the call's (B_5) worst-case occupancy (3) and the maximum occupancy at the entry of `B` plus the size of `B`'s stack frame $(2 + 1)$. Since the size of `D`'s stack frame is equal to the stack cache size, all content of the stack cache has to be evicted by its reserve instruction. This results in a worst-case spilling of 3 blocks, as indicated by the bound $\langle 3 \rangle$ at instruction D_2. The bounds derived for the other `sres` and `sens` instructions are also indicated in angle brackets in Fig. 1.

```
(A1) func A()
(A2)  sres 2 ⟨0⟩
(A3)  call B
(A4)  sens 2 ⟨2⟩
(A5)  sfree 2
```
(a)

```
(B1) func B()
(B2)  sres 1 ⟨0⟩
(B3)  call C
(B4)  sens 1 ⟨0⟩
(B5)  call D
(B6)  sens 1 ⟨1⟩
(B7)  sfree 1
```
(b)

```
(C1) func C()
(C2)  sres 1 ⟨0⟩
(C3)  nop ↯
(C4)  sfree 1
```
(c)

```
(D1) func D()
(D2)  sres 4 ⟨3⟩
(D3)  sfree 4
```
(d)

Fig. 1 Program consisting of 4 functions, reserving, freeing, and ensuring space on the stack cache (cache size: 4). The annotations in angle brackets, e.g., ⟨2⟩, indicate the maximum filling/spilling behavior of stack cache control instructions. **a** code of A, **b** code of B, **c** code of C, **d** code of D

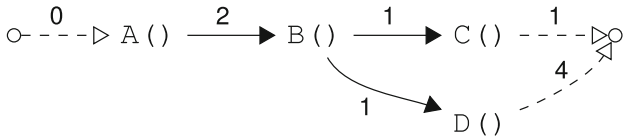

Fig. 2 A weighted call graph representing the program from Fig. 1. The edge weights indicate the amount of stack space reserved in the respective functions, and can be used to compute the minimum/maximum displacement

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, with regard to the preempted task, consists of three steps: (1) *saving* the original task's execution context (registers, address space, device configurations,…), (2) running another task, and finally (3) *restoring* the original 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 and restored as well. This may induce some overhead that has to be accounted for during schedulability analysis. For traditional caches (Lee et al. 1998) this overhead is known as CRPD. We will later formally define a static program analysis that allows us to bound this overhead for the stack cache for every program point where a preemption might occur. However, we will start first with a motivating example, illustrating the underlying problem:

Example 2 Assume that a preemption occurs right before the `sfree` instruction C_3 (↯) of the code in Fig. 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`, `B`, and `C`) need to be transferred, both, during context saving and restoration, as illustrated by Fig. 3a. This overhead can be reduced as illustrated by Fig. 3b. The stack data of `C` will be freed immediately after the preemption and thus is *dead*, i.e., the data can never be accessed after the preemption. This reduces the cost of context saving to a transfer of 3 cache blocks (of `A` and `B`) instead of 4. Also the context restoration costs are reduced. Actually, no cache block needs to be restored here. It thus suffices to re-reserve a single block on the stack cache for `C`'s

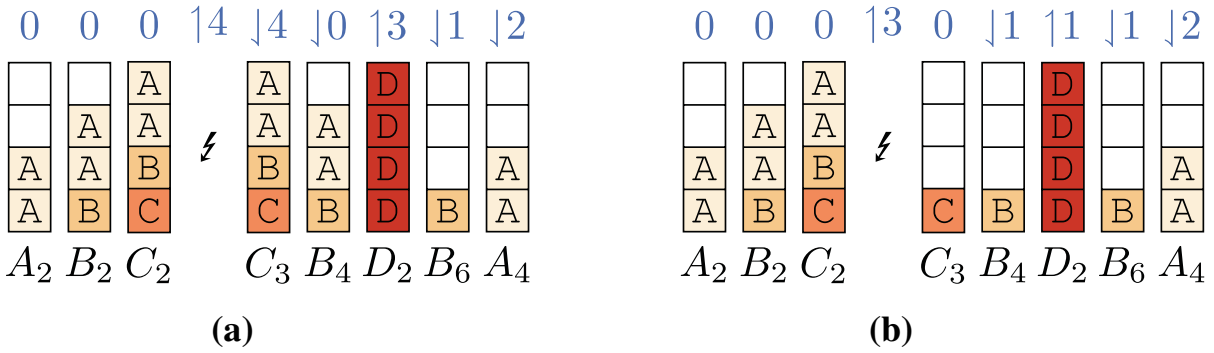

Fig. 3 Cache states after executing the indicated instructions (below) and number of blocks transferred (above). **a** simple approach, **b** improved approach

dead stack data. The blocks of `B` are only accessed after returning from `C`. The `sens` B_4 will automatically restore the necessary data. According to our initial SCA this instruction does not fill any block in the worst-case without preemption, i.e., an additional overhead to transfer `B`'s cache block needs to be accounted for as preemption cost. The cache blocks of `A` are similarly restored by the corresponding `sens` A_4. This time, the restoration will not cause any additional costs, since the standard SCA already accounts for the filling of two cache blocks. At the same time, the occupancy before the next function call to `D` is reduced from 3 to 1, since only `B`'s stack frame was actually restored. Consequently, the spilling of `D's` reserve instruction D_2 is reduced. With preemption, actually fewer cache blocks are spilled than computed by the standard SCA—thus reducing the preemption costs. In comparison to the simple approach, transferring 14 cache blocks, the transfer costs only amount to 8 blocks.

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 (Lee et al. 1998).

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`, see Abbaspour et al. 2014), 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., dead 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

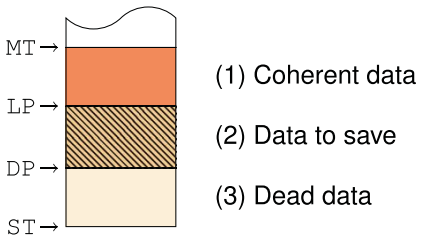

Fig. 4 Partitioning of the stack: *(1)* coherent data above LP, *(2)* data that actually needs to be saved, and *(3)* dead data below DP

blocks potentially holding data need to be transferred, i.e., the maximum occupancy provided by SCA is a safe bound. However, not all data in the stack cache actually needs to be considered, as illustrated by the motivating example.

The lazy pointer (Abbaspour et al. 2014) readily allows to track coherent data that can be ignored during context saving, i.e., data known to have the same value in the cache and in main memory. Since the LP is implemented as a hardware register it can immediately be exploited by any context switching mechanism. Also the proposed analysis is immediately applicable and can be reused for the Context Saving Analysis. We thus do not provide details regarding the analysis here and simply assume that its results are available for the final cost computation of the CSA (see below).

Another class of data, that can be ignored during context saving, is dead data, i.e., data that will never be accessed by the program. Data in the stack cache may become dead starting from a given program point due to two reasons: (1) data that will be overwritten by an `sts` instruction (without an intermittent `lds`) in all executions or (2) data that is freed by an `sfree` (without an intermittent `lds`) in all executions. Inversely, data that is potentially used by a subsequent `lds` instruction has to be considered live.

Note that individual bytes on the stack cache might be live or dead depending on the actual usage of each individual byte, which would necessitate an analysis that is able to track individual bytes. However, due to the structure of typical stack frames generated by the compiler, we observe that dead data usually resides at the bottom of the stack, i.e., right above ST. The following analysis takes advantage of this fact in order to simplify the analysis complexity.

Inspired by the LP, we define a virtual pointer that allows us to track blocks of dead data residing right above ST. This virtual marker is only used by the analysis and is *not* realized as a hardware register. We call this virtual marker the *dead pointer* (DP):

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

The lazy pointer (LP) and the dead pointer (DP) define a partitioning of the stack cache's content into three distinct regions shown in Fig. 4. 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. Note that this model only allows to detect dead data at the bottom of the stack cache—which we observed to be the usual case in the code generated by the compiler. The obtained results are thus a conservative approximation, i.e., more dead data might actually be present in the cache, which is not detected by the analysis and thus cannot be exploited. Likewise, more coherent

data might be present below the position of the LP determined by the analysis. Both of these cases may lead to an over-estimation of the worst-case cost determined by the CSA, but do not compromise the analysis' correctness. Also note that this approach simplifies the actual context saving, since only a contiguous block of data needs to be transferred.

The analysis of the DP is based on a typical *backward* 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), or, in the case of the stack cache, an sfree. As for the traditional SCA, only the relative position of the DP with regard to ST needs to be known, which further simplifies the CSA. Our analysis is a function-local, backward data-flow analysis, conservatively tracking the lowest possible position of the DP relative to ST, i.e., for each program point the minimum value min($\mathbf{DP} - \mathbf{ST}$) is computed over all possible executions of the analyzed program. This ensures that the analysis is conservative and only considers the least amount of dead data actually in the cache for the cost computation.

As indicated above, only three kinds of instructions may modify the position of the DP. Whenever an lds is encountered, it must be ensured that DP is below its frame-relative address FA starting from ST, i.e., $\mathbf{DP} \leq \mathbf{FA}$, since the value loaded by the instruction is known to be live. Recall that the analysis proceeds in a backward fashion, so the loaded value is live at all program points *before* the lds, up to a preceding sts instructions potentially overwriting the same FA. An sts, on the other hand, might push the DP upward as the overwritten data is dead immediately before the store. This is only possible when the analysis is able to show that the newly discovered dead data is right above the contiguous block of dead data, such that a new contiguous block can be formed. The sts overwrites data at a given FA in the cache, the overwritten value thus can no longer be accessed and is dead at all program points before the store instruction, up to a join (conditional branch) and/or an lds instruction potentially rendering the data live. Finally, with regard to a function, all its data is dead immediately before its sfree, since none of the data in the stack frame can be accessed from this point on. The DP then is at its highest possible position, i.e., the stack frame's size k. In addition to these three instructions that may directly have an impact on the DP, the analysis also needs to consider conditional branches, i.e., instructions that may have multiple successors in the CFG. Since the analysis proceeds backward, the successor's DP values might be different. The analysis thus needs to apply a join operator (Sect. 2), which selects a conservative approximation. In the case of CSA, the minimum, i.e., the least amount of dead data, is considered.

The following data-flow equations specify how individual instructions (Eq. 3) and joins (Eq. 4) may modify the relative position of the DP with regard to the stack frame of a function:

$$\mathrm{OUT}(i) = \begin{cases} \mathtt{k} & \text{if } i = \mathbf{sfree}\ \mathbf{k} \\ \min(\mathrm{IN}(i), \mathbf{FA}) & \text{if } i = \mathbf{lds}\ \mathbf{FA} \\ \mathrm{IN}(i) + 1 & \text{if } i = \mathbf{sts}\ \mathbf{FA} \wedge \mathbf{FA} = \mathrm{IN}(i) \\ \mathrm{IN}(i) & \text{otherwise} \end{cases} \tag{3}$$

$$\mathrm{IN}(i) = \begin{cases} 0 & \text{if } i = t \\ \min_{s \in Succs(i)}(\mathrm{OUT}(s)) & \text{otherwise} \end{cases} \tag{4}$$

The position of the DP before (and after) each instruction in the function can then be computed by applying these equations iteratively until a fixed-point is reached. The initial values assigned to $\mathrm{IN}(i)$ and $\mathrm{OUT}(i)$ for each instruction i have to be chosen such that the iterative processing actually converges and delivers a safe approximation. The above data-flow equations compute the lowest position of the DP, it thus suffices to initialize the equations with the size of the stack cache $|SC|$ or the size of the current stack frame k – both are upper bounds on the maximum value of DP. The initialization of $\mathrm{IN}(t)$ to 0, where t represents the function's exit point, along with the use of the minimum as the join operator ensures that the analysis converges towards the minimum value of the DP and consequently gives a safe approximation.

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 Abbaspour et al. 2014), the size of the dead area $\mathrm{DA}(i)$ (given by Eq. 3), and the maximum occupancy $\mathrm{Occ}(i)$:

$$savingCost(i) = \widehat{c_s} \max(0, \mathrm{Occ}(i) - \mathrm{CA}(i) - \mathrm{DA}(i)) \tag{5}$$

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 is readily supported by the respective analyses and can easily be considered in the above equations. The costs would then, of course, also be context-dependent.

It would, in addition, be possible to consider the calling-context when analyzing the dead area ($DA(i)$). Whenever all data in a function's stack frame is dead, the size of its caller's dead area can be added to $DA(i)$. 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.

Example 3 Consider the CFG of the function shown in Fig. 5, which consists of three blocks of straight-line code. The sequence of the top most block is assumed to end with a conditional branch having the two other blocks as successors (indicated by the edges on the left). The goal of the analysis is to track the area **DP** − **ST** of data that is known to be dead by computing the lowest possible position of DP at each program point. The analysis processes the CFG backwards, starting at the return instruction **ret** at the bottom. The computation of the analysis and its results are indicated in blue to the right of the code. Since the return is the last instruction in the function, the DP is initialized to 0. All stack data is potentially live here. The DP value is then propagated to its predecessor the **sfree** instruction. All stack data is known to be dead right before this instruction, the DP is thus set to 2, the instruction's argument (k).

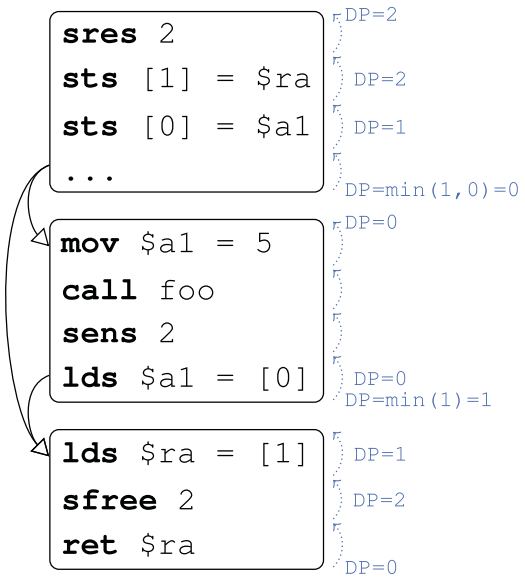

Fig. 5 Propagation of the DP (shown on the right in blue) within a function: stack data becomes dead right before `sfree` and `sts` instructions, while it becomes live before `lds` instructions. Other instructions do not impact the DP (Color figure online)

Next, the **`lds`** instruction is processed. The top most stack element of the function's stack frame is accessed (using the frame-relative address `[1]`) and thus becomes live, which is indicated by the new DP value of 1. The second load (`[0]`) in the block above is processed similarly. Here the DP drops to 0, indicating that both stack elements are live. The remaining instructions (**`mov`**, **`call`**, **`sens`**) in the same block have no effect on liveness. The last instruction of the top-most block has 2 successors with different DP values (1 and 0). The join operator conservatively takes the minimum to safely over-approximate the actually live stack data. The algorithm eventually processes the store instructions at the top. The first **`sts`** (`[0]`) overwrites the first stack element, whose value becomes dead. The DP thus is incremented to its new value 1. The subsequent **`sts`** (`[1]`) then overwrites the top element, rendering all stack data dead (**DP** $= 2$). Note, that instructions before the **`sres`**/after the **`sfree`** conceptually belong to the caller.

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 from main memory to the cache. A simple solution would again 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 Fig. 6: (1) cache blocks containing dead data only (given by Eqs. 3 and 4), (2) blocks potentially containing live data that have to be restored, and (3) blocks that are restored by a subsequent `sens`. Since only a 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 Sect. 3.2.1, while case (3) and (4) require inter-procedural analyses covered in Sects. 3.2.2 and 3.2.3.

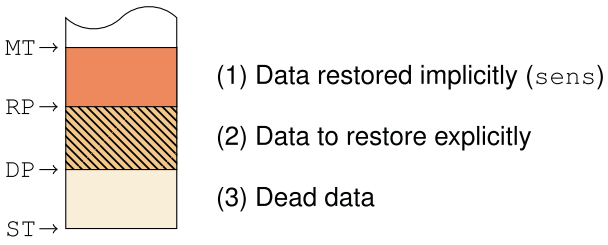

Fig. 6 Partitioning of the stack: *(1)* data restored by `sens` of the current as well as other functions, *(2)* data to restore, and *(3)* dead data

3.2.1 Local restore analyses

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

$$allocationCost(i) = \begin{cases} \widehat{c_a} \text{ if } \mathrm{DA}(i) \neq 0 \\ 0 \quad otherwise \end{cases} \tag{6}$$

Blocks containing live data have to be restored and thus transferred back from main memory. This can be done explicitly during the context restoration or implicitly by an `sens` instruction executed later by the program. While the explicit transfer always induces additional overhead that needs to be accounted for, the implicit restoration might be for free. This happens when the maximum filling computed by the traditional SCA for the `sens` instruction is non-zero. The overhead associated with the explicit restoration is then, at least partially, accounted for in the program's WCET.

In order to account for the overhead of implicit and explicit transfers two quantities have to be determined: (1) the amount of data that needs to be restored explicitly and (2) the cost of implicit memory transfers performed by `sens` instructions. We introduce another virtual marker to model the former quantity. This pointer represents an over-approximation of the amount of *live* data in the stack cache that is *not* implicitly restored by an `sens` instruction before a subsequent access rendering the data live:

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 Sect. 3.2.2.

The analysis of the `RP` is a function-local, *backward* analysis that tracks the highest possible position of the pointer relative to `ST` (i.e., **`RP`** $-$ **`ST`**), which means that an over-approximation needs to be computed. The position of the `RP` depends on the

amount of data restored implicitly as well as the amount of live data. The analysis thus needs to consider the impact of `sens` instructions, which lower the position of the `RP`, as well as memory accesses, which may increment the `RP`. Whenever an `sens` instruction is encountered by the analysis the position of the `RP` is set to 0, which simply means that no data needs to be restored in case of a preemption that occurs immediately before that ensure (recall that the analysis proceeds backward). The `sens` simply reloads the entire stack frame when the task gets resumed. Data becomes live whenever it is accessed by an `lds` instruction, the position of the `RP` thus has to be larger or equal to the `FA` of any load instruction encountered. In order to simplify the handling of dead data, `lds` and `sts` instructions are both considered to increment the `RP` – despite the fact that stores do not actually render the data live. Dead data is excluded from explicit and implicit transfers anyways using the `DP` (as indicated above). Also note that there is no strict ordering between the `DP` and the `RP`, i.e., it might happen that $\mathbf{DP} > \mathbf{RP}$. This usually happens when dead data is present at an `sens` instruction, which sets the `RP` to 0, but has no impact on the `DP`. In addition to the instructions that have an immediate impact on the `RP` the analysis also needs to account for control-flow joins at conditional branches, which may have multiple successors with diverging `RP` values. The analysis always selects the maximum value and propagates this information upwards in order to ensure that the position of the `RP` is safely over-approximated. The following equations capture the evolution of the `RP` relative to `ST`:

$$\mathrm{OUT}_{RP}(i) = \begin{cases} 0 & \text{if } i = \mathbf{sens\ \ k} \\ \max(\mathrm{IN}_{RP}(i), \mathbf{FA}) & \text{if } i = \mathbf{lds\ FA} \vee i = \mathbf{sts\ FA} \\ \mathrm{IN}_{RP}(i) & \text{otherwise} \end{cases} \tag{7}$$

$$\mathrm{IN}_{RP}(i) = \begin{cases} 0 & \text{if } i = t, \\ \max_{s \in Succs(i)}(\mathrm{OUT}_{RP}(s)) & \text{otherwise} \end{cases} \tag{8}$$

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 ($\mathrm{DA}(i)$, Eq. 3) and the size of the restore area ($\mathrm{RA}(i)$, Eq. 7):

$$transferCost(i) = \widehat{c_r} \max(0, \mathrm{RA}(i) - \mathrm{DA}(i)) \tag{9}$$

Example 4 Figure 7 illustrates the propagation of the `RP` through the CFG from the previous example. The processing again starts at the bottom of the CFG at the return instruction. At this point all data is dead ($\mathbf{RP} = 0$) and thus does not need to be restored explicitly. This changes when the first **`lds`** instruction (`[1]`) is encountered, which renders the top-most element of the stack frame and all elements below it live. This is indicated by the new `RP` value of 2. The `RP` does not change until the **`sens`** instruction is processed. At this point all the stack frame's data is known to be live. However, the **`sens`** instruction ensures that all stack data is filled into the cache. Thus no explicit restoration is required and the new value of `RP` becomes 0. The other instructions in the block above the **`sens`** do not have an impact on the `RP`. As before, the last instruction of the top-most block is assumed to be a conditional branch with two

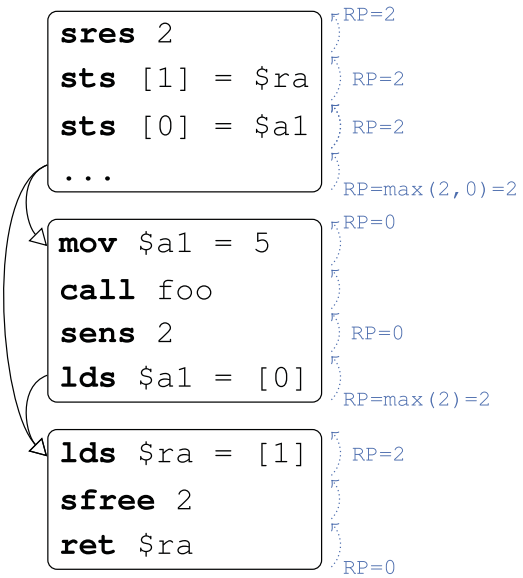

Fig. 7 Propagation of the RP (shown on the right in blue) within a function: only lds, sts, and sens instructions impact the RP, while other instructions do not modify its value value (Color figure online)

successors having different RP values. This time, the maximum value 2 is propagated upwards in order to conservatively over-approximate the amount of potentially live data. The subsequently processed instructions do not have an impact on the RP since it is already at the maximum position (2), which would indicate that the entire stack frame needs to be restored explicitly. However, since the DP was shown to be non-zero (see Example 3), the stack frame only needs to be restored partially.

It remains to account for the implicit transfer costs at sens instructions in the current function that are not already included in the program's WCET. This situation arises whenever the RP pointer is *not* at its maximum position (the size of the function's stack frame k). The function's stack frame is thus only partially restored by the explicit transfer after a preemption and some additional cache blocks need to be filled from main memory implicitly by the next sens instruction. The additional cost of this transfer depends on the size of the function's stack frame, the size of the restore area (RA(i) from above), and the number of cache blocks that need to be transferred by the sens instruction for a regular execution without preemption, which is provided by the standard SCA in the form of an annotation to the instruction ($\langle b \rangle$). The overhead is trivially upper-bounded by the function's stack frame size k. A more precise bound would be $\mathbf{k} - \mathrm{RA}(i)$, which reflects the reduction of the cost of the implicit transfer cost by deducting the explicitly transferred blocks. Another bound can be derived from the maximum filling bound b associated with an sens instruction. The additional costs due to the implicit restoration cannot exceed $\mathbf{k} - b$, which represents the maximum number of cache blocks whose transfer costs are not accounted for in the program's WCET.

The following cost function combines both of the above approaches. However, before the cost function can be defined, an intermediate step has to be performed, which propagates the maximum filling bounds associated with individual sens instructions to all program points throughout the function. This allows to determine for each instruction, also those that are not an sens, the number of cache blocks that are potentially filled by any subsequent sens instruction. This intermediate step can be implemented using a function-local, *backward* DFA, propagating the difference

between the `sens`'s argument `k` and its filling bound $\langle b \rangle$ (obtained from the standard SCA) upwards through the CFG:

$$\mathrm{OUT}_{FL}(i) = \begin{cases} \mathbf{k - b} & \text{if } i = \mathbf{sens\ k}\ \langle \mathbf{b} \rangle \\ \mathrm{IN}_{FL}(i) & \text{otherwise} \end{cases} \tag{10}$$

$$\mathrm{IN}_{FL}(i) = \begin{cases} 0 & \text{if } i = t, \\ \max_{s \in Succs(i)}(\mathrm{OUT}_{FL}(s)) & \text{otherwise} \end{cases} \tag{11}$$

The overhead caused by implicit memory transfers of `sens` instructions can then be computed from the number of cache blocks that are filled implicitly ($\mathrm{FL}(i)$, Eq. 10) and the number of blocks that were explicitly restored, i.e., the size of the restore area ($\mathrm{RA}(i)$, Eq. 7):

$$ensureCostLocal(i) = \widehat{c_r} \max(0, \mathrm{FL}(i) - \mathrm{RA}(i)) \tag{12}$$

As for the analysis of the `DP` before, the above data-flow equations for the `RP` and the local filling need to be initialized properly in order to ensure that the fixed-point computation converges. Since both analyses define the join operator as the maximum over all successors, the equations have to be initialized to 0 before the resolution process starts. In the case of the `RP` analysis this indicates that no data needs to be restored explicitly after the function's `sfree`. The first access to stack data encountered by the analysis will then increment the `RP` value accordingly. The iterative processing then ensures that the analysis converges towards a safe upper bound. A similar argument applies to the propagation of the local filling bounds. The equations also contain an explicit initialization to 0 for the function's exit point t. This initialization is, in fact, redundant, given the fact that t cannot have any successors and that all data-flow equations are initialized to 0 anyways.

3.2.2 Global ensure analysis

The analyses in the previous subsections exclusively focus on the state of the stack frame of a single function and account for additional costs related to the restoration of the stack frame of the function whose execution was interrupted by a preemption. The stack frames of other functions that are currently on the call stack are not explicitly restored. This is done via implicit memory transfers, which are performed by the `sens` instructions that are placed after every function call. The underlying idea is very similar to the local reserve analysis discussed before, with the main difference that no explicit memory transfer is performed whatsoever.

To analyze the costs associated with these implicit memory transfers, an over-approximation needs to be computed that considers all possible states of the call stack, i.e., all possible chains of nested function calls leading up to a call to the function under analysis. This is sufficient, since the additional overhead is only induced by the `sens` instructions that are executed upon returning from functions along the call stack. The program's call graph (CG) is a well-known representation capturing all chains of nested function calls that may occur during the execution of the program (see Sect. 2).

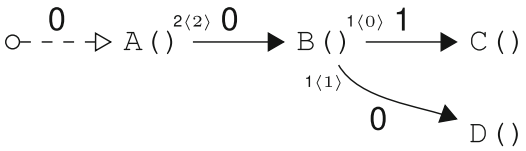

Fig. 8 Weighted CG of the code in Fig. 1 used to bound the additional transfer costs at `sens` instructions of other functions

Each such chain observed during the execution of the program corresponds to a path in the call graph starting at the program's entry point (typically the `main` function) and leading to the graph node representing the current function. In order to compute the desired over-approximation, the analysis thus needs to consider all paths leading to the currently considered function and associate a cost with each path.

We model this problem as a longest path search on a weighted CG, considering all paths 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) (Eq. 11) of the site's `call` instruction. Note that this problem is very similar to the computation of the maximum displacement of the original SCA (Abbaspour et al. 2013). However, the length of the path is bounded: (1) by the maximum occupancy at the call site (which is itself bounded by the stack cache size) and (2) by the minimum amount of stack data remaining in the stack cache after returning from the function, i.e., $\max(0, |SC| - D(f))$, where $|SC|$ denotes the stack cache size and $D(f)$ the function's maximum displacement. The latter case is particularly interesting, since no computation is required when the function's displacement is larger than or equal to the stack cache size. The length of the path and the restoration costs then simply become 0. Also note that the global ensure costs are always the same, independent of where the program is interrupted in the function. It is thus sufficient to pre-compute the costs only once for each function.

Our algorithm thus pre-computes the global ensure costs as follows. A weighted call graph is constructed beforehand, where the edge weights are provided by the local ensure analysis (Eq. 11). The algorithm then processes each function f separately. First, it is verified whether the maximum displacement $D(f)$ of f exceeds the stack cache size. If this is the case, the global ensure costs are bounded by 0, and the algorithm simply proceeds with the next function. Otherwise, an integer linear program (ILP) is constructed, which is similarly structured as the traditional IPET approach (Li and Malik 1995). The ILP encodes all paths originating at the root node of the CG leading to the current function, such that the objective function represents the length of the path. An ILP solver then computes the longest such path, by maximizing the objective function. Note that this approach allows to handle any kind of program, including those with recursion. The original work on the SCA includes a detailed description on the handling of recursion (Abbaspour et al. 2013). Note that for programs without recursion the longest path for *all* functions can be computed in linear time using dynamic programming (Cormen et al. 2009). Given the length FLG(f) of such a path for function f, the costs induced at other functions is:

$$ensureCost(f) = \widehat{c_r}\,\mathrm{FLG}(f) \tag{13}$$

Example 5 Consider the code from the initial example in Fig. 1. The algorithm begins by constructing a weighted call graph as shown in Fig. 8. Apart from the edge weight that is shown in the middle of each edge, the figure also indicates the information provided by the local ensure analysis at the respective call site. The numbers at the origin of each edge represent the argument `k` of the next ensure instruction following the call site as well as its filling bound in angle brackets. The edge weight simply correspond to the difference between these two values.

The edge weight for the call from `B` to `C`, for instance, evaluates to 1, since the corresponding ensure instruction may transfer an additional block, which is not accounted for by its original bound $\langle 0 \rangle$ ($1 - 0 = 1$). Similarly, the edge weight of the call from `A` to `B` evaluates to 0. The corresponding `sens` transfers up to 2 blocks, of which both are already accounted for by the bound $\langle 2 \rangle$ ($2 - 2 = 0$).

For `C` the longest path has a length of 1, since an additional block needs to be transferred if a preemption were to happen during the execution of `C`. The longest path from the program's entry to function **`D`** has length 0, i.e., all cache blocks of calling functions are restored for free as they are accounted for by the original bounds. The same result could have been computed from **`D`**'s maximum displacement (4), which is equal to the cache size (4).

3.2.3 Global reserve analysis

Lazily restoring the stack cache content not only allows us to avoid explicit memory transfers during context switches, but it may also turn out to be profitable. Even in the worst-case only the stack frame of the current function is restored, which leaves the remaining space in the stack cache free and thus effectively reduces the stack cache's occupancy. This may be beneficial for subsequent `sres` instructions, since the reduced occupancy may also reduce the maximum spilling. This, consequently, may reduce the running time of the program under analysis. There are two scenarios where such a gain might be observed: (1) at an `sres` of another function that is called from the current function and (2) at an `sres` of another function that is called after returning from the current function. It is important to note here, that multiple `sres` instructions may profit from the reduced occupancy, i.e., when several calls are nested or are performed in sequence with increasing displacement values. The analysis thus needs to be able to accumulate gains of multiple function calls, while providing a conservative under-estimation of the actual gains. We will initially focus on the first scenario and limit our attention to a single function call, and later extend this solution in order to handle the accumulation of gains as well as gains from the second scenario.

Recall that the WCET of the program under analysis already includes an estimate of the maximum spilling at `sres` instructions, which is computed for each function individually from the maximum occupancy before entering the function and the amount of stack space (`k`) reserved by the function's `sres`. This can be generalized to several nested function calls by considering the displacement at the outer-most function call. The maximum spilling performed by all called functions can then be bounded by considering the maximum occupancy along with the maximum displacement of the nested function calls. While the minimum spilling can be bounded by considering the minimum occupancy along with the minimum displacement. Consequently, a conser-

vative estimation of the minimum gain can be computed by comparing the minimum spilling of a normal execution with the minimum spilling after a preemption. More formally, given a call instruction i with a minimum occupancy $\mathrm{mOcc}(i)$ and a minimum displacement $d(i)$ the minimum spilling during a normal execution is given by:

$$minSpill(i) = \max(0, \mathrm{mOcc}(i) + d(i) - |SC|) \tag{14}$$

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 current function's stack cache control instructions `k`:

$$minSpillPr(i) = \max(0, \mathtt{k} + d(i) - |SC|) \tag{15}$$

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

$$siteGain(i) = \max(0, minSpill(i) - minSpillPr(i)) \tag{16}$$

A simple solution to account for the impact of the *next* function call is to propagate the gain at call sites backward through the CFG. The following equations determine the minimal gain that is guaranteed to occur for only one of the *subsequent* function calls. At joins, the equations select the minimum, while the maximum is selected on straight line code:

$$\mathrm{OUT}_{GN}(i) = \begin{cases} \max(\mathrm{IN}_{GN}(i), siteGain(i)) & \text{if } i = \texttt{call} \\ \mathrm{IN}_{GN}(i) & \text{otherwise} \end{cases} \tag{17}$$

$$\mathrm{IN}_{GN}(i) = \begin{cases} 0 & \text{if } i = t, \\ \min_{s \in Succs(i)}(\mathrm{OUT}_{GN}(s)) & \text{otherwise} \end{cases} \tag{18}$$

As before, the data-flow equations have to be initialized in order to ensure that the analysis converges. The equations have to be initialized to the maximum possible gain, i.e., $|SC|$, the size of the stack cache, except for the function's exit node t. Since, at this moment, the analysis only considers local gains due to calls from within the current function, the gain at the end of the function evaluates to 0 for t (Eq. 18). The analysis converges towards a minimum gain, despite the fact that on straight-line code the maximum value is propagated (which initially indeed is $|SC|$). This is ensured by the initialization of t (0) and the fact that the minimum value is selected at joins (Eq. 18). The analysis thus will eventually reevaluate the data-flow equations of all program points reachable in the reversed CFG from the function's exit node t and converge towards a minimum.

Since we initially did not expect considerable returns from this analysis, our initial publication (Abbaspour et al. 2015) adopted this simple approach only without developing it further. Though simple to compute, this solution is conservative. The gain of subsequent calls can, in fact, be accumulated since the occupancy remains lower than in a regular execution even after returning from the called functions. Unfortunately,

this accumulation of costs cannot directly be encoded using data-flow equations. The accumulated costs in cyclic regions of the CFG would grow infinitely and thus yield wrong results.

Since then, we noticed that the underlying problem can, in fact, be seen as a shortest path problem on a weighted CFG. The edge weights in the graph represent the gain associated with individual call sites (Eq. 16), while the length of the shortest path from an instruction i to the CFG's sink node t represents the accumulation of gains for all of the visited call sites. This is possible since every call site is uniquely identified even if some function is called many times in some execution path. However, the analysis has to make sure that the gain of visited call sites is not accumulated more than once. Fortunately, this cannot occur since the only way to revisit the same call site again would be in a loop. Such a scenario is naturally avoided by the shortest path search algorithm, since revisiting the same call site would increase the path length. Given the length $\mathrm{LSG}(i)$ of the shortest path for instruction i, it is now possible to account for the actual gain associated with all function calls possibly executed within the current function after a preemption at i:

$$reserveGainLocal(i) = \widehat{c}_s \mathrm{LSG}(i) \tag{19}$$

Note that the length of the path, and thus the local gain, is bounded. The gain can never exceed $|SC| - \mathtt{k}$, since the lazy restoration may in the worst-case only reload the local stack frame, whose size is given by `k`. This can also be seen by assuming that the minimum occupancy ($\mathrm{mOcc}(i)$ in Eq. 14) evaluates to $|SC|$. Simplifying the formulas (cf. Eq. 15 and 16) yields the same result. This bound holds for nested function calls and sequences of function calls. The nesting of function calls is conservatively modeled using the minimum displacement ($d(i)$) in the formulas. The effects of function calls that are performed in sequence are conservatively modeled by considering the minimum occupancy, provided by the standard SCA. Recall that the minimum occupancy can be bounded locally by considering the impact of function calls through their maximum displacement (see Sect. 2). The gain of each function call in a sequence thus is reduced by preceding calls due to the reduced minimum occupancy, which immediately depends on the calls' maximum displacements. The accumulated local gain thus cannot exceed the aforementioned bound since the gain gradually approaches 0 due to the interplay between minimum occupancy and maximum displacement of intermittent calls.

Example 6 The gain due to the reduced spilling at the function entry of `B` needs to be analyzed, right after its reserve instruction B_2. The function first calls `C`, whose minimum occupancy, provided by the standard SCA, evaluates to 3, while its displacement evaluates to 1. Spilling will never occur while executing `C`, since the stack frames of `A`, `B`, and `C` fit into the stack cache ($2 + 1 + 1 = 3 + 1 <= 4$). The local reserve gain associated with the respective call site thus is 0. Likewise, the minimum occupancy before the call to `D` is 3, its displacement, however, is 4. The `sres` instruction D_1 consequently spills 3 blocks ($3 + 4 - 4$) during a regular execution (Eq. 14), while only a single block ($1 + 4 - 4$) is spilled after a preemption (Eq. 15). The call site is thus associated with a weight of 2 ($3 - 1$). Since both calls are executed in any case,

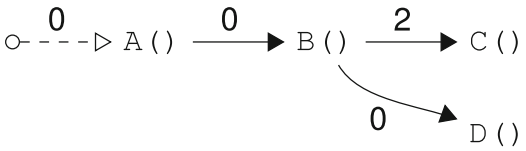

Fig. 9 Weighted CG of the code in Fig. 1 used to bound the global gain due to `sres` instructions of other functions

the length of the shortest path from the preemption point to the end of `B` evaluates to 2 (cf. Eq. 19).

The previous analysis only accounts for the gain due to function calls within the current function. In addition, it is also possible to account for potential gains after returning from the current function. Recall that the stack frames of all functions currently on the call stack are lazily restored. The occupancy after a preemption may thus also be lower for these functions compared to a regular execution without preemption. Similar to the computation of the global ensure costs, we can account for this gain through a path search on a weighted CG. The edge weights for this graph are given by the local reserve gain (Eq. 19) at the respective call sites.

The minimal gain that is guaranteed to occur for all executions has to be computed. The algorithm thus has to search for the shortest path in the CG instead of the longest. Given the length $\mathrm{GSG}(f)$ of the path for function f, the global gain due to `sres` instructions is given by:

$$reserveGain(f) = \widehat{c_s}\,\mathrm{GSG}(f) \tag{20}$$

An interesting observation at this point is that the global reserve gain is bounded just as the local reserve gain before. As the analysis climbs upwards through the call graph (towards the program's entry function), the displacement of the functions increases. With this increase the potential gain of subsequent function calls diminishes (as before), limiting the global reserve gain to the minimum of either the minimum occupancy at the function's entry or $|SC| - \mathbf{k}$, where `k` represents the current function's stack frame size.

Example 7 Consider the code from the initial example in Fig. 1. The algorithm begins by constructing a weighted call graph as shown in Fig. 9. The edge weights correspond to the local gain associated with each call site, given by Eq. 19. For the call from `B` to `C`, for instance, the edge weight evaluates to 2. This is because a preemption that occurs in `C` will eventually return to its caller `B` with a reduced occupancy. This will lead to reduced spilling during the subsequent call to `D`, as explained in more detail in Example 6. The local reserve gain after the call to `C` at instruction B_4 thus gives the above edge weight. The same applies for the call from `B` to `D`. Here, the local reserve gain at instruction B_6 yields the edge weight 0, since no additional function calls appear after that instruction. The global reserve gain at `C` thus evaluates to 2 ($0+0+2$), which correspond to the length of the path from the call graph's root to the node representing the function. For all other functions the global gain simply evaluates to 0.

3.2.4 Context restore costs

The total context restoration costs are then bounded by accumulating the individual costs for space allocation, the explicit and implicit transfer of cache blocks at `sens` instructions (locally and globally). In addition, the preemption costs are partially amortized by the reduced spilling at `sres` instructions (locally and globally). Note that $f(i)$ denotes the function containing instruction i:

$$\begin{aligned} restoreCost(i) = {} & allocationCost(i) && \text{(Eq.6)} \\ & + transferCost(i) && \text{(Eq.9)} \\ & + ensureCostLocal(i) && \text{(Eq.12)} \\ & + ensureCost(f(i)) && \text{(Eq.13)} \\ & - reserveGainLocal(i) && \text{(Eq.19)} \\ & - reserveGain(f(i)) && \text{(Eq.20)} \end{aligned} \tag{21}$$

Example 8 Consider again the preemption point at instruction C_3 in the code shown in Fig. 1. The context restoring analysis first determines the minimal/maximal offset of the `DP` and `RP` with regard to `ST` respectively (Eqs. 3 and 7). The `RP` offset is 0, due to the absence of `lds` and `sts` instructions in the code, which are omitted for brevity. `DP` on the other hand, is equal to 1, since all data is dead right before the `sfree` instruction C_4. Therefore a single block has to be allocated to properly rebuild `C`'s stack frame, i.e., $allocationCost(C_3) = 1$. Since all data of the current stack frame is dead, neither an explicit memory transfer nor an implicit restoration by an `sens` instruction is necessary, i.e., $transferCost(C_3) = 0$ and $ensureCostLocal(C_3) = 0$.

After returning to its caller, the ensure instruction B_5 has to restore the entire stack frame of `B`. The additional cost has not been considered by the bound provided by the standard SCA ($\langle 0 \rangle$). The instruction thus fills an additional cache block. The `sens` instruction A_4, on the other hand, restores all of `A`'s cache blocks for free ($\langle 2 \rangle$). Therefore, the global ensure cost accounts for the transfer of an additional cache block ($ensureCost(\mathbf{C}) = 1$), as illustrated before by Example 5.

As **`C`** does not call any other function, it cannot profit from a local reserve gain ($reserveGainLocal(C_3) = 0$). However, the analysis determines the potential gain for function calls after returning from `C`. The occupancy before the call to **`D`** is reduced by 2 blocks compared to a regular execution. The local gain associated with the corresponding call site is thus 2. Since there is no other subsequent call in **`B`** nor in **`A`**, no further gain can be considered. The global reserve gain for function **`C`** thus amounts to 2 cache blocks ($reserveGain(\mathtt{C}) = 2$), as shown in Example 7.

The total cost, associated with context restoration after a preemption at the indicated program point C_3, is thus given by $1 + 0 + 0 + 1 - 0 - 2 = 0$, assuming unit costs of $\widehat{c_a} = \widehat{c_s} = \widehat{c_r} = 1$.

3.3 Computational complexity

The overall complexity of the CSA and CRA depends on the various analysis steps, which consist of four classes of analysis problems: (1) function-local data-flow anal-

yses, (2) longest path searches on the CG, (3) shortest path searches on the CG, and finally (4) shortest path searches on the CFG of individual functions.

We assume that the data-flow equations of the various DFAs are solved using a traditional worklist algorithm, which iterates until a fixed point is reached. Various complexity bounds can be considered for different classes of DFAs, depending on the size of the CFG as well as on characteristics of the analysis domain.

In our case, the domains are essentially natural numbers in the range $[0, \mathbf{k}]$ (cf. the analyses of the `DP`, `RP`, and local filling) or $[0, |SC|]$ (local gain), where `k` in turn is also bounded by $|SC|$. The considered analyses are monotone, i.e., the analyzed values steadily increase or decrease until either the minimum or maximum value of the domain is reached. This is called the height of the domain, which can be bounded by $|SC|$ for all considered DFAs.

The iterative worklist algorithm then propagates the domain values along the control-flow edges in the CFG. This leads to a first, conservative, complexity bound for the previously described analyses, which is in $O(|E||SC|)$ considering the height of the domain $|SC|$ and a CFG with $|E|$ edges. Another bound can be derived using the *loop connectedness* of the reversed CFG (since all considered problems are backward problems). This parameter characterizes the nesting of loops in a CFG G with respect to a spanning tree T of G (Hecht and Ullman 1973; Kam and Ullman 1976) and usually is denoted as $d(G, T)$. The number of iterations performed by the worklist algorithm can be bounded by this parameter when the *order* in which the CFG edges are processed is well chosen. The iterative processing may then process each CFG edge at most $d(G, T) + 3$ times, resulting in a complexity bound of $O(|E|(d(G, T) + 3))$, where $|E|$ again denotes the number of CFG edges in the CFG G. The loop connectedness can be considered constant in practice (since loops tend to have a simple structure). Similarly, the size of the stack cache $|SC|$ can be considered constant with regard to the analysis problems. The overall complexity of all the previously described DFAs is thus linear in the size of the CFGs of the individual functions in the program under analysis.

The global ensure analysis relies on longest path searches on the CG in order to bound the cost induced by implicit memory transfers of `sens` instructions. For programs with recursion (which are often forbidden in the context of real-time systems) this requires the construction of an ILP (similar to the well-known IPET approach by Li and Malik (1995)) for each node of the CG. The ILPs are subsequently solved by an external solver (such as CPLEX or LPSolve). While it is possible to bound the complexity of constructing an ILP in linear time with respect to the size of the CG, it is difficult to bound the solving times. Integer linear programming in general is NP-hard. However, it appears that today's solvers are able to handle the problem instances we encountered in our experiments quite well. The solvers almost instantly provide an optimal solution—even open-source solvers that do not apply sophisticated heuristics. For programs without recursion, the longest path search for all functions can be performed in $O(|F| + |A|)$ (Cormen et al. 2009), where $|F|$ represents the number of functions in the CG and $|A|$ the number of call sites. Note, furthermore, that the two approaches can be combined, i.e., dynamic programming is applied to a reduced CG where cyclic regions are collapsed. The potentially expensive ILP solving can then be limited to the recursive functions only (Jordan et al. 2013).

The shortest path searches on the CG (global reserve gain) and the CFGs of individual functions (local reserve gain) can be performed in quadratic time in the size of the respective graphs using simple algorithms. More advanced algorithms allow to reduce the complexity to almost linear time, e.g., the algorithm of Thorup (2004) yields a complexity in $O(|E| + |V| \log\log |V|)$. The local reserve gain can thus be computed in almost linear time with regard to the size of the CFGs of individual functions. The same applies to the global reserve gain, which can be computed in almost linear time with regard to the number of functions and call sites in the program. These bounds are independent from the graphs' shape, which may well contain cycles, i.e., loops in the case of CFGs or recursion in the CG.

The complexity of the proposed CSA and CRA analyses thus is dominated by the longest/shortest path searches, whose complexity depends on the size of the program under analysis (both in terms of function size as well as the size of the CG). Lastly, the complexity of the standard SCA needs to be taken into consideration, since intermediate results of this analysis are reused in various analysis steps of the CSA and CRA. The SCA is similarly based on longest/shortest path searches that are combined with function-local DFAs. The complexity analysis for the SCA is almost identical to the discussion from above (Jordan et al. 2013). The overall complexity to compute the preemption costs is thus not impacted and is also dominated by the longest/shortest path searches.

3.4 Discussion

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` instructions. 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 cache control instruction in a function is logically considered to be part of the immediate caller.

4 Preemption mechanism

In Sect. 3, we provided analyses to bound the preemption cost for every instruction in the program. The information generated by the analyses is rich and includes various parameters involved in the preemption cost depending on the *precise* location of the preemption. The potentially large quantity of information, although precise, is of little use if the preemption mechanism cannot take advantage of it.

The preemption mechanism describes a set of operations that the real-time scheduler has to perform in order to execute a context switch. This includes saving and restoring the processor's register values, resetting the memory management unit (if

```
func SCSave(DP)
 sub rx = |SC| - DP
 sresr rx
 stw MT
```

(a)

```
func SCRestore(RP,DP)
 ldw MT
 mov ST = MT
 mov LP = MT
 sub rx = RP - DP
 sensr rx
 mov ry = DP
 sresr ry
```

(b)

Fig. 10 Low-level functions to save/restore the stack cache content. **a** SCSave, **b** SCRestore

one exists), as well as re-configuring other shared hardware resources. If a stack cache is present its content needs to be saved and restored explicitly, as shown by the assembly code in Fig. 10. The `SCSave` function saves the content of the stack cache to main memory, depending on the size of the stack cache and the amount of dead data. We assume that the stack cache size is known statically ($|SC|$), while the amount of dead data is assumed to be a parameter of the function (`DP`). A simple approach to save the entire content of the stack cache is to perform an **`sres`** `|SC|`, which temporarily allocates a *stack frame* of the total size of the stack cache. Consequently, the entire content of the stack cache is spilled to main memory and thus saved. This approach automatically avoids the spilling of coherent data (`LP`). However, dead data would be saved as well. The `SCSave` function thus subtracts the size of the dead data from the stack cache size (**`sub`**) and stores the difference in a register (`rx`). The value of this register is then used by an **`sresr`** instruction, which is equivalent to a regular reserve with the only difference that the instruction's argument is a register. Finally, we save the `MT` pointer (**`stw`**), which points to an address in main memory where all of the current task's stack data is now saved (excluding dead data at the bottom). This address is later needed during context restoration.

The `SCRestore` function restores the stack cache content after a preemption and, for this, requires the `RP` and `DP` pointers as arguments. Before any stack data can be transferred to the cache, the previously saved `MT` pointer needs to be reloaded first (**`ldw`**). The `ST` and `LP` pointers are set to the same address, which represents an empty stack cache. Then we proceed to the restoration by explicitly filling cache blocks between `RP` and `DP` using an **`sensr`** instruction. As before, an **`sensr`** takes its argument from a register (`rx`). The value of the register is computed from the difference between the two arguments `RP` and `DP` (**`sub`**). Note that we assume here that **`RP`** $\geq$ **`DP`** in order to simplify the assembly code. The ensure transfers live data from main memory and updates the `MT` pointer. Neither the `LP` nor the `ST` is modified since the reloaded data is known to be coherent. Finally, we take care of dead data, which was excluded from context saving before. It suffices to allocate a matching amount of cache blocks on the stack cache using an **`sresr`** instruction. This ensures that subsequent accesses to the stack cache succeed, while avoiding a useless memory transfer.

With the help of the `SCSave` and `SCRestore` functions any desired context saving mechanism can be implemented. One only has to ensure that the functions are called at the right moment and do not cause any side-effects, e.g., on registers of the

involved tasks. However, it remains to resolve one issue: how do the functions obtain the parameter values for `DP` and `RP`?

One possible, but impractical solution, would be to store these parameter values in a look-up table. It would then be possible to retrieve the precise values of both pointers, as determined by the analysis, during context switching. The memory footprint of the table as well as the costs associated with the table look-up disqualify this solution. Therefore, we need means to exploit the rich analysis information without impacting the predictability or inducing excessive overhead. This, however, highly depends on the underlying preemption policy.

In combination with a scheduling algorithm, the preemption policy determines the circumstances under which a running task is allowed to be preempted. For instance, the fully-preemptive policy allows preemptions to occur freely at any time and at any position in a task's program. Whereas the non-preemptive policy does not allow any preemption to occur, and a new task is started only after the running task has terminated. Although fully-preemptive approaches may offer better schedulability, they make it difficult to provide tight WCET estimates using cache analyses, since the preemption point is not known in advance. Hybrid approaches have been introduced to tackle this problem, either by statically fixing preemption points or limiting the number of preemptions that tasks may suffer. Examples of such approaches include the deferred preemption model (Burns 1995), the floating non-preemptive region model (Baruah 2005), or the preemption threshold (Wang and Saksena 1999). So, in order to provide a preemption mechanism that best matches a preemption policy, it is of major importance to consider whether the policy relies on statically fixed or non-fixed preemption points.

4.1 Handling fixed preemption approaches

The main advantage of fixed preemption points is that it gives precise control over the execution of real-time programs. It is a powerful approach allowing the preemption mechanism to take full advantage of all the capabilities of a cache analysis. This allows to choose interesting preemption points within a single task depending on the overhead determined by a cost analysis (considering, among others, cache analyses such as the CSA and CRA). These candidate preemption points are then considered globally by schedulability tests to ensure that the constraints of the entire system are respected. This may help to reduce the overhead due to preemption with regard to the global system utilization.

Once preemption points are chosen the full results of the previously described analyses (CSA and CRA) can be exploited easily, since dedicated code triggering a context switch can be inserted. This code may simply invoke the `SCSave` function before yielding the processor to the operating system kernel. Once the task is reactivated it suffices to call `SCRestore`. In both cases the function's parameters are immediately available and can be considered by the inserted code. This, furthermore, allows the WCET analysis to include the respective code and its overhead. A downside of this approach is, however, that the stack cache content is potentially transferred to/from main memory even when the operating system decides not to preempt the running task. Alternatively, the two functions could be implemented in the operating system,

which then invokes them as needed. The interface between the task and the operating system then needs to be revised such that the task can communicate the DP and RP parameters when yielding the processor, e.g., by explicitly setting predefined registers.

4.2 Handling non-fixed preemption approaches

In contrast to the previous strategy, handling non-fixed preemption approaches is quite challenging as preemption locations are not known in advance. This means that the preemption mechanism cannot pass, as described above, predetermined parameter values for DP and RP to the SCSave and SCRestore functions respectively. One solution, already mentioned before, is to store the parameter values in a look-up table. The operating system would then simply retrieve the parameter values considering the precise location of the preemption, e.g., by using the task's program counter as an index. The size of the look-up table inevitably disqualifies this solution. However, it might be possible to compress the table or reduce its size. For example, the table size could be reduced by storing only a single value for each of the two pointers for each function in the program, instead of storing the pointers for *all* possible program points. This would drastically decrease the table size at the expense of a slight increase of look-up costs. The values stored for each function have to be safe approximations. For the RP the maximum value over all program points in the function has to be chosen, while for the DP the minimum value has to be selected. This solution still appears impractical. However, the idea to attach approximations to limited regions within a program can be generalized.

In the following, we will present two solutions to this problem, based on lightweight extensions to the hardware and/or instruction set. The first solution relies on conservative approximations at the granularity of whole functions using an additional stack cache control register. The second solution requires a modification of the instruction set, which allows to embed analysis information in the standard stack cache control instructions.

4.2.1 Stack cache control register

This solution is motivated by our experiments, which are explained in more detail in Sect. 6. Our measurements indicate, not too surprisingly, that in most of the cases the stack frame of the current function needs to be restored entirely, either by an explicit memory transfer (Eq. 7) or an implicit memory transfer (Eq. 10). The preemption mechanism may thus simply restore the entire stack frame of the function where the preemption occurred, since the overhead for this operation already has been accounted for in any case. The parameters DP and RP for the SCSave and SCRestore functions are then simply approximated by 0 and k respectively, where k is the size of the function's stack frame.

The problem is that the standard stack cache does not track the size of the current stack frame. It only *knows* the ST and MT pointers representing the occupancy. The occupancy may reflect three different situations. Firstly, the stack cache only holds a subset of the frame. The occupancy thus is smaller than k. Secondly, the stack cache

contains only the frame. The occupancy here matches k. Finally, the stack cache may hold data of other functions in addition to the frame. The occupancy then is larger than k. It is obviously not possible to derive the size of the current frame from the state of a standard stack cache.

We thus propose to introduce an additional stack cache control register FP that keeps track of the size of the current stack frame. The stack cache control instructions are then required to keep this register up-to-date. The sres and sens instructions simply copy the value of their respective arguments into this register. On the other hand, sfree instructions merely reset the register to 0, since they destroy the stack frame and no stack cache access may occur until the next ensure.

The preemption mechanism may then retrieve the value of the FP register and pass it as the parameter RP to the SCRestore function. The DP parameter is conservatively set to 0 for both, the SCSave and the SCRestore functions.

This solution only requires minimal modifications to the stack cache hardware. The additional FP register and the logic needed to update it is negligible and thus has virtually no impact on the hardware cost and clock frequency. The timing behavior of instruction is not modified as the register update can be performed in parallel with other operations in a single cycle. The time-predictable behavior of the stack cache is thus preserved. Finally, the solution does not incur any overhead whatsoever with respect to the program's memory footprint or execution time. A shortcoming of this approach is that the value of the RP parameter is frequently overestimated, while the DP parameter is not exploited at all. The approach thus effectively discards all information regarding function-local analyses.

4.2.2 Instruction set extension

An alternative approach is to modify the stack cache control instructions, such that they can be used to piggy-back the analysis information. The basic idea is to add two additional operands to the sres and sens instructions that explicitly specify the values of the DP and RP parameters. The values of these operands are copied into two dedicated stack cache control registers, which then can be consulted by the preemption mechanism to invoke the SCSave and SCRestore functions. The sfree instruction does not receive additional operands and instead simply resets the two control registers to 0. This allows to express changing values of the parameters at a much finer level of granularity, independently from the size of the stack frame. More specifically, the operand values apply to all program points between two successive stack cache control instructions. The operand values can easily be computed by considering the maximum value for the RP and the minimum value for the DP in the corresponding region of the program. Function calls can be ignored in this computation, as will be explained below.

In order to illustrate the approach we consider two scenarios of successive instructions: an sres followed by an sres of another function (callee) and an sfree followed by an sens instruction of another function (caller).

In the first case, the operand values of the first sres instruction apply to all program points up to the execution of the second sres instruction, which automatically overrides the corresponding control registers. Note, in particular, that this includes

all program points in the called function before its `sres` (see Sect. 3.4). This is safe since these instructions cannot have any impact on the stack cache. Consequently, the operand values of the first `sres` can be computed by considering local program points belonging to the same function as the reserve, i.e., calls can be ignored. Furthermore, all instructions between a function call and its `sens` instruction cannot have an impact on the stack cache either. It is thus safe to consider all function-local program points between any two subsequent stack cache control instructions in order to compute the operands. Note that this reasoning also applies to other pairs of instructions, i.e., `sens-sens`, et cetera.

In the latter case, the `sfree` instruction destroys the stack frame of the current function. Implicitly, the stack frame of the caller now becomes active. However, at the moment when the `sfree` instruction is executed, the characteristics of the caller's stack frame are not known and thus cannot be embedded as operands in the free instruction. We solve this issue by simply resetting both control registers for the `DP` and the `RP` to 0. This means that, from the perspective of the preemption mechanism, the instructions between an `sfree` and the subsequent `sens` belong to the callee, which slightly differs from the model explained in Sect. 3.4. The preemption cost bound remains safe under this interpretation, since the implicit filling at the `sens` instruction is correctly accounted for by the global ensure costs.

The hardware overhead of this solution, again, is marginal, since only two additional registers as well as logic to update them are needed. The impact on the hardware cost and clock frequency remains negligible. Also, the timing behavior of the stack cache control instructions does not change, preserving the time-predictability of the stack cache. However, the additional operands require space in the instruction encoding, which may either increase the instruction size or otherwise impose limits on the maximum stack frame size – depending on the characteristics of the instruction set architectures and the number of free bits in the original instruction encoding of the stack cache control instructions. Assuming that the operands can be encoded using otherwise unused bits, this solution does not impose any overhead w.r.t. the program's memory footprint or execution time.

5 Virtual stack caches

The previous sections presented the timing analysis of preemption costs and mechanisms for context switching assuming a single stack cache. However, an important question arises regarding the integration of this timing analysis into a schedulability test. This section evokes some of the issues that may emerge during this process and propose virtual stack caches as a possible solution.

5.1 Schedulability analysis issue

When a preemption occurs, the content of classical data caches will be updated as the preempting task performs 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

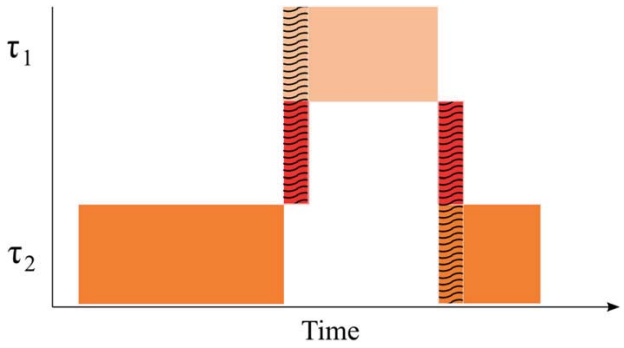

Fig. 11 Context switch overhead caused by preemption

integrating CRPDs can then be performed, as shown by Altmeyer et al. (2012) for instance. When considering a stack cache, the stack data of the preempted task must be saved before the preempting task can set up its own stack space. The preempting task is thus delayed. While the cost for saving the stack cache content of the preempted task can be bounded using the CSA, this delay may come with undesirable side-effects. Let us illustrate this through an example.

Example 9 Figure 11 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 Fig. 11) 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.

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 exists in caches with a write-back policy, which are not recommended for real-time systems (Wilhelm et al. 2009). 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.

5.2 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 [**`vscOffset`**, **`vscOffset`** + **`vscSize`**] in

the underlying memory. On a context switch, only `vscOffset` and `vscSize` have to be restored, 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.

5.3 Opportunities and challenges

The dynamic restoration of the VSCs under the control of the task scheduler, opens new research perspectives that may be explored. We will briefly enumerate some of those opportunities and the associated challenges and refer the reader to a first preliminary evaluation in our previous work (Abbaspour et al. 2015).

The task scheduler clearly requires a task model that allows to express constraints related to the VSCs (size, preemption costs,…). The task scheduler, in addition, has to reason about the bandwidth requirements of the necessary memory transfers associated with preemptions and needs a means to ensure that sufficient bandwidth ultimately is available to perform the transfers in time. Alternatively, the schedulability test may account for additional stall time that may occur when memory transfers cannot be guaranteed to be completed. This also requires associated analyses that allow to determine a lower bound on the bandwidth that can be guaranteed by the memory in parallel with the execution of a given task. These problems consequently touch several research domains, including operating system design, schedulability tests, computer architecture, as well as WCET analyses.

6 Experiments

This section presents an evaluation of the preemption costs associated with the stack cache. We cover full results from the static analysis (Sect. 3) as well as analysis results considering the proposed preemption mechanisms (Sect. 4). We furthermore propose means to implement these mechanisms.

The benchmarks are taken from the MiBench benchmark suite (Guthaus et al. 2001), 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 (Schoeberl et al. 2011). The instruction set of the processor follows the *Very Long Instruction Word* (VLIW) paradigm and may execute up to two instructions that are grouped into parallel bundles

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

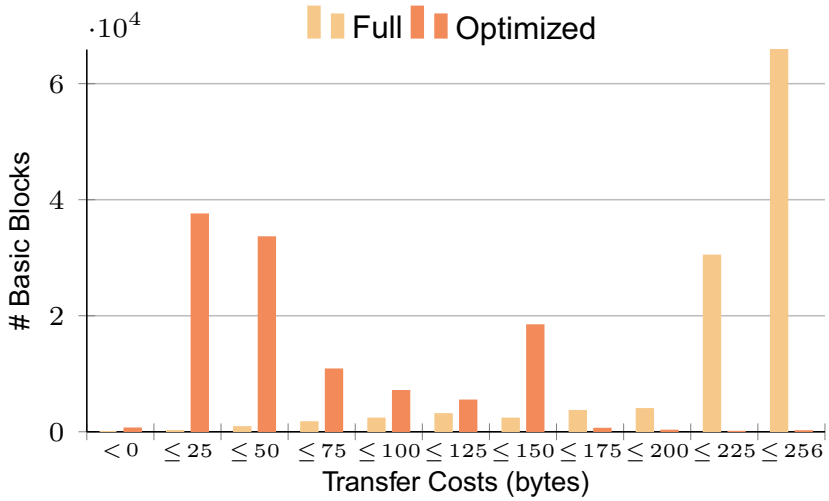

Fig. 12 Histogram of transfer sizes (in bytes) for context restoration at basic blocks using max. Occupancy (full) and our approach (optimized). Lower is better

at the same time. All instructions explicitly take a predicate operand, which allows to conditionally nullify instructions depending on the predicate value that is evaluated at runtime. The hardware of the platform is configured with a 64 KB, 4-way set-associative data cache using LRU replacement, and a write-through policy (recommended for real-time systems, see Wilhelm et al. 2009). Code is cached by a 64 KB method cache (Schoeberl et al. 2011) with LRU replacement and 32 code block entries. The stack cache is 256b small and uses a lazy pointer (Abbaspour et al. 2014). 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 cache control instructions still operate in words, 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, i.e., sequences of straight-line code that are typically terminated by a (conditional) branch instruction.

6.1 Context restoring analysis

The context restoring 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 114,257 basic blocks of which, 113,596 (99.4%) show an improvement. In the mean, over all benchmarks, the improvement is 4.1 fold (min. 3×, max. 7× per benchmark). Fig. 12 nicely illustrates these improvements. An unoptimized, full restoration typically reloads 250b

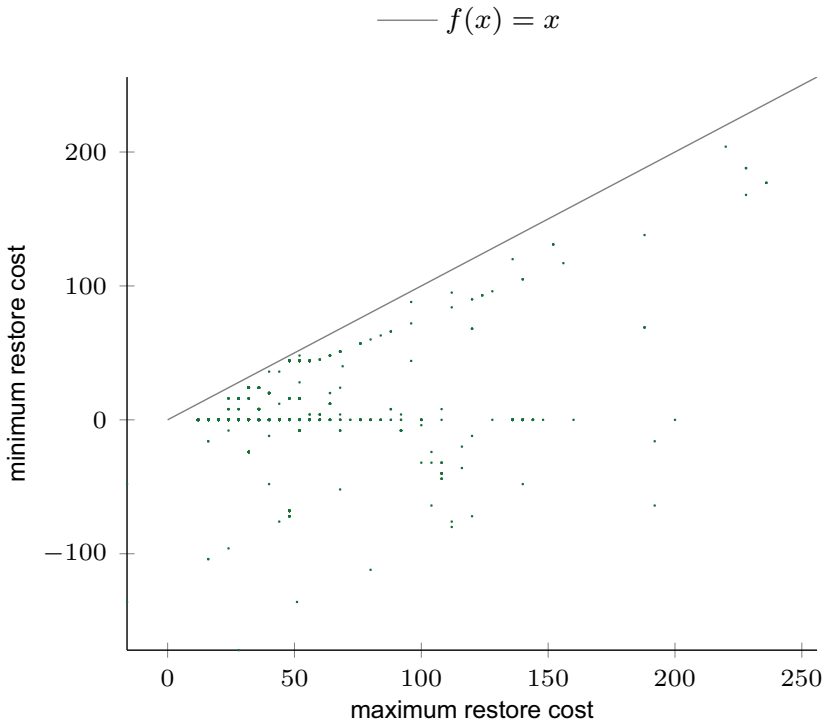

Fig. 13 Minimum vs. Maximum cost reduction (in bytes) for context restoration at functions. Smaller distance to the reference line is better

or more (50,281 basic blocks or 44%), while our optimized approach typically only reloads up to 50b (71,658 or 67%) with another peak between 126b and 150b.

In many cases no explicit memory transfer is needed at all (49,494 or 43.4%), i.e., Eq. 9 evaluates to 0, while for virtually all other cases the entire local stack frame is explicitly restored (62,934 or 56%). Out of the 49,494 cases, where no explicit memory transfer is required, 39,558 will eventually have their entire stack frame reloaded by a subsequent `sens`. Consequently, in 93.9% of the cases the preemption costs will have to account for the restoration of the current function's entire stack frame (either by a subsequent `sens` or by an explicit memory transfer). This suggests that simplified preemption mechanisms, such as those presented in Sect. 4, should yield reasonable results without inducing a considerable loss in precision. This is confirmed by our evaluation of the preemption mechanisms presented later in Sect. 6.3.

Furthermore, a close look at the minimum and maximum restore costs reveals that there often is no variation with regard to the restoration costs within individual functions. Out of the 8588 functions, 6575 (77%) show no variation at all. The variation for the remaining 2013 functions is illustrated by Fig. 13, which relates the maximum restoration costs against the minimum. In addition, we show the identity function $f(x) = x$ as a reference. Values close to the shown line indicate low variation. In our measurements, 1287 functions (64%) show an absolute variation below 32b and only 218 functions (10%) have a large variation above 64b. Due to the fact that the minimum restoration costs often evaluates to 0 (1436 functions or 71%), a relative comparison is difficult.

As can be seen in Fig. 12 and 13, even a few cases can be observed where the total restoration cost becomes negative (609 basic blocks or 0.5%), i.e., the program runs faster since the total transfer size to restore the cache content is smaller than the gain due

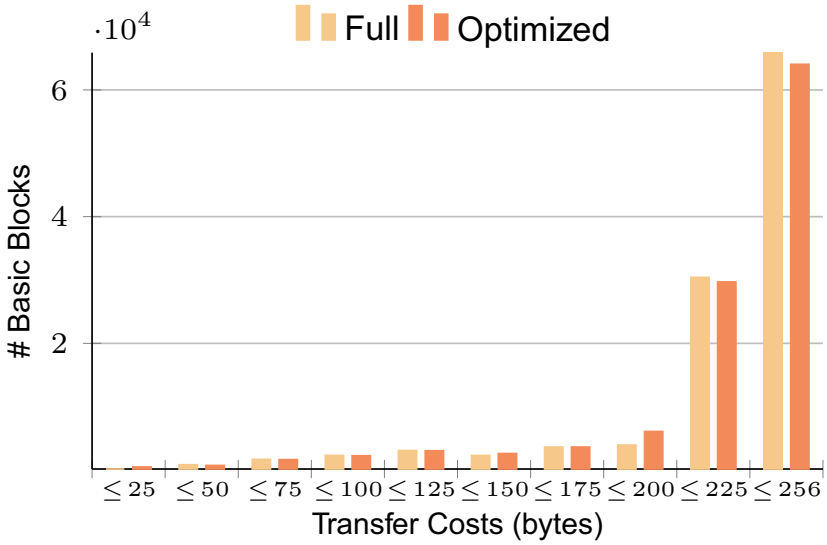

Fig. 14 Histogram of transfer sizes (in bytes) for context saving at basic blocks using max. Occupancy (full) and our approach (optimized) from Sect. 3. Lower is better

to reduced spilling (cf. Eq. 17). For 3488 basic blocks a non-zero gain due to reduced spilling was found (3%). Our new analysis algorithm improves upon the previous version (Abbaspour et al. 2015) by 77% with regard to the average local reserve gain and by a factor of 3.72 when considering the local and global gain combined.

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 (Sect. 3.2.2 and 3.2.3). 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 to compute the longest path.

6.2 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 114,257 basic blocks in the benchmarks 11,618 (10.1%) show a reduction in the context saving overhead. However, the reductions are moderate, as can be seen in the histogram of Fig. 14. For the basic blocks with lower transfer size, the reduction amounts to 8.9% on average over all benchmarks (minimum 5.9%, maximum 20.7% on average, 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 restoring 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 Sect. 5, thus appear to be important to limit this impact.

6.3 Preemption mechanisms

Now we evaluate the impact of the preemption mechanisms, as described in Sect. 4, on the analysis of preemption costs. Recall that the preemption mechanism invokes the `SCSave` and `SCRestore` functions, which require two input parameters: the amount of dead data and the amount of data that needs to be restored explicitly, which are denoted as `DP` and `RP` respectively. For the following experiments we consider three different implementation variants: (1) ISA-full, which is based on an instruction set extension that allows to specify two additional operands for both, the `DP` and the `RP` parameters, (2) ISA-RP, an instruction set extension covering only the `RP` parameter, and (3) FP, which represents a solution based on a single stack cache control register (`FP`) holding the current stack frame size. These configurations are compared against the optimized configuration from the previous experiments, representing the most precise preemption costs provided by an optimized analysis. Note that the results of the optimized analysis can be used in the setting of fixed preemption points.

An obvious difference between these preemption mechanisms is the level of granularity. The optimized analysis is able to compute precise results for each instruction in the program. In the previous experiments the analysis results were, however, limited to the beginning of basic blocks. Over all benchmark programs the number of basic blocks amounts to 114,257 (each potentially consisting of multiple instructions). As a reference, the ISA-full and ISA-RP variants operate at a coarser level of granularity, only considering stack cache control instructions. The number of these instructions depends on the number of defined functions and call sites in the program. In the considered benchmarks 8588 functions are defined, which are referenced by 10,475 call sites. In total 27,651 stack cache control instructions can be found in all benchmark programs combined (two for each function and one for each call site). The number of locations that may reflect changes in the underlying analysis information is thus reduced to about a quarter (24%). The FP variant essentially operates at the level of whole functions. This reduces the level of granularity even further to 8588 (8%) different locations.

In order to evaluate the transfer costs induced by the various mechanisms, a common level of granularity has to be chosen. The basic block level seems to be a reasonable choice, as it allows to demonstrate the performance of the underlying preemption mechanisms at a tight granularity and allows us to easily compare them against the optimized approach. However, it is important to note that all three preemption mechanisms have a diverging interpretation of the transfer costs when returning from a function. More precisely, the costs associated with the program points between an `sfree` of the callee and the corresponding `sens` of the caller differ from the optimized analysis. These program points rarely coincide with the beginning of basic blocks, since call instructions are not considered terminators for basic blocks in LLVM. The impact of this design choice is thus not explicitly captured by the presented numbers. At the same time, the concerned regions at most contain two program points, one before and

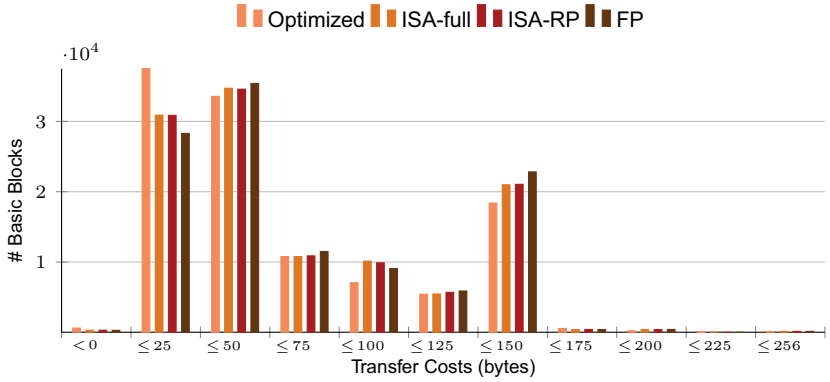

Fig. 15 Histogram comparing the transfer sizes (in bytes) for context restoration at basic blocks using the ISA-full, ISA-RP, and FP preemption mechanisms to the optimized analysis. Lower is better

one after the corresponding return instruction. The compiler also often manages to put the `sfree` instruction in the return's delay slot, which only leaves a single program point between the `sfree` and the `sens` executed immediately afterward.

Starting with the context restoration, we first observe that the proposed preemption mechanisms overall follow a similar pattern as the optimized approach, as illustrated by Fig. 15. The ISA-full and ISA-RP variants perform slightly better, showing only a moderate degradation in terms of precision. The difference between these two variants is insignificant, which indicates that the `RP` pointer is more profitable than the `DP`. This is not surprising, as the `RP` is regularly reset to 0 at every `sens` instruction, which limits the propagation of high `RP` values throughout large parts of a function. This is different from the `DP`, whose value evolves depending on stack cache accesses only and thus might be propagated throughout large parts of a function. A closer look reveals that, for these two approaches, the context restoration cost remains below 50b in the majority of the cases (57%). In only 0.8% of the basic blocks the context restoration exceeds 150b—this almost matches the optimized analysis. A noticeable drop is, however, observed for cases with very low transfer costs between 0b and 25b. The drop amounts to 6600 basic blocks, which represents about 21% of the 30,894 basic blocks in that cost range for the optimized analysis. The transfer costs of the respective basic blocks slightly increase, which corresponds to a slight shift to the right and explains the increased bar heights nearby. For instance, the restoration costs for 53% of these 6600 basic blocks now fall into the next higher cost range (26b to 50b) and another 24% of the blocks fall into the cost ranges after that (51b to 125b). The costs of the remaining cases then fall into the range from 126b to 150b. This indicates a moderate loss of precision, which is mainly due to the coarser granularity of these two approaches. The two approaches also succeed to conserve nearly 50% of the cases with negative restoration costs, i.e., reflecting potential runtime gains.

The FP approach generally follows the same trends. The shift in the diagram from the left side to the right is albeit more pronounced. The drop for the cost range from 0b to 25b amounts to 9211 basic blocks (30%). About half of the basic blocks appear in the next higher cost range (26b to 50b), while 28% of the cases can be found in the cost range from 51b to 125b. The remaining basic blocks (22%) move into the cost range above 125b, with two cases falling into the range from 151b to 200b. Despite the fact that the relative numbers appear to be close to the instruction-set-based

Table 2 Increase of restoration cost for the FP preemption mechanism in comparison to the optimized analysis, illustrating the movement of basic blocks to the right side of the histogram in Fig. 15

	< 0	≤ 25	≤ 50	≤ 75	≤ 100	≤ 125	≤ 150	≤ 175	≤ 200	≤ 225	≤ 256
< 0	−318	5	53	103	56	75	25	0	1	0	0
≤ 25	–	−9211	4566	1251	1177	188	2027	1	1	0	0
≤ 50	–	–	−2773	706	1618	45	404	0	0	0	0
≤ 75	–	–	–	−1342	461	111	759	0	11	0	0
≤ 100	–	–	–	–	−1301	772	529	0	0	0	0
≤ 125	–	–	–	–	–	−726	698	28	0	0	0
≤ 150	–	–	–	–	–	–	−9	3	6	0	0
≤ 175	–	–	–	–	–	–	–	−185	182	0	3
≤ 200	–	–	–	–	–	–	–	–	−7	0	7
≤ 225	–	–	–	–	–	–	–	–	–	−6	6
≤ 256	–	–	–	–	–	–	–	–	–	–	0

Smaller numbers are better

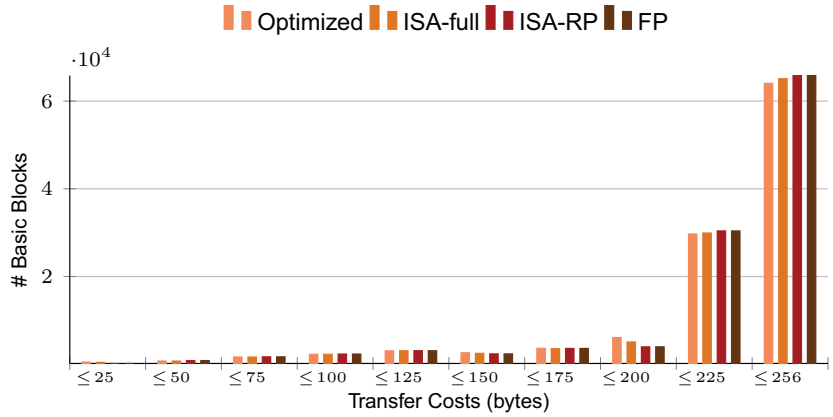

Fig. 16 Histogram comparing the transfer sizes (in bytes) for context saving at basic blocks using the ISA-full, ISA-RP, and FP preemption mechanisms to the optimized analysis. Lower is better

preemption mechanisms, the absolute numbers are considerably more pronounced. This explains, for instance, the noticeable peak for the cost range from 126b to 150b for this preemption mechanism. A detailed overview of the movements between the different cost ranges is illustrated by Table 2. The negative numbers on the diagonal indicate the number of basic blocks whose restoration costs were increased, while the positive numbers indicate to which cost range these basic blocks moved.

As for context saving, we can observe a very slight shift to the right for the ISA-full approach, as can be seen in Fig. 16. This mainly concerns 1141 basic blocks, whose transfer costs already were high (between 176b and 200b) even for the optimized analysis. The precision loss here mostly stems from the `DP` pointer, which suffers from the previously mentioned propagation of unfavorable values throughout the coarser regions. This shift is more pronounced for the ISA-RP and FP variants, since both do not exploit the `DP` parameter (which is conservatively set to 0). The number of basic blocks impacted almost doubles (2180). This also applies for other cost ranges and here

Table 3 Increase of saving cost for the FP preemption mechanism in comparison to the optimized analysis, illustrating the movement of basic blocks to the right side of the histogram in Fig. 16

	≤25	≤50	≤75	≤100	≤125	≤150	≤175	≤200	≤225	≤256
≤25	−318	145	156	17	0	0	0	0	0	0
≤50	–	−34	4	8	19	3	0	0	0	0
≤75	–	–	−122	113	4	0	0	1	0	4
≤100	–	–	–	−77	51	11	3	0	10	2
≤125	–	–	–	–	−41	38	2	1	0	0
≤150	–	–	–	–	–	−307	296	1	2	8
≤175	–	–	–	–	–	–	−332	51	281	0
≤200	–	–	–	–	–	–	–	−2180	2082	98
≤225	–	–	–	–	–	–	–	–	−1668	1668
≤256	–	–	–	–	–	–	–	–	–	0

Smaller numbers are better

in particular the range from 201b to 225b. A detailed breakdown of the movements in the histogram is given by Table 3.

From the results above, one can conclude that ISA-full provides some advantage over the other two mechanisms as it allows to exploit (at least to some degree) the analysis information concerning dead data. On the downside, the instruction-set-based approaches require additional changes to the hardware, the instruction set, as well as the compiler. In particular, the changes to the encoding of the stack cache control instructions might be problematic in practice. We will explore this issue using the Patmos processor and its instruction set as an example. Patmos instructions are encoded either using 64 bits or 32 bits depending on the corresponding instruction formats. The 64-bit format is dedicated to simple arithmetic instructions with a full 32-bit *long* immediate. All stack cache control instructions are encoded according to the 32-bit-wide Stack Control format (STC), which reserves 1 bit to indicate bundled (VLIW) instructions, 4 bits for the predicate operand, 5 bits to indicate the instruction format, and additional 4 bits for the instruction opcode. Consequently, 18 of the 32 bits are used to encode the instruction's operand (either an immediate or register index for the standard stack cache). Assuming a cache block size of 4b, this allows the stack cache to manage stack frame sizes of up to 1MB, which appears generous for most embedded applications. In order to implement the proposed instruction set extensions for the ISA-full and ISA-RP preemption mechanisms these 18 bits need to be split between either 3 (`RP`, `DP`, `k`) or 2 (`RP`, `k`) operands respectively. An even distribution would then either leave 6 or 9 bits for each operand. This would reduce the maximum stack frame size, but not the total stack cache size, to 256b or 2KB. The limit of 256b is sufficient for the experiments conducted here. Even when the stack cache size is essentially unbounded, most of the considered benchmarks exhibit a maximum stack frame size on the stack cache of 140b, while the largest stack frame encountered is merely 240b large. In a general setting, this restriction might, however, become limiting. Increasing the cache block size might remedy this

problem. The limit of 2KB, on the other hand, appears to be practical even for large embedded systems. The FP variant, based only on an additional internal stack cache control register, does not face such restrictions and might thus be easier to use in such larger systems. Overall, all of the three proposed preemption mechanisms appear to be practical.

6.4 Hardware implementation

We implemented all of the aforementioned hardware extensions in a Patmos hardware model. From the original model, specified in Scala, hardware is synthesized using the Altera Quartus II 13.1 tool suite for an an Altera DE2-115 board.

The implementation of the FP preemption mechanism requires an additional special register as well as some logic circuits that are needed to keep track of the current stack frame size. In particular, `sres` and `sens` instructions have to copy their argument to this special register, while `sfree` reset the register to zero. Only a dozen of code lines were needed to extend the stack cache model (originally about 500 code lines). Ignoring other components of the processor core, the hardware overhead in comparison to the original stack cache design is very minimal and is evaluated to 2.2 and 3.5% in logic cells and logic registers respectively. We also looked at the resulting overhead at the core level, which includes, among others, the computational units, a stack cache, a data cache, an instruction cache, and a local scratchpad memory. Once again, the hardware overhead is negligible and costs around 0.6 and 0.1% in logic cells and logic registers respectively. The impact on the maximum clock frequency, on the other hand, is surprisingly positive. We observed a slight improvement from 82 MHz to 83 MHz. Note that this improvement may be caused by some slight change in the complex heuristics employed by the synthesis software.

The overhead of the ISA-full and ISA-RP approaches is comparable to that of the FP approach. Some logic registers holding the instruction operands have to be reorganized, while the remaining changes are virtually identical. The modifications thus do not have a relevant impact on the hardware level.

We furthermore implemented the virtual stack cache extension. The results show that only very little additional hardware resources are required, i.e., most of the existing components are reused, some hardware resources can be removed, while new components need to be introduced. In total an additional hardware overhead of only 1.8% is incurred with regard to the entire Patmos core. Note that this does not reflect the gains due to merged memories. Moreover, the maximum processor frequency drops to 81 MHz, due to a longer combinatorial path for the address computation.

In conclusion, the implementation of the proposed hardware extensions is very simple and only incurs insignificant additional hardware costs.

7 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. Lee et al. (1998) use UCB

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, the worst-case preemption delay of a task is calculated. Altmeyer and Burguiere (2009) later introduced the notion of definitely-cached UCB (DC-UCB) to detect cache misses that are included in the CRPD bound as well as in 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 (Lee et al. 1998).

We now review some existing work using hardware support to optimize context switching. Tune et al. (2004) and Mische et al. (2010) introduce hardware support to optimize the context switching in real-time systems, but at the register file level. For instance, Tune et al. (2004) use 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 Soundararajan and Agarwal (1992), 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 by Olson et al. (2014), 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.

8 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 hold the stack data of 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 explicitly 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.

We furthermore proposed three different preemption mechanisms that allow the task scheduler to exploit the analysis information during context switching. Two of these mechanisms are based on an instruction set extension that attaches analysis information as operands to the stack cache control instructions. In comparison to the full static analysis these mechanisms may reflect changes in the analysis information at a much coarser level. Our experiments nevertheless showed that only a moderate loss in precision is incurred. A downside of these approaches is, however, that, in addition to support from the operating system, modifications to the compiler are required. An alternative solution relies solely on an additional stack cache control register that is updated by the stack cache control instructions in a transparent manner. Apart from the modifications to the task scheduler no additional tool support is required. This, however, comes at a cost: the granularity at which changes in the underlying analysis information can be reflected is limited to whole functions. This incurs an additional loss in precision.

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 (Jan et al. 2010; Chabrol et al. 2013), we also plan to evaluate the behavior of the stack cache, in particular the dynamic partitioning of the virtual cache and the associated prefetching techniques.

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

Abbaspour S, Brandner F (2014) Alignment of memory transfers of a time-predictable stack cache. In: Proceedings of the junior researcher workshop on real-time computing

Abbaspour S, Brandner F, Schoeberl M (2013) A time-predictable stack cache. In: Proceedings of the workshop on software technologies for embedded and ubiquitous systems

Abbaspour S, Jordan A, Brandner F (2014) Lazy spilling for a time-predictable stack cache: Implementation and analysis. In: Proceedings of the workshop on worst-case execution time analysis, OASICS vol 39. pp 83–92

Abbaspour S, Brandner F, Naji A, Jan M (2015) Efficient context switching for the stack cache: implementation and analysis. In: Proceedings of the international conference on real time and networks systems, ACM, RTNS ’vol 15, pp 119–128

Aho AV, Lam MS, Sethi R, Ullman JD (2006) Compilers: principles, techniques, and tools, 2nd edn. Addison-Wesley, Boston

Altmeyer S, Burguiere C (2009) A new notion of useful cache block to improve the bounds of cache-related preemption delay. In: Euromicro conference on real-time systems, ECRTS vol 09, pp 109–118

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

Baruah S (2005) The limited-preemption uniprocessor scheduling of sporadic task systems. In: 17th euromicro conference on real-time systems (ECRTS'05), pp 137–144

Burns A (1995) Advances in real-time systems. In: preemptive priority-based scheduling: an appropriate engineering approach, Prentice-Hall, Inc., Upper Saddle River, pp 225–248

Chabrol D, Roux D, David V, Jan M, Hmid MA, Oudin P, Zeppa G (2013) Time- and angle-triggered real-time kernel. In: Design, automation and test in Europe, DATE vol 13, pp 1060–1062

Cormen TH, Leiserson CE, Rivest RL, Stein C (2009) Introduction to algorithms, 3rd edn. MIT Press, Cambridge

Guthaus MR, Ringenberg JS, Ernst D, Austin TM, Mudge T, Brown RB (2001) MiBench: a free, commercially representative embedded benchmark suite. In: Proceedings of the workshop on workload characterization, WWC '01

Hecht MS, Ullman JD (1973) Analysis of a simple algorithm for global data flow problems. In: Symposium on principles of programming languages (POPL'73), ACM, pp 207–217

Jan M, David V, Lalande J, Pitel M (2010) Usage of the safety-oriented real-time oasis approach to build deterministic protection relays. In: Symposium on industrial embedded systems, SIES'10, pp 128–135

Jordan A, Brandner F, Schoeberl M (2013) Static analysis of worst-case stack cache behavior. In: Proceedings of the conference on real-time networks and systems, RTNS'13, pp 55–64

Kam JB, Ullman JD (1976) Global data flow analysis and iterative algorithms. J ACM 23(1):158–171

Lee CG, Hahn J, Seo YM, Min SL, Ha R, Hong S, Park CY, Lee M, Kim CS (1998) Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. IEEE Trans Comput 47(6):700–713

Li YTS, Malik S (1995) Performance analysis of embedded software using implicit path enumeration. In: Proceedings of the design automation conference, ACM, DAC '95, pp 456–461

Metzlaff S, Guliashvili I, Uhrig S, Ungerer T (2011) A dynamic instruction scratchpad memory for embedded processors managed by hardware. In: Proceedings of the architecture of computing systems conference, Springer, pp 122–134

Mische J, Uhrig S, Kluge F, Ungerer T (2010) Using smt to hide context switch times of large real-time tasksets. In: Proceedings of conference on embedded and real-time computing systems and applications, RTCSA'10, pp 255–264

Olson LE, Eckert Y, Manne S, Hill MD (2014) Revisiting stack caches for energy efficiency. Tech. Rep. TR1813, University of Wisconsin

Reineke J, Liu I, Patel HD, Kim S, Lee EA (2011) PRET DRAM controller: bank privatization for predictability and temporal isolation. In: Proceedings of the conference on hardware/software codesign and system synthesis, pp 99–108

Rochange C, Uhrig S, Sainrat P (2014) Time-predictable architectures. ISTE Wiley, London

Schoeberl M, Schleuniger P, Puffitsch W, Brandner F, Probst C, Karlsson S, Thorn T (2011) Towards a time-predictable dual-issue microprocessor: the patmos approach. In: Proceedings of bringing theory to practice: predictability and performance in embedded systems, OASICS, vol 18, pp 11–21

Soundararajan V, Agarwal A (1992) Dribbling registers: a mechanism for reducing context switch latency in large-scale multiprocessors. Tech. rep

Thorup M (2004) Integer priority queues with decrease key in constant time and the single source shortest paths problem. J Comput Syst Sci 69(3):330–353

Tune E, Kumar R, Tullsen DM, Calder B (2004) Balanced multithreading: Increasing throughput via a low cost multithreading hierarchy. In: Proceedings of the symposium on microarchitecture, MICRO'04, pp 183–194

Wang Y, Saksena M (1999) Scheduling fixed-priority tasks with preemption threshold. In: Real-time computing systems and applications, 1999. RTCSA '99. sixth international conference on, pp 328–335

Wilhelm R, Grund D, Reineke J, Schlickling M, Pister M, Ferdinand C (2009) Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. Trans Comput-Aided Des Integr Circ Syst 28(7):966–978