A Comparative Study of the Precision of Stack Cache Occupancy Analyses

Amine Naji
U2IS
ENSTA ParisTech
Université Paris-Saclay
amine.naji@ensta-paristech.fr

Florian Brandner
LTCI, CNRS
Telecom ParisTech
Université Paris-Saclay
florian.brandner@telecom-paristech.fr

ABSTRACT

Utilizing a stack cache in a real-time system can aid predictability by avoiding interference between accesses to regular data and stack data. While loads and stores are guaranteed cache hits, explicit operations are required to manage the stack cache. The (timing) behavior of these operations depends on the cache occupancy, which has to be bounded during timing analysis. The precision of the computed occupancy bounds naturally impacts the precision of the timing analysis. In this work, we compare the precision of stack cache occupancy bounds computed by two different approaches: (1) classical inter-procedural data-flow analysis and (2) a specialized stack cache analysis (SCA). Our evaluation, using MiBench benchmarks, shows that the SCA technique usually provides more precise occupancy bounds.

Categories and Subject Descriptors

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

Keywords

Program Analysis, Stack Cache, Real-Time Systems

1. INTRODUCTION

To meet the timing constraints in systems with hard deadlines, the worst-case execution time (WCET) of software needs to be bounded. Many features of modern processor architectures, such as caches, improve the average performance, but have an adverse effect on WCET analysis. Time-predictable computer architectures [7] thus propose alternative designs that are easier to analyze, particularly focusing on the memory hierarchy [5, 6]. One such design is the stack cache [1, 8], i.e., a cache for stack data complementing a regular data cache. This promises improved analysis precision, since unknown access addresses can no longer interfere with stack accesses (and vice versa). Secondly, the stack cache design is simple and thus easy to analyze [4].

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 and the compiler (programmer) is responsible to enforce that all stack data is present in the cache when needed using three stack control instructions: reserve (`sres`), free (`sfree`), and ensure (`sens`). The worst-case (timing) behavior of these instructions only depends on the worst-case spilling and filling of `sres` and `sens` respectively, which can be bounded by computing the maximum and minimum cache occupancy [4], i.e., the value of $\mathtt{MT} - \mathtt{ST}$.

Stack cache occupancy bounds, and the associated spill/fill costs can be computed using the recently proposed Stack Cache Analysis (SCA) [4]. The approach splits the analysis problem into several smaller steps, using context-insensitive data-flow analyses to capture function-local properties and longest/shortest path searches on the call graph to model calling contexts. An alternative solution would be to simply model the problem as a traditional inter-procedural data-flow analysis (iDFA) [2]. This appears simpler to implement, as the various steps of SCA are modeled in a single concise analysis. However, the impact on analysis precision has not been investigated so far. Indeed, overestimating the occupancy can increase the spill costs associated with `sres` instructions, while underestimating the occupancy can increase the fill costs of `sens` instructions. This work thus compares the precision of the two analysis approaches with respect to the attained max./min. occupancy bounds.

The paper is structured as follows: Section 2 provides some background related to the stack cache as well as static program analysis. We then present the two approaches to analyze the occupancy bounds for the stack cache. The analyses are evaluated in Section 4 before concluding.

2. BACKGROUND

The stack cache is implemented as a ring buffer with two pointers [1]: *stack top* (`ST`) and *memory top* (`MT`). The top of the stack is represented by `ST`, which points to the address of all stack data either stored in the cache or in main memory. `MT` points to the top element that is stored only in main memory. The stack grows towards lower addresses.

The difference $\mathtt{MT} - \mathtt{ST}$ (*occupancy*) represents the amount of occupied space in the stack cache, which cannot exceed the size of the cache's memory $|SC|$, thus $0 \leq \mathtt{MT}{-}\mathtt{ST} \leq |SC|$.

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

The stack control instructions manipulate the two stack pointers and initiate memory transfers to/from the cache to main memory, while preserving the equation from above. A brief summary is given below, details are available in [1]:

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

The compiler manages the stack frames of functions quite similar to other architectures with exception of the ensure instructions. For brevity, we assume a simplified placement of these instructions. Stack frames are allocated upon entering a function (`sres`) and freed immediately before returning (`sfree`). A function's stack frame might be (partially) evicted from the cache during calls. Ensure instructions (`sens`) are thus placed immediately after each call. We also restrict functions to only access their own stack frames.[2]

2.1 Data-Flow Analysis

Data-flow analysis (DFA) is used to gather information about a program without executing it. A DFA is 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. The join operator ($\sqcap$) and transfer function (T) are instantiated to form $\mathrm{IN}(i)$ and $\mathrm{OUT}(i)$ functions, which are associated with an instruction i and represent values over $\mathcal{D}$. The resulting (recursive) equations are finally solved by iteratively evaluating these functions until a fixed-point is reached [2].

Inter-procedural analyses additionally consider the call relations between functions. In this case, additional data-flow equations are constructed modeling function calls and returns [2]. Often these analyses are *context-sensitive*, i.e., the analyses distinguish between (bounded) chains of functions calls. Such a chain of nested function calls is then called a *call string*, which defines a calling context that can be distinguished from other parts of the program calling the same function. Call strings typically have a length limit. The longer the call strings, the higher the ability to distinguish different contexts. Consequently, the analysis results are more precise. Increasing the call string length may also increase the computation complexity and the required memory footprint since additional data-flow equations are created for each context. A call string length of zero corresponds to a context insensitive data-flow analysis.

3. CACHE OCCUPANCY ANALYSES

We present how to compute the cache's occupancy, which ca be used to bound timing, using an inter-procedural data-flow analysis (iDFA) and a tailored stack cache analysis.

3.1 Inter-procedural Data-flow Analysis

The domain of the iDFA approach are positive integer values in $\mathcal{D} = \{0, \ldots, |SC|\}$, where $|SC|$ represents the stack cache's size. Since both, the min. and the max. occupancy are needed, two analysis problems have to be defined. We will start with the max. occupancy. The analysis starts at the program entry, where the occupancy is assumed to be 0. It then propagates occupancy values along all execution paths, while considering the effect of the instructions along the path. Only the stack control instructions (see Section 2) can have an impact: (1) `sres` instructions increase occupancy by their argument k, (2) `sens` instructions make sure that the occupancy is larger than k, and (3) `sfree` instructions reduce the occupancy by k. The resulting data-flow equation for an instruction i are given below:

$$\mathrm{OUT}_{Occ}(i) = \begin{cases} \min(\mathrm{IN}_{Occ}(i) + k, |SC|) & \text{if } i = \texttt{sres } k \\ \max(\mathrm{IN}_{Occ}(i), k) & \text{if } i = \texttt{sens } k \\ \max(0, \mathrm{IN}_{Occ}(i) - k) & \text{if } i = \texttt{sfree } k \\ \mathrm{IN}_{Occ}(i) & \text{otherwise} \end{cases}$$

The occupancy right before an instruction (due to control-flow joins) is derived by taking the maximum occupancy from any of its predecessors (*Preds*), except for the program's entry. In the case of inter-procedurual analysis, predecessors can also be calls or returns from other functions:

$$\mathrm{IN}_{Occ}(i) = \begin{cases} 0 & \text{if } i = \texttt{entry}, \\ \max_{s \in Preds(i)}(\mathrm{OUT}_{Occ}(s)) & \text{otherwise} \end{cases}$$

The data-flow equations to compute the min. occupancy are very similar. Only the max operator of the $\mathrm{IN}_{Occ}(i)$ equation needs to be replaced by the min operator. Context sensitivity can easily be ensured by adding context information to the data-flow equations of the respective instructions.

This model is also implemented and validated in Absint's aiT timing analyzer tool [10].

3.2 Stack Cache Analysis

The stack cache analysis (SCA) [4] relies on similar DFA analyses. However, instead of a single, large inter-procedural DFA, several smaller function-local analyses are used. The impact of other functions at function calls in these DFAs are modeled through minimum and maximum *displacement* values, which represent the min./max. amount of data potentially evicted from the the stack cache during a function call. Displacement values are computed by performing shortest/longest path search on program's call graph whose weights represent the reserved stack space k. Complex context-sensitive analysis thus can be avoided.

The analysis is based on the observation that the occupancy at any instruction within a function can be computed from the occupancy at the function's entry and the displacement of all the potential function calls on any path leading to the particular instruction. The min. occupancy thus can be computed by considering the initial min. occupancy and the max. displacement. Likewise, the min. displacement allows to derive the max. occupancy.

The program is thus analyzed in several steps. First, the min. and max. displacement of each function is computed using longest/shortest path searches on a weighted call graph. Next, local DFAs are performed to compute local lower and local upper bounds on the min. and max. occupancy within each function assuming a stack cache that is full at function entry. Finally, the concrete occupancy bounds are computed for each function considering the occupancy bounds at its respective callers and the previously

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

computed local occupancy bounds. The final phase can deliver fully context-sensitive information, if so desired.

This analysis was implemented and validated against runtime measurements in previous work [8].

4. EXPERIMENTS

We evaluated both approaches using the LLVM-based compiler framework of the Patmos processor [9], which comes with a stack cache and its associated control instructions. Benchmarks of the MiBench benchmark suite [3] were compiled using optimizations (`-O2`) and subsequently analyzed using both techniques, assuming a stack cache size of 256 byte, 4 byte cache blocks, and a contexts string length of 0. Figure 1 shows the percentage of functions where the occupancy bound at function entry of SCA was either greater, equal, or smaller than that computed by iDFA.

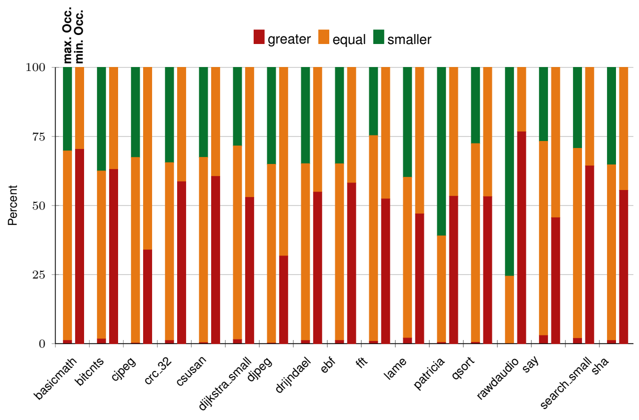

Figure 1: Percentage of occupancy bounds (max./min) by SCA being (1) greater, (2) equal, or (3) smaller than iDFA.

When considering max. occupancy, SCA is less precise when the delivered bound is greater, i.e., the lower portion of the first bar of each benchmark should be as small as possible. Indeed, these cases are rare ($< 3\%$ over all benchmarks), while SCA is often more precise (34% on average).

The situation is inverse when considering min. occupancy. Here, SCA is less precise when the delivered bounds is smaller. This would be represented by the upper portion of the second bar. However, this appears for one function of the three benchmarks `tiff2bw`, `tiffdither`, and `tiffmedian` respectively. SCA is usually even more precise (52% on average).

We repeated these experiments for iDFA with other call string lengths (1, 2, 3, 10 and 20). However, we only observed minor improvements for max. occupancy and almost no change for min. occupancy. The `bitcnts` benchmark, for instance, has a maximum call depth of 20, ignoring recursive functions, and still does not show relevant improvements with call strings of length 20 due to the impact of recursion elsewhere as explained later.

Overall, SCA is almost always as precise or even more precise than iDFA. The results are similar, albeit less pronounced, with longer context string lengths.

4.1 Discussion

A closer look reveals that the imprecision of iDFA is mostly due to chains of function calls, whose lengths exceed the analysis' context string length (e.g., due to recursion). Let us first examine such situations for max. occupancy.

The problem of iDFA with long call chains is that calling contexts are no longer distinguished, i.e., all information is merged in a single calling context. The occupancy information computed for these regions is, as expected, rather pessimistic, leading to considerable overestimation of the max. occupancy. Even worse, the overly conservative occupancy level is propagated out of these merged calling contexts along control-flow edges of function returns. Recall that the meet operator for this analysis is the max operator. This means that the conservative max. occupancy bounds are even further propagated, way beyond the merged calling contexts that initially caused the imprecision. This particularly applies to recursive functions.

Example 1. Figure 2 shows an example illustrating this situation. Assume that function `A` consists of one basic block and that function `B` is called before function `D`. Since `B` and `C` recursively call each other, their respective max. occupancy grows until they reach the stack cache size during the fixed-point computation of iDFA (unless unbounded call strings are used). The transfer functions for the return instructions then propagate the maximum to their respective callers, which leads to a max. occupancy that is close to the stack cache size right after the function call to `C` within `B` (and vice verse). A similarly high occupancy is propagated out of the recursion to the instruction succeeding the call from `A` to `B`. The high occupancy might actually occur within the recursion. However, the actual occupancy at this point is much lower. The overestimation is further propagated to function `D`. Resulting in overly conservative analysis results there, even when the context string length is not exceeded.

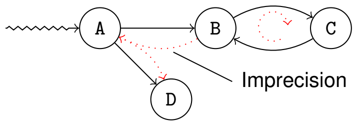

Figure 2: Imprecision propagated out of recursive functions when computing max. occupancy with iDFA.

Patmos' newlib C library contains (potentially) recursive functions in the start-up code of each program. iDFA thus assumes that the stack cache is filled up entirely before even reaching the program's `main` function. Since the computed max. occupancy at `main` is considerably overestimated, imprecision is propagated throughout large portions of the considered benchmarks. An important observation here is that increasing the call string length will not help fixing this problem, as the precision limit will be reached before the end of the recursion (unless infinite call strings are used). The SCA approach does not face this problem. Instead of relying on the occupancy propagated outwards by the recursive functions, it simply relies on their displacement values. A possible fix for this problem for iDFA would be to memorize the occupancy level before each call. The occupancy propagated backwards from a return then always has to be smaller than the memorized value. However, the potentially large displacement of the called function is ignored, which may still lead to considerable overestimation.

A similar problem arises for non-recursive programs with deep call chains containing two subsequent function calls that eventually invoke the same function. iDFA then behaves similar to recursive programs as shown in Figure 3.

Example 2. Assume that, in this example, the function call from `B` to `C` appears before the call from `B` to `D`. Then, iDFA initially propagates an accurate occupancy level through the calls from A to B and finally to C. At first, even the occupancy at D is computed correctly. However, due to the deep call chain leading up to C, both calling contexts for D (originating from B or D) are merged. Due to the intermittent execution of D the occupancy is higher for this call chain. This increases the max. occupancy of C. The increase is subsequently propagated out of C to both of its callers. This incidentally increases the occupancy after the call to C within B. Which then again increases the occupancy at the following call to D. This leads to a feedback loop similar to that seen for recursive functions in the previous example.

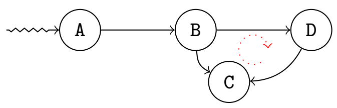

Figure 3: Feedback loop enforcing imprecision of non-recursive functions for iDFA computing max. occupancy.

Still, iDFA can be more precise than SCA (as shown by or results). This is explained by an underestimation of the min. displacement. As mentioned before, the min. displacement is obtained by performing a shortest path search on the program's call graph. The path here represents nested function calls and its length the minimal amount of stack space required in the stack cache by the functions stack frames respectively. Now, consider a case where two leaf functions[3] are called within a single basic block, i.e., when one function is called the other function is called too. In this case, the minimal path search will chose the function with the smaller stack frame to compute the min. displacement. However, since both functions are called, the actual min. displacement is determined by the larger stack frame. This situation can, of course, also appear in more general forms. The imprecise min. displacement ultimately leads to an underestimation of the max. occupancy observed in our experiments. However, this appears to be of minor importance in practice.

For min. occupancy iDFA appears to be even more imprecise. For one, this is explained by the fact that the max. displacement (in contrast to the min. displacement) can be computed precisely. SCA's min. occupancy thus does not suffer from inherent imprecision. In addition, iDFA spreads imprecision as before in the presence of deep call chains. This may even lead to feedback loops in non-recursive programs as described before. Two observations are particularly interesting at this point. While the iDFA approach is amenable to improvements by memorizing the max. occupancy before calls, such a fix appears to be impossible here. The problem is that a lower bound cannot be established as easily for function calls when the min. occupancy is computed. Secondly, it appears that the precision could be improved using very long call strings (ignoring cases incurring recursion). This, however, leads to a paradox situation: the precise computation of the min. occupancy would then require high levels of context sensitivity in order to compute the worst-case filling at `sens` instructions. The filling, however, only depends on the nesting of functions called right before the ensure and thus is by its nature context-insensitive. The SCA approach exploits precisely this property, and evidently achieves excellent results.

[3]Leaf functions do not call any other function.

5. CONCLUSION

We compared the precision of stack cache occupancy bounds computed by two different approaches. On the one hand, the iDFA approach, which models the problem as a traditional inter-procedural data-flow analysis. On the other hand, the SCA approach that splits the analysis problem into several smaller steps, using context-insensitive data-flow analyses along with longest/shortest path searches on the call graph. Our experiments revealed that iDFA suffers from imprecision in nearly all benchmarks of the MiBench benchmark suite. The lack of precision is due to chains of function calls, whose lengths exceed the analysis' context string length (e.g., due to recursion). As a future work, we plan to compare the efficiency (i.e computation complexity and required memory footprint) of iDFA and SCA.

Acknowledgments

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

6. REFERENCES

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

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

[3] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. of the Workshop on Workload Characterization*, WWC '01, 2001.

[4] A. Jordan, F. Brandner, and M. Schoeberl. Static analysis of worst-case stack cache behavior. In *Proc. of the Conf. on Real-Time Networks and Systems*, pages 55–64. ACM, 2013.

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

[6] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee. PRET DRAM controller: Bank privatization for predictability and temporal isolation. In *Proc. of the Conference on Hardware/Software Codesign and System Synthesis*, pages 99–108. ACM, 2011.

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

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

[9] M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, C. Probst, S. Karlsson, and T. Thorn. *Towards a Time-predictable Dual-Issue Microprocessor: The Patmos Approach*, volume 18, pages 11–21. OASICS, 2011.

[10] T-CREST. Report on architecture evaluation and WCET analysis. Technical report, 2013.