Static Analysis of Worst-Case Stack Cache Behavior

Alexander Jordan
Compilers and Languages Group
Institute of Computer Languages
Vienna University of Technology
ajordan@complang.tuwien.ac.at

Florian Brandner and Martin Schoeberl
Embedded Systems Engineering Section
Dep. of Applied Mathematics and Computer Science
Technical University of Denmark
{flbr, masca}@imm.dtu.dk

ABSTRACT

Utilizing a stack cache in a real-time system can aid predictability by avoiding interference that heap memory traffic causes on the data cache. While loads and stores are guaranteed cache hits, explicit operations are responsible for managing the stack cache. The behavior of these operations can be analyzed statically. We present algorithms that derive worst-case bounds on the latency-inducing operations of the stack cache. Their results can be used by a static WCET tool. By breaking the analysis down into subproblems that solve intra-procedural data-flow analysis and path searches on the call-graph, the worst-case bounds can be efficiently yet precisely determined. Our evaluation using the MiBench benchmark suite shows that only 37% and 21% of potential stack cache operations actually store to and load from memory, respectively. Analysis times are modest, on average running between 0.46s and 1.30s per benchmark, depending on the size of the stack cache.

Categories and Subject Descriptors

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

General Terms

Theory, Algorithms, Measurement, Performance

Keywords

Program Analysis, Stack Cache, Real-Time Systems

1. INTRODUCTION

Real-time systems need a time-predictable computer platform to enable static worst-case execution time (WCET) analysis of the associated software. Caches pose a particular challenge with regard to WCET analysis, as a potentially huge state space reflecting a long execution history of the program has to be tracked. A solution to this problem is splitting the caches according to access patterns. For instance, it is recommended to split data and instruction caches to avoid interference [4]. Data caches can similarly be split to adapt the caching strategy to the access patterns of different memory areas [11], e.g., stack and static data.

Stack accesses via a standard data cache are typically hard to analyze, as the accessed addresses depend on the value of the stack pointer. The cache analysis thus needs to first find potential address ranges for the stack pointer, which inherently depend on the nesting of function calls and thus requires high levels of context-sensitivity. Having a closer look at typical usage patterns of stack data, one finds that (1) data in the stack frame of a function is usually exclusive to that function and (2) the data in the stack frame is usually only accessed while the function is active. This suggests a caching strategy that follows the nesting of function calls such that the stack frame of the active function is readily available in the cache. Recent work proposed such a *stack cache* using a rather simple ring buffer [1]. Even though simple, this cache design handles up to 75% of the dynamic data accesses of embedded benchmarks. In this work, we explore analysis techniques for this stack cache design.

The stack cache is explicitly controlled by the compiler (or programmer) using dedicated instructions to *reserve* and *free* space on the stack cache for the stack frames of functions. During a reserve operation the requested space might exceed the cache's capacity and cause *spilling* of (parts of) the stack cache's content. If the stack frame of a function has been spilled to memory it can be reloaded, or *filled*, using an *ensure* operation, e.g., when a function becomes active after returning from another function. Using these stack control primitives, a sliding window of cached data is realized that follows the nesting of function calls and ensures that all accesses to the stack data of a function are guaranteed hits with constant latency, independent of the precise address of the access and independent of the current value of the stack pointer. This results in a considerably simpler analysis model. The task of the stack cache analysis is to determine the worst-case filling and spilling of the *ensure* and *reserve* primitives by defining two analysis problems.

Both analysis problems depend on the *occupancy* level of the stack cache before the respective control primitives. An interesting observation here is that information on this occupancy can be pre-computed locally within functions using the *stack cache displacement* at function calls, i.e., the amount of stack data potentially evicted from the stack cache while executing the call. The analysis problem for

RTNS 2013 , October 16 - 18 2013, Sophia Antipolis, France

http://dx.doi.org/10.1145/2516821.2516828.

ensures is thus entirely context-insensitive, while the analysis for reserves can be performed on the program's call graph – without reanalyzing the instructions within the functions.

The paper is organized as follows: Section 2 provides background on the stack cache, the program representation for the analysis, data-flow analysis. The following section describes the stack cache analysis. Sections 4 and 5 describe some extensions and applications of the analysis. We evaluate our approach before discussing related work and concluding in Section 8.

2. BACKGROUND

In this section, we describe the general functionality of a stack cache. A detailed description of the implementation in the Patmos processor [12] can be found in [1].

2.1 Stack Cache

Compared to accessing heap storage, a stack cache enables accesses within a stack-allocated memory region to complete with a small and predictable latency. The stack cache relevant load and store operations address values relative to a base address (*stack pointer*). Furthermore, the model we consider herein uses three primitives to manage the stack cache. These are responsible for *reserving* space for stack-allocated data, *freeing* the same space again, and *ensuring* that data, which will be subsequently used, is available.

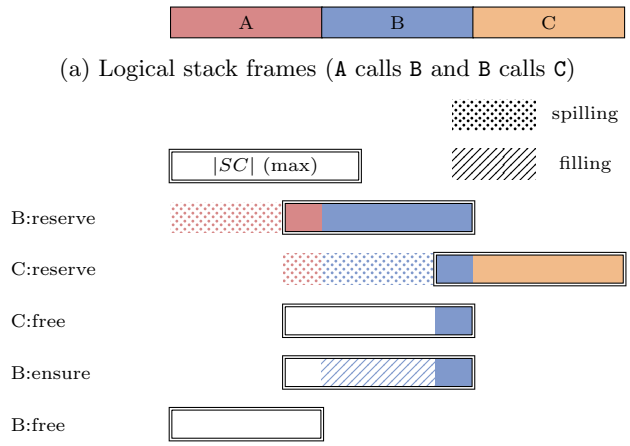
Conceptually, the stack in question grows towards lower addresses and is divided into a set of equally sized blocks. *The number and actual size of the blocks are properties left to the implementor.* Each operation takes an argument representing a block count. In detail, the semantics for stack cache manipulation are as follows:

Reserve: `sres` k
: Allocates an area of k blocks in the stack cache and sets the stack pointer to the beginning of this region. If k and the number of currently reserved blocks counted together exceed the capacity of the stack cache, some blocks need to be *spilled* (i.e., saved to main memory). The stack cache always selects a minimal number of blocks from the earliest reserved (highest address) blocks for spilling.

Free: `sfree` k
: Discards the k most recently reserved (lowest address) blocks and adjusts the stack pointer accordingly. The contents of the stack cache are not changed.

Ensure: `sens` k
: If not all of the k blocks at the current stack pointer are available in the stack cache, (only) the missing blocks are *filled* (i.e., loaded from main memory).

Only an `sres` or `sens` operation may access main memory and can cause a variable-time latency. Particularly WCET analysis benefits from this characteristic, as it is sufficient to know (an upper bound on) the number of blocks being transferred, to statically calculate the latency. Our model assumes *uniform cost* for the transfer of every stack cache block, but this can be trivially adapted to the behavior of any implementation (as long as it remains predictable). *This stack cache model can be implemented in several ways. In the concrete case of the Patmos processor, all three stack cache operations are exposed in the instruction set. Also, the hardware is responsible for spilling data to and loading data from external memory. Without any impact on the results herein, some (or all) functionality could be shifted to software.*



(a) Logical stack frames (`A` calls `B` and `B` calls `C`)

(b) Stack cache behavior

Figure 1: Sliding window visualization of the stack cache with three functions

The rationale for splitting the `sens` from the `sfree` operation can be easily seen when unwinding the allocated stack: when the unwinding logic issues a series of `sfree`s, it is not interested in any values on the stack. Thus, filling the stack cache from main memory at this point is unnecessary.

Stack cache operations can be placed at arbitrary points in the program (with some inherent rules for legal programs). Commonly they appear around function calls, i.e., `sres` is placed after function entry, `sfree` before the return, and if required, the caller places `sens` immediately after the call. In the following, when we present our analysis, we first assume the simple case of placement being restricted around calls. A generalization is discussed in Section 4.

We further assume that every function reserves a value k less than, or equal to the size of the stack cache. *This requirement is guaranteed by the compiler.*

The spill and fill operations inherently provide a WCET bound. In the most pessimistic manner, we could assume that at every `sres` and `sens` the entire size of the argument k, needs to be spilled and filled respectively. In the following, our analyses shall reduce this worst-case bound.

Example 1. *Figure 1 visualizes the operation of a stack cache for a series of function calls. In the first row of Figure 1(b), `A` reserves its stack space, which is partly evicted when `B` is called. Calling `C` then partly evicts the stack space of `B`, while the remainder of the stack cache is allocated to `C`'s data. `C` frees its space before returning, leaving the cache empty except for those parts of `B`'s stack space that were not evicted. `B` then reloads its stack data (assuming it needs to access some of it). The stack cache is now almost fully occupied by `B`'s stack space, while the rest remains empty. Finally, when `B` returns, the stack cache becomes completely empty and `A` would need to reload all of its previously evicted stack data.*

2.2 Program Representation

To calculate worst-case bounds for stack cache operations, we integrate two analyses on different program levels. Function-local (i.e., intra-procedural) stack cache analysis targets the control-flow graph of each function individually. To model the inter-procedural effects of the stack cache we use the program's *call graph*.

DEFINITION 1. *The control-flow graph* $CFG = (N, E, r, e)$ *is a directed graph, with node set* N *and edge set* E*. Nodes* $n \in N$ *represent instructions, edges* $(m, n) \in E$ *control-flow between them, and artificial nodes* r *and* e*, which denote the unique start and end node in* CFG*, respectively. Additionally we define* $Preds(n) = \{m \mid (m, n) \in E\}$*, the set of immediate predecessors of* n*.*

Note that only a small subset of instructions is relevant for the analysis. It consist of *calls* to other functions and the three instructions that manipulate the stack cache: `sres`, `sfree`, and `sens`. All remaining instructions in the CFG (including all loads and stores) are transparent with regard to the stack cache analysis.

DEFINITION 2. *The call graph* $CG = (N, E, s, t)$ *is a directed graph, consisting of nodes in* N*, the distinguished source and sink nodes* $s, t \in N$*, and directed edges* E*. Nodes represent functions of a program, while each edge* $(m, n) \in E$ *represents an individual call (site) from function* m *to* n*.*

The source and sink nodes exist for convenience and are established by connecting s to the entry function (e.g., `main`), while all functions containing a call-free path, i.e., a path through the function's CFG that does not contain a call instruction, are connected to the sink node.

2.3 Data-flow Analysis

Data-flow analysis is used to gather information about the behavior of a program without executing it. For a specific property (e.g., variable liveness) the effect of every instruction (i.e., every node in some CFG) and its dependencies on other instructions are examined. A common way to perform data-flow analysis is to attach equations that relate input and output information to every node, then solving these equations for every node repeatedly, until their results no longer change (i.e., a *fixpoint* for the whole system is reached). Two parts of our stack cache analysis closely resemble this concept and are thus best described in the standard way for data-flow analyses.

We are interested in bounds for the stack cache occupancy at certain points in the program. I.e., a value from a finite subset of $\mathbb{N}^0$ ($\mathbb{N} \cup \{0\}$) bounded from above by the number of blocks in the stack cache $|SC|$. Formally, the value domain for the analysis is $\mathcal{D} = \{0, \ldots, |SC|\}$. To set up the system of equations, every instruction i in the CFG is associated with two variables, $IN(i)$ and $OUT(i)$, which can take values from $\mathcal{D}$ and represent the occupancy bound before and after the instruction respectively. Data-flow equations (also known as *transfer functions*) between the variables define (1) the change of the worst-case occupancy bound induced by instructions and (2) how to merge those bounds at control-flow joins. The algorithm to solve these equations will not be shown here for brevity. Together with a detailed description of the related theoretical foundations, it can be found in standard literature on the topic [2, 8].

3. STACK CACHE ANALYSIS

In order to determine the worst-case behavior of a real-time program utilizing a stack cache, two analysis problems have to be solved: (1) for every `sres`-instruction in the program, the worst-case *spilling* behavior has to be computed and (2) for every `sens`-instruction, the worst-case *filling* behavior has to be computed. We refer to these two problems as the *reserve* and *ensure* analyses respectively.

Reserve analysis We will first tackle reserve analysis, which has to consider the (maximum) stack cache occupancy on all potential executions leading up to an `sres`-instruction. This information cannot be computed using local information alone and thus requires a context-sensitive analysis. The worst-case behavior of an `sres` depends on the stack cache occupancy of the current function, which in turn depends on the occupancy at the invoking call instruction, thus on the occupancy of the function surrounding the call, and so on until the program's entry point is reached. In order to compute this information efficiently, the analysis has to account for the change of stack occupancy that occurs between the entry of a function and each call instruction. Note that the stack occupancy not necessarily increases here. If, for instance, parts of the stack cache are intermittently spilled, the occupancy may also decrease. A key observation here is, that the worst-case occupancy at call sites can be bounded by a function-local analysis, which is based on the *minimum displacement* occurring between the function entry and the call. This displacement determines how much of the allocated stack space remains allocated in the worst-case when calling a function. Using the results of this preliminary analysis, a function-local data-flow analysis can propagate *worst-case occupancy bounds* along all paths from a function's entry to all call sites, initially assuming a fully occupied stack cache. The final step of reserve analysis is an inter-procedural data-flow analysis that propagates context-sensitive stack occupancy on the program's call graph.

Ensure analysis The analysis of ensure instructions similarly relies on the stack displacement at function calls. In contrast to the reserve analysis, however, the maximum displacement is required, i.e., how much of the stack space allocated by the current function is spilled to main memory by another function in the worst-case. Based on this preliminary analysis of the maximum displacement, the ensure analysis can be formulated as a function-local data-flow analysis that tracks the amount of stack space belonging to the current function that has been potentially spilled and thus needs to be reloaded by ensure instructions.

EXAMPLE 2. *Consider a stack cache of size* 4 *and the example program in Figure 2. The stack cache occupancy at the entry of function* `C` *depends on its three calling contexts (*i_5^A*,* i_3^B*,* i_5^B*). We are only interested in the last calling context (*i_5^B*) for now. By assuming a full stack cache at the entry of the calling function* `B` *and examining the path leading to the call instruction, we realize that the minimum displacement along this path can be used to bound the worst-case occupancy for this context. Since the* `sres`*-instruction at the entry of* `B` *cannot further increase the occupancy, the first*

```
(i1A) func A() {
(i2A)  sres 2;
(i3A)  B();
(i4A)  sens 2;
(i5A)  C();
(i6A)  sens 2
(i7A)  sfree 2;
(i8A) }
```

(a) Code of A

```
(i1B) func B() {
(i2B)  sres 3;
(i3B)  C();
(i4B)  sens 3
(i5B)  C();
(i6B)  sens 3
(i7B)  sfree 3;
(i8B) }
```

(b) Code of B

```
(i1C) func C() {
(i2C)  sres 2;
(i3C)  sfree 2;
(i4C) }
```

(c) Code of C

Figure 2: Program consisting of 3 functions, reserving, freeing and ensuring space on the stack cache.

call to `C` *(i_3^B) displaces* 2 *blocks from the full stack cache, resulting in a worst-case occupancy of* 2 *when returning. Due to the displacement, the worst-case occupancy at i_5^B cannot be higher than* 2*. The* `sens`*-instruction (i_4^B) again raises the occupancy to* 3*, which bounds the occupancy at i_5^B.*

Finally, during the context-sensitive part of the reserve analysis, we use this bound to derive the occupancy at the call site without having to reanalyze the instructions in `B`*. When the occupancy at the entry of* `B` *plus the locally reserved space exceeds the bound, it is sufficient to propagate the bound to* `C`*. Otherwise, the smaller occupancy level has to be propagated.*

The ensure analysis similarly first pre-computes the maximum displacement of function calls and then propagates information within functions. Function `A` *for instance, reserves* 2 *blocks on the stack cache. The displacement of the call to functions* `B` *is determined to be* 4*. The ensure analysis thus finds that the entire stack space of* `A` *may have been evicted after returning from* `B`*. The* `sens`*-instruction i_4^A thus has to fill* 2 *blocks from main memory in the worst-case.*

Combined Analysis The reserve and the ensure analysis both rely on related underlying computations and can be combined. The combined analysis then consists of three main phases: (1) the pre-computation of the minimum and maximum displacements on the call graph, followed by (2) the function-local data-flow analyses for the reserve and ensure analysis, and finally (3) the context-sensitive reserve analysis on the program's call graph. We will discuss each of the phases in the following sub-sections.

3.1 Computing the Stack Cache Displacement

Computing the minimum displacement of a call site, as required by the reserve analysis, corresponds to a shortest path search in the call graph, where edges are annotated with weights representing the amount of stack space reserved by the calling function. We will later see that ensure analysis depends on the maximum displacement, which can be computed in the same way, but performing a *longest path* search. Assuming the placement of ensure and free instructions as defined in Section 2 for now, this technique can easily be extended to the larger class of *well-formed* programs as described in Section 4.

Algorithm 1 Algorithm to compute the displacement at call sites (`ComputeMinimumDisplacement`, `ComputeMaximumDisplacement`).

Require: $ACG = (N, E, s, t)$ ... An annotated call graph.
Ensure: The minimum/maximum displacement at each call site is returned.

1: **foreach** $n \in N$ **do**
2: ▶ Sub-graph with zero costs.
3: $N_0 = \{v_0 | v \in N, v \neq t\}$
4: $E_0 = \{(u_0, v_0, 0) | (u, v, w) \in E, v \neq t\}$
5: ▶ Weighted sub-graph.
6: $N_W = V \setminus \{s\}$
7: $E_W = E \setminus \{(u, v, w) \in E | u = s \vee v = s\}$
8: ▶ Edges to transition between sub-graphs.
9: $E_T = \{(u_0, n, 0) | (u, n, w) \in E\}$
10: **let** $ACG' = (N_0 \cup N_W, E_0 \cup E_W \cup E_T, s_0, t)$ **in**
11: $D[n] =$ `ComputePathOver`(ACG', n)
12: **return** D

Definition 3. *The annotated call graph $ACG = (N, E, s, t)$ is a call graph, with weighted edges $(u, v, w) \in E \subseteq N \times N \times \mathbb{N}^0$. The weight w represents the stack space reserved in function u. The edge connecting s to the entry function is assumed to have weight* 0*, while edges incident to t are normally annotated with the stack space reserved in the respective functions.*

Definition 4. *For a call site (u, v, w) of an annotated call graph $ACG = (N, E, s, t)$ the minimum (maximum) displacement is given by the shortest (longest) tail from v to the sink node t for any path of the form $(s, \ldots, u, v, \ldots, t)$.*

Acyclic Call Graphs In the case of acyclic call graphs, i.e., programs without recursion, the minimum and maximum displacement of all nodes can be computed using dynamic programming [3] in linear time ($O(|N| + |E|)$). The nodes are traversed in reverse topological order, computing each node's displacement from the displacement of their respective successors in the graph.

3.1.1 Call Graphs With Recursion

Shortest and longest path searches for programs with cyclic call graphs, i.e., with recursion, can be modeled using *integer linear programming* (ILP). The technique resembles approaches from standard WCET analysis [9, 10]. Instead of searching for the shortest (longest) path, the path with the shortest (longest) tail, where the tail starts with a specific node, needs to be computed. The reason for this will become clear when we later introduce *user constraints*.

We model the computation of these paths on a transformed call graph, which is constructed by duplicating the original graph twice (see Alg. 1 and Fig. 3). One duplicate represents the paths' tails and is thus associated with the weights of the original graph (l. 6). The other duplicate is associated with zero costs (l. 3) and represents the heads of the paths. The two sub-graphs are connected only at the node whose displacement is to be computed, i.e., edges lead from the node's duplicate in the sub-graph with zero costs to the respective duplicate in the weighted sub-graph. The shortest or longest path search is then performed on this transformed graph (l. 11). Note that not all nodes and edges need to be duplicated in practice. Only the nodes and edges that may appear on a path from s to t and passing through the currently considered node need to be considered (this optimization is not shown here for brevity).

For the path search `ComputePathOver` (Alg. 1, l. 11) takes the transformed call graph ACG' and a target node n as arguments and constructs an ILP, which models the nesting of function calls that can be observed (in the worst-case) when executing the program. Each ILP variable represents the number of times a function has been called, or more precisely, how often a specific call site was used to call a function, in this nesting. The ILP variables can be seen as representing *flow* that has to meet constraints. For instance, the flow entering a function has to leave that function again, i.e., the sum of the adjacent ILP variables needs to be equal.

The (flow) constraints of the ILP for the transformed call graph $ACG' = (N', E', s', t')$ are formally defined as:

$$\left.\begin{aligned} \mathcal{V}(v) &= \sum_{e=(u,v,w)\in E'} \mathcal{V}(e) \\ \mathcal{V}(v) &= \sum_{f=(v,u,w)\in E'} \mathcal{V}(f) \end{aligned}\right\} \quad \forall v \in N' \qquad (1a)$$

$$\mathcal{V}(s') = 1 \tag{1b}$$
$$\mathcal{V}(t') = 1 \tag{1c}$$
$$\mathcal{V}(n) > 0 \tag{1d}$$

With integer variables:

$$\mathcal{V}(e), \mathcal{V}(v) \in \mathbb{N}^0 \quad \forall e \in E', \forall v \in N' \tag{1e}$$

Variables (1e) are created for every node and every edge in the transformed call graph and the functions $\mathcal{V}(n)$ and $\mathcal{V}(e)$ map nodes and edges of the graph to their respective ILP variables. For each node the incoming and outgoing flow has to match (1a). In order to get a legal nesting of function calls including the node n three more flow constraints have to be added. These force the flow at the source (1b) and sink node (1c) to 1 and the flow over the target node (1d) to be non-zero.

The optimization objective is either to minimize the objective function, when the shortest path is to be computed:

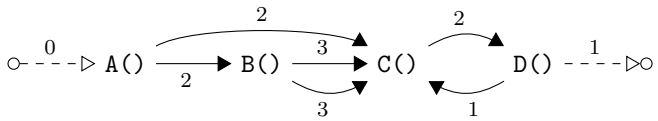
$$\min \sum_{e=(u,v,w)\in E'} w\mathcal{V}(e), \tag{1f}$$

or maximize the same function for the longest path search:

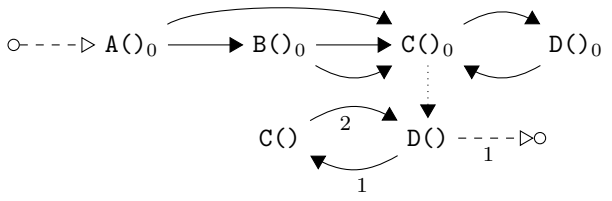
$$\max \sum_{e=(u,v,w)\in E'} w\mathcal{V}(e) \tag{1g}$$

The ILP formulation presented above expresses all possible nestings of function calls that might potentially be observed when the program is executed. However, in particular when the objective function is to be maximized, many of these nestings cannot actually occur in practice, e.g., because the recursion in the program is limited to a certain depth. This information can be supplied by the user and expressed as an additional constraint for the affected ILP variable. More complex *user constraints* can be expressed by linear equations using different variables of the ILP.

Example 3. *Consider the annotated call graph in Figure 3(a). Function* `D` *has a call-free path and is thus connected to the sink. Assuming that the displacement of* `D` *shall be determined, Figure 3(b) shows two duplicates of the graph, where the sub-graph with zero costs is above the weighted one. The edge that allows to transition between the two sub-graphs (E_T) is highlighted using a dotted line. (Only nodes and edges on a path from the source to the sink and passing through* `D`*, as well as non-zero edge weights are shown.)*



(a) Call Graph



(b) Transformed Graph

Figure 3: A recursive call graph and the corresponding transformed graph to compute `D`'s displacement.

Assuming that function `C` *can appear at most* 10 *times on any legal nesting of function calls, we can add a user-specific constraint such as:* $\mathcal{V}(C()) + \mathcal{V}(C()_0) < 11$. *(Due to space restrictions the full ILP cannot be shown here.) Since* `D` *can only be invoked by* `C`*, this implies a maximum displacement for* `D` *of* $9 \cdot 1 + 9 \cdot 2 + 1 = 28$. *Because the constraint on* `C` *only defines an upper bound, it is easy to see that the minimum displacement for* `D` *(shortest path) is* 1.

Certainly, as long as a user constraint does not interfere, shortest path search can be solved efficiently on the original graph (e.g., using *Dijkstra's algorithm*). Also note that the approach for cyclic and acyclic graphs can be combined by collapsing the nodes of the individual *strongly connected components* (SCCs) of the original call graph into representative nodes. This results in an acyclic graph that can be traversed as described in Section 3.1. Whenever the representative of an SCC is visited during the traversal, an ILP is constructed as shown by Algorithm 1 based on the subgraph induced by the SCC in the original call graph.

3.2 Bounding the Stack Cache Occupancy

Once the minimum displacements are computed, the stack cache occupancy can be bounded for each call site by an intra-procedural data-flow analysis. Assuming a full stack cache at function entry, the analysis propagates an upper bound on the stack occupancy (o_{bound}) along all paths from the entry to the call sites within the function. The minimal displacement of call instructions as well as `sens`-instructions along the path may have an impact on this bound, while all other instructions can safely be ignored.

A standard data-flow analysis based on the framework defined in Section 2.3, with domain $\mathcal{D}$ is then performed on the CFG. The transfer functions for an instruction i are:

$$d_{loc}(i) = \min(|SC|, d_{min}(i)) \tag{2}$$

$$OUT(i) = \begin{cases} \max(IN(i), k) & \text{if } i = \texttt{sens k} \quad (3a) \\ \min(IN(i), |SC| - d_{loc}(i)) & \text{if } i = \mathit{call} \quad (3b) \\ IN(i) & \text{otherwise} \quad (3c) \end{cases}$$

When i is an ensure instruction (`sens k`) with its argument $k \in \mathbb{N}^0$, the current upper bound is increased to k (3a). The transfer function of a call i depends on the size of the stack cache $|SC|$ and the minimal displacement $d_{min}(i)$ of the functions that are potentially invoked by the call (3b). ($d_{min}(i)$ is computed as described in Section 3.1.) For all other kinds of instructions the transfer function is the identity function (3c).

Between instructions, the o_{bound} also needs to be propagated (i.e., from the OUT-values of the predecessors of an instruction to the instruction's IN-value). This is done using the transfer functions:

$$IN(i) = \begin{cases} |SC| & \text{if } i = r \quad (4a) \\ \max\limits_{p \in Preds(i)} (OUT(p)) & \text{otherwise} \quad (4b) \end{cases}$$

At joins in the CFG, the maximum (max) of all incoming values is used as the *meet operator* (4b). In order to model a full cache at the entry of the current function, the IN-value of the first instruction in the function's CFG r is initialized with the full size of the stack cache $|SC|$ (4a). For the remaining initial values at each program point, we use the value $0 \in \mathcal{D}$ which is the neutral element with respect to max.

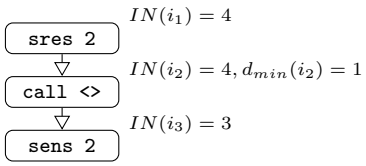


Figure 4: Propagation of stack cache bounds in a function-local control-flow graph.

EXAMPLE 4. *Given the control-flow graph shown in Figure 4 and a stack cache size of* 4*, the analysis starts by associating the IN-value of instruction* i_1 *with the full stack cache size (*$|SC| = 4$*). Since* i_1 *is an* `sres`*-instruction the identity function leaves the stack cache occupancy bound unchanged for the following instruction.* i_2 *is a call with a minimal displacement of* 1 *(*$d_{min}(i_2) = 1$*). Applying the corresponding transfer function (see Eq. 3b) yields a stack occupancy of* 3 *after the call, which is propagated to the IN-value of instruction* i_3*.*

3.3 Worst-Case Spilling of Reserves

Having determined bounds for the stack cache occupancy, we can finally define `AnalyzeReserves`, which computes the worst-case spilling of reserve instructions within the program. More precisely, an `sres`-instruction will cause spilling when the occupancy before the reserve is too high and the requested space is not available in the stack cache. As we have noted before, the maximum occupancy at function entry is specific to a calling-context and depends on the accumulated occupancy of the nested function calls leading to the entry. On the other hand, we need to account for spilling caused by calls to other functions that may have evicted parts of the stack cache before the execution reaches the respective reserve. Given the o_{bound} value (see Section 3.2) for a call site, the context-dependent analysis of its maximum stack occupancy becomes simple: when a new stack occupancy is derived for the entry of the enclosing function, either (1) the new stack occupancy plus the locally reserved space is propagated to the call site or (2) the bound is propagated to the call site, which ever is smaller. From the context-dependent stack occupancy a graph can be constructed that can be used to represent the spill costs of the `sres`-instructions of the individual contexts:

DEFINITION 5. *The* spill cost analysis graph *is a directed graph* $SCA = (ACG, N_c, E_c)$ *consisting of nodes in* N_c *representing occupancy-annotated calling-contexts and edges in* $E_c \subseteq N_c \times N_c$ *that correspond to call sites of the call graph* ACG*. The nodes are pairs* $(n, o) \in N_c$*, where* n *is a node of* ACG *and* $o \in \mathbb{N}^0$ *is the context's stack cache occupancy.*

Algorithm 2 constructs an SCA graph from an annotated call graph (see Def. 3) using a simple work list. The analysis starts at the sink node s, which is assumed to have a stack occupancy of 0 (l. 3–2). From this initial context other SCA contexts are derived by processing one context from the work list at a time (l. 7). The context is removed from the work list and new contexts are constructed considering the current occupancy and the weighted call sites associated with the corresponding call graph node of the context (l. 11). Note the use of the occupancy bound that was computed before (see Section 3.2). If the so discovered

Algorithm 2 Constructing the Spill Cost Analysis Graph (SCA), as part of `AnalyzeReserves`.

Require: $ACG = (N, E, s, t)$ ... An annotated call graph.
O_{bound} ... The occupancy bounds of call sites.
Ensure: Context-sensitive stack cache occupancy derived for the $SCA = (ACG, N_c, E_c)$.

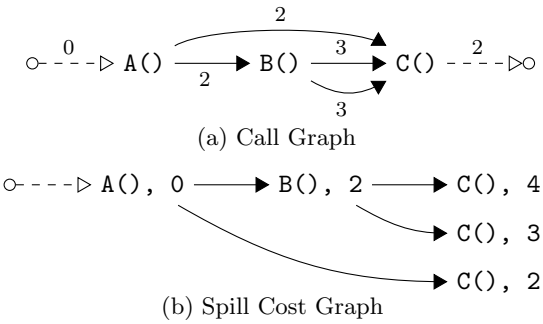
```
1: ▶ Initialize the SCA graph and work list
2: E_c = ∅; N_c = {(s, 0)}
3: W = {(s, 0)}
4: ▶ Iteratively derive new SCA contexts
5: while W ≠ ∅ do
6:   ▶ Process some context from the work list
7:   let c = (u, o) ∈ W in
8:     W = W \ c
9:     foreach e = (u, v, w) ∈ E do
10:      ▶ Derive a potentially new SCA context
11:      let c' = (v, min(o + w, O_bound[e])) in
12:        ▶ Update the work list
13:        if c' ∉ N_SCA then
14:          W = W ∪ c'
15:        ▶ Update the SCA graph
16:        N_c = N_c ∪ c'
17:        E_c = E_c ∪ (c, c')
18: return SCA
```

contexts were not yet known, they are added to the work list (l. 14). Finally, the SCA graph is updated to cover the newly discovered contexts (l. 16).

Using the occupancy information of the SCA graph, the spill costs of the individual reserve instructions in the program can immediately be derived. Assuming uniform per-block spill cost $\widehat{c}_s$ and a context $c = (n, o)$ of an `sres`-instruction i that reserves k blocks, the spill cost is:

$$spillcost(i, c) = \widehat{c}_s \cdot \max(0, o + k - |SC|) \qquad (5)$$

EXAMPLE 5. *Assuming a stack cache with* 4 *blocks, the spill cost analysis graph shown in Figure 5 is constructed from the example program from Figure 2. While only a single context is constructed for the functions* `A` *and* `B` *respectively, three different contexts are created for* `C`*. The stack occupancy of these contexts are* 4*,* 3*, and* 2 *blocks. This results in a worst-case spilling of* 1 *and* 2 *blocks respectively for the first two context. No spilling is performed in the last context. Note that the edges in the SCA graph correspond to edges in the call graph (shown in Figure 5(a)).*



(a) Call Graph

(b) Spill Cost Graph

Figure 5: Annotated call graph and spill cost graph of the program from Figure 2.

3.4 Worst-Case Filling of Ensures

What remains is to analyze the filling behavior of `sens`-instructions. Before we define the analysis formally, let us consider the cases when refilling the stack cache becomes necessary. In the model specified in Section 2, an ensure instruction always refers to the stack space reserved by the `sres`-instruction at the entry of the current function. This means that filling may only become necessary when some of the locally reserved stack space is evicted from the cache on any path between the reserve and the ensure. It thus suffices to examine the minimum occupancy induced by the *maximum displacement* along all paths between the two instructions, while accounting for the effects of intermittent ensure instructions. An interesting aspect of this analysis problem is that once the maximum displacement of individual call instructions is known (by solving the longest path problem from Section 3.1), the analysis can be performed locally for each function. This makes ensure analysis considerably easier (context-insensitive) compared to its reserve counterpart. Note that a similar property holds for well-formed programs discussed in Section 4.

Formally, `AnalyzeEnsures` has to solve a data-flow analysis problem similar to that described in Section 3.2. The algorithm takes two arguments: the function's CFG G and a mapping from call sites to their respective maximum displacement D_{max}. The analysis is based on the framework defined in Section 2.3, with domain $\mathcal{D}$. The transfer functions are as follows:

$$d_{loc}(i) = \min(|SC|, d_{max}(i)) \tag{6}$$

$$OUT(i) = \begin{cases} k & \text{if } i = \texttt{sres k} \\ \max(IN(i), k) & \text{if } i = \texttt{sens k} \\ \min(IN(i), |SC| - d_{loc}(i)) & \text{if } i = \mathit{call} \\ IN(i) & \text{otherwise} \end{cases} \tag{7}$$

$$IN(i) = \begin{cases} 0 & \text{if } i = r \\ \min_{p \in Preds(i)}(OUT(p)) & \text{otherwise} \end{cases} \tag{8}$$

Opposite to Section 3.2, the goal is to find a *minimum* bound. Thus the meet operator in (8) accordingly changes to min and the initial value is $|SC|$.

With the solution of the analysis above, we can compute the worst-case filling cost for every `sens`-instruction (assuming uniform per-block fill cost $\widehat{c}_f$):

$$fillcost(i) = \widehat{c}_f \cdot (OUT(i) - IN(i)) \tag{9}$$

Example 6. *Consider function `A` in Figure 2(a) and a stack cache with size $|SC| = 4$. The ensure analysis starts by applying the transfer function (Eq. 7) of the `sres`-instruction i_2^A. This causes a minimum stack occupancy of 2 to be propagated to the following call i_3^A to `B`. From the analysis of the maximum displacement it is known that `B` might spill the entire content of the stack cache since ($d_{max}(i_3^A) = 5$) including what `A` reserved. The minimal stack occupancy after the call thus has to be assumed to be 0 in the worst-case, which is propagated to $IN(i_4^A)$ of the following ensure. As the stack cache might be empty, the ensure has to reload the 2 blocks specified as its argument and its OUT-value thus becomes 2. The next call instruction i_5^A invoking `C` has a maximum displacement $d_{max}(i_5^A) = 2$. The analysis determines that $|SC| - 2 = 2$ and thus equal to the minimum occupancy (Eq. 7), which therefore does not change (i.e., the content of `A`'s and `C`'s stack frames both fit into the stack cache). Consequently it is not necessary for the final ensure instruction i_6^A to fill any data. The instruction could even be removed without any side-effect.*

3.5 Combining the Analyses

From the individual analyses above a simple algorithm that solves both stack cache analysis problems at the same time can be devised (see Alg. 3). The algorithm takes a call graph and a set of control-flow graphs (one for each function in the program) as input and associates every `sens`- and `sres`-instruction with information on their respective worst-case filling and spilling behavior. It proceeds by first computing the minimum and maximum displacement of the call sites within the program using the input call graph (l. 2-3). Next, the worst-case stack occupancy at call sites is bounded (l. 8) using the previously computed minimum displacements (D_{min}) for each function separately. The ensure analysis (l. 6), which relies on the maximum displacements (D_{max}), is similarly performed for each function individually. Information on the minimum stack occupancy is propagated locally from call sites to the ensure instructions. Finally, the reserve analysis (l. 10) is performed on the call graph. It uses the occupancy bound (`BoundOccupancy`, l. 8) and propagates context-dependent information on the stack occupancy to the individual reserve instructions.

Computational Complexity Examining the computational complexity of the individual analysis phases, `AnalyzeReserves` is bounded by the number of possible contexts, i.e., the number of functions times the constant number of stack cache states. The data-flow analysis problems of `BoundOccupancy` and `AnalyzeEnsures` are similarly linear, but in the number of CFG nodes (due to the constant and typically low $|\mathcal{D}|$, the number of iterations until the analysis fixpoint is sub-polynomial). The remaining two functions are those that compute the minimum and maximum displacement. Their underlying path search problems are polynomial and NP respectively, when considered without user constraints. Also when solved through an ILP, the problems have shown to be efficiently solvable even for graphs larger than the ones encountered here (the number of functions in a program being naturally low). The shortest and acyclic longest path searches are quadratic and linear, respectively.

Algorithm 3 Main steps of the stack cache analysis for both, the ensure and reserve analysis problems.

Require: ACG ... The call graph of the program.
$CFGs$ ... The CFGs of all functions.
Ensure: Annotate `sens`- and `sres`-instructions with their worst-case filling and spilling behavior.

1: ▶ Minimum/maximum displacement at call sites.
2: $D_{min} =$ `ComputeMinimumDisplacement`(ACG)
3: $D_{max} =$ `ComputeMaximumDisplacement`(ACG)
4: **foreach** $G \in CFGs$ **do**
5: ▶ Ensure analysis.
6: `AnalyzeEnsures`(G, D_{max})
7: ▶ Bound worst-case occupancy at call sites.
8: $O_{bound} = O_{bound} \cup$ `BoundOccupancy`(G, D_{min})
9: ▶ Reserve analysis.
10: `AnalyzeReserves`(ACG, O_{bound})

SCA Graph Pruning During the SCA graph construction (Alg. 2), several contexts might be created for the same function whose spill costs evaluate to 0, but have different occupancy values. When analyzing the worst-case spilling behavior of a program, these contexts are equivalent and can be merged. Note that this merging could also be done during the construction at the expense of conservatively collapsing descendent contexts having different costs.

Furthermore, potentially infeasible contexts could be created, e.g., when the recursion depth of a recursive function is limited, which can be eliminated during a post-processing phase or during graph construction. The maximum displacement, for instance, can be used to prune parts of the graph already while processing it.

In addition to the lossless pruning opportunities mentioned above, the size of the graph can also be reduced by merging contexts and annotating the merged context with the larger occupancy and higher spill cost. This would, for instance, allow us to reduce the complexity of any subsequent analysis that takes spill cost into account. The advantage of this approach is that the degree of context merging can be decided on-demand, which allows to trade analysis precision against computational complexity.

4. WELL-FORMED PROGRAMS

The presented algorithms are based on the simple program model described in Section 2, which assumes a single `sres`-instructions at the entry of a function and a single `sfree`-instruction at the function exit. However, the approach is easy to extend to the larger class of *well-formed* programs. We define a well-formed program in terms of the paths through the program's functions:

DEFINITION 6. *A program is well-formed when all functions in the program are well-formed.*

DEFINITION 7. *A function with CFG $G = (N, E, r, e)$ is well-formed, if every path of nodes $p = (n_1, \ldots, n_m)$, where $\forall\, 0 < i < m\colon (n_i, n_{i+1}) \in E$, satisfies one of the conditions:*

- *No instruction $n_i \in p$ is an `sres`- or `sfree`-instruction.*
- *Two indices $0 < i_r < i_f \leq m$ exist, such that n_{i_r} is the first `sres`-instruction and n_{i_f} is the last `sfree`-instruction on the path, and the amount of space reserved by n_{i_r} is equal to the amount freed by n_{i_f}, and the path $p' = (n_{i_r+1}, \ldots, n_{i_f-1})$ is empty or well-formed.*

The above definition can also be extended to cover `sens`-instructions, which we omit here for brevity. Note that the definition of well-formed paths is based on *all* possible paths through the CFG, including potentially infeasible paths. Well-formed programs have the nice property that allows them to be analyzed using the previously described algorithms with only minor modifications:

LEMMA 1. (Proof omitted for space reasons.) *Given the CFG $G = (N, E, r, e)$ of a well-formed function and a node $n \in N$, the accumulated amount of space reserved on the stack cache locally within the function is identical at n for all paths from the root node r to n.*

The previous lemma allows us to adapt the definition of the annotated call graph (Def. 3) to weight the edges in the graph using the accumulated amount of stack space reserved locally within functions. The algorithms to compute the minimum and maximum displacement (Sec. 3.1) can then be used without modification. The data-flow analyses (Sec. 3.2 and 3.4) have to be adapted to account for the nesting of `sres`-instructions, but otherwise proceed as before. The SCA graph also needs to be adapted to capture reserve instructions within functions explicitly using *logical* contexts. This can easily be done as the propagation rules to construct logical contexts from reserves are the same as those for calls. It is even possible to model the effect of loops explicitly in the SCA graph by loop peeling, i.e., several logical contexts may be constructed for one reserve instruction.

5. WCET-TOOL INTEGRATION

The stack cache analysis presented in this work is intended to be used as part of a worst-case execution time (WCET) analysis tool. The standard technique to perform WCET analysis is IPET [9, 10], which expresses the potential execution flows of the program as an ILP. The ILP variables express the execution frequency of the basic blocks and control-flow edges in the program, while ILP constraints express the legal flow of execution. Maximizing a weighted sum of the ILP variables then gives the WCET of the program. The standard IPET approach can be extended to be context-sensitive, by duplicating the ILP variables for each context and adapting the ILP constraints accordingly.

The results of the previously described stack cache analysis, can be used to extended the traditional ILP formulation of the IPET approach. As ensure analysis is context-independent, the cost it computes can trivially be added to the weight of the respective code block in the ILP. This can also be done when the IPET problem is context-sensitive.

Integrating spill cost requires more care, as the spill cost analysis graph is fully context-sensitive with regard to the observable stack cache state. The contexts considered by the reserve analysis might thus differ from the contexts used by the IPET analysis. Two options can be considered to overcome this problem. The first option is to merge the SCA contexts (as described above) so they match the contexts of the IPET analysis. The merging might then cause some loss of precision. The other option is to account for the full context information encoded in the SCA graph by introducing additional ILP variables representing the nodes and edges of the SCA graph. ILP constraints then connect these ILP variables to those of the original IPET problem. For instance, in an IPET formulation without contexts, the execution frequency of the basic block containing an `sres`-instruction has to be equal to the sum of the ILP variables of the corresponding SCA contexts. Using this approach the full context information of the SCA graph can be encoded.

6. EVALUATION

We evaluated our approach using the LLVM-based[1] compiler framework of the Patmos processor [12], which comes with a stack cache and its associated control instructions [1]. Both, the compiler and the processor, are available as open-source.[2] Benchmarks of the MiBench benchmark suite [7] were compiled using aggressive optimizations (`-O3`) and subsequently analyzed using our technique, assuming stack cache sizes of 1024 (`1k`), 512 (`512b`), and 256 (`256b`) byte with 4 byte blocks. Note that the compiler automatically allocates

[1] `http://www.llvm.org/`

[2] `http://github.com/t-crest`

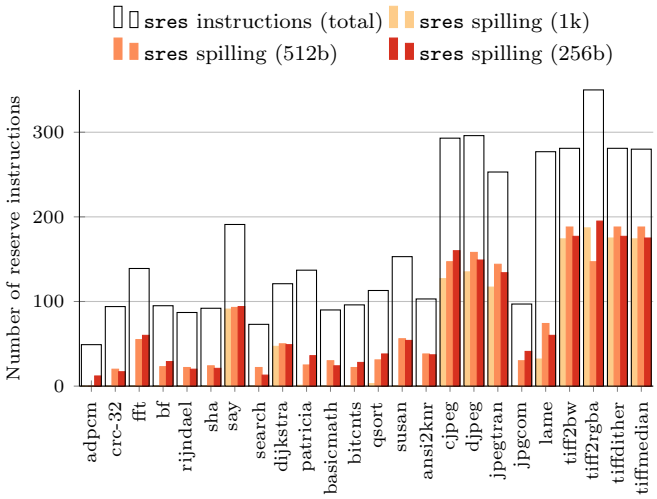


Figure 6: Total number of `sres`-instructions and number of `sres`-instructions potentially spilling in the worst-case (lower is better).

stack data on the stack cache [1]. The compiled benchmarks contain between 9550 and 74291 instructions, of which 0.3 - 0.5% are `sres`- and 0.1 - 5.5% are `sens`-instructions.

Figure 6 summarizes the result of the reserve analysis (Section 3.3). It shows the total number of `sres`-instructions in the benchmarks (white bar) as well as the number of potentially spilling instructions among theses (colored bars). For the `1k` configuration the analysis can prove that almost no spilling will occur at runtime as the benchmarks have a rather shallow nesting of function calls. Even for the smaller cache configurations, the analysis finds that only about 37% of `sres`-instructions cause spilling in the worst-case. The analysis results reflect the observed spilling behavior at runtime reported in previous work [1].

More detailed numbers are presented in Table 1. In total 728 ILPs are generated (column ILP) during the displacement analysis (Section 3.1), 94 of which are due to `djpeg`. Note that the shortest path search was also performed using ILP to disambiguate calls through function pointers.

The number of contexts computed by the reserve analysis (Section 3.3) before pruning grows linearly with the stack cache size (columns O_{1k}, O_{512b}, O_{256b}), while the growth is much smaller for the pruned graphs (columns P). The highest number of contexts is initially computed for the `1k` configuration, where up to 53487 distinct contexts (`tiffdither`) are computed. Most of these are irrelevant for the worst-case spilling and can thus be eliminated. The pruned graphs only retain 15% of the original contexts in the mean (ignoring empty graphs). For the `256b` configuration fewer contexts are computed, at most 8198 distinct contexts for `djpeg`. However, a larger number of contexts is retained during pruning (29%).

The ensure analysis (Section 3.4) finds that very few of the ensure instructions may cause filling in the worst-case (columns F_{1k}, F_{512b}, F_{256b}). Up to 99% (`adpcm`) of the ensures never perform any filling for the `256b` configuration (in the mean 79%, column $Ratio_{256b}$).

As most parts of the analysis are linear in the program size, the computational overhead of the entire analysis is low. Using an unoptimized executable, the average analysis time was 1.30s, 0.75s, and 0.46s for the `1k`, `512b`, and `256b` configurations respectively.

			SCA graph size (original/pruned)						Number of `sens`-instructions (filling/non-filling)						
Benchmark	Fun.	ILP	O_{1k}	P_{1k}	O_{512b}	P_{512b}	O_{256b}	P_{256b}	F_{1k}	NF_{1k}	F_{512b}	NF_{512b}	F_{256b}	NF_{256b}	$Ratio_{256b}$
adpcm	63	12	755	0	428	0	270	46	0	99	0	99	1	98	0.99
crc-32	103	2	1750	0	821	62	415	80	0	358	4	354	68	290	0.81
fft	159	14	5060	0	1751	258	761	207	0	842	10	832	182	660	0.78
bf	111	14	2574	0	1136	66	509	104	0	368	2	366	64	304	0.83
rijndael	95	2	2710	0	1051	119	352	61	0	358	7	351	78	280	0.78
sha	101	2	2655	0	1048	97	338	76	0	360	5	355	73	287	0.80
say	217	20	25170	2422	8945	1781	2218	700	17	1907	77	1847	271	1653	0.86
search	78	2	1521	0	581	75	231	29	0	311	6	305	67	244	0.78
dijkstra	138	16	8402	483	2868	410	653	197	2	586	8	580	167	421	0.72
patricia	157	16	3100	63	1565	133	706	159	2	629	7	624	164	467	0.74
basicmath	99	2	1691	0	780	92	363	82	0	839	18	821	85	754	0.90
bitcnts	117	16	2193	0	930	76	407	77	0	365	5	360	67	298	0.82
qsort	124	4	2037	14	1010	103	373	110	2	588	8	582	166	424	0.72
susan	173	14	8989	0	3455	411	964	311	0	760	10	750	101	659	0.87
ansi2knr	119	14	3478	0	1424	139	639	173	0	397	11	386	76	321	0.81
cjpeg	351	50	45599	17434	19615	9570	7086	4061	24	1669	487	1206	666	1027	0.61
djpeg	363	94	52677	14783	22725	8619	8198	3771	158	1388	421	1125	588	958	0.62
jpegtran	301	90	45664	17605	19168	9427	6597	3870	119	1306	438	987	614	811	0.57
jpgcom	113	14	5394	0	2196	168	730	211	0	387	7	380	72	315	0.81
lame	303	14	18134	237	7098	1196	2403	574	4	3898	70	3832	322	3580	0.92
tiff2bw	327	72	53482	16436	21748	9215	6771	3522	82	1449	269	1262	392	1139	0.74
tiff2rgba	402	72	53189	17542	21371	5806	6195	3111	129	2022	395	1756	518	1633	0.76
tiffdither	326	72	53487	17254	21753	9213	6776	3520	76	1457	263	1270	386	1147	0.75
tiffmedian	325	72	53065	17410	21079	8943	6055	2980	79	1407	267	1219	336	1150	0.77

Table 1: The number of functions, the number of ILP runs, the SCA graph size before and after pruning, as well as the number of ensure instructions that are potentially filling or are certain to not cause any filling.

7. RELATED WORK

Static analysis [14, 6] of caches typically proceeds in two phases: (1) potential addresses of memory accesses are determined, (2) the potential cache content for every program point is computed. The stack cache allows for a simpler analysis model that does not require the precise knowledge of addresses. This eliminates a source of complexity and imprecision. The hardware states of the stack cache can, furthermore, be summarized using the stack occupancy. The analysis is thus simplified drastically, allowing us to compute fully context-dependent cache states. This information can be encoded on-demand into the actual timing analysis. This is further supported by the observation that the stack cache serves up to 75% of the dynamic memory accesses [1].

Tidorum Ltd.'s WCET analysis tool Bound-T supports the analysis of over- and underflow of the register-window mechanism of the SPARC architecture [13, Section 2.2]. They compute bounds on the number of register windows that are pushed to/popped from the stack and use these bounds to classify the corresponding `save` and `restore` instructions as trapping or non-trapping. The analysis imposes several limitations on the program structure and operating system's trap handler. Additionally, it is unclear how calling contexts and recursion are handled.

Our approach to compute the stack cache displacement has some similarity to techniques used to statically analyze the maximum stack depth [5, 13]. Instead of only finding the maximum stack depth for the program's entry function , i.e., a longest path on a weighted call graph, the displacement information is required for every function. We duplicate the call graph, such that one copy represents the cost-free head and the second copy the weighted tail of the desired path.

8. CONCLUSION

We have shown how to efficiently analyze the worst-case behavior of a stack cache. Implemented in a real-time system, the stack cache benefits cache predictability and in combination with a tailor-made analysis has the potential to lower the WCET bound for real-time programs. We thoroughly investigated the behavior of *reserve* (spill) and *ensure* (fill) operations. Ensure analysis has proven to be a local problem and can be solved with minimal computational overhead. In general, our techniques combine intra-procedural data-flow analysis with longest (as well as shortest) path search on the inter-procedural call graph. By propagating bounds induced by stack cache size, we can avoid a costly context-sensitive data-flow analysis at program scope.

To find the maximum remaining stack depth starting from arbitrary points in the call graph, we augment the graph and enable path search on the weighted tail.

We also introduced the SCA graph, which fully models the context-sensitive spill cost of the stack cache and is suitable for integration in a worst-case timing analysis. Opportunities for pruning the graph with and without loss of analysis precision have been presented.

For future work, we intend to investigate the feasibility of integrating the SCA graph with a WCET tool without the loss of analysis precision.

Acknowledgment

This work was partially funded under the European Union's 7th Framework Programme under grant agreement no. 288008: Time-predictable Multi-Core Architecture for Embedded Systems (T-CREST). Alexander Jordan was supported by the Austrian Science Fund (FWF) under contract P21842.

9. REFERENCES

[1] S. Abbaspour, F. Brandner, and M. Schoeberl. A time-predictable stack cache. In *Proceedings of the Workshop on Software Technologies for Embedded and Ubiquitous Systems*, SEUS '13, 2013.

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

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

[4] C. Cullmann, C. Ferdinand, G. Gebhard, D. Grund, C. Maiza, J. Reineke, B. Triquet, S. Wegener, and R. Wilhelm. Predictability considerations in the design of multi-core embedded systems. *Ingénieurs de l'Automobile*, 807:36–42, September 2010.

[5] C. Ferdinand, R. Heckmann, and B. Franzen. Static memory and timing analysis of embedded systems code. In *Proceedings of Symposium on Verification and Validation of Software Systems*, VVSS '07, pages 153–163. Eindhoven University of Technology, 2007.

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

[7] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workshop on Workload Characterization*, 2001.

[8] U. Khedker, A. Sanyal, and B. Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, Inc., 1st edition, 2009.

[9] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the Design Automation Conference*, DAC '95, pages 456–461. ACM, 1995.

[10] P. P. Puschner and A. V. Schedl. Computing maximum task execution times - a graph-based approach. *Real-Time Systems*, 13(1):67–91, July 1997.

[11] M. Schoeberl, B. Huber, and W. Puffitsch. Data cache organization for accurate timing analysis. *Real-Time Systems*, 49(1):1–28, Jan. 2013.

[12] M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, C. W. Probst, S. Karlsson, and T. Thorn. Towards a time-predictable dual-issue microprocessor: The Patmos approach. In *Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, PPES '11, pages 11–20, 2011.

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

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