WCET-Aware Stack Frame Management of Embedded Systems using Scratchpad Memories

Yooseong Kim[1], Mohammad Khayatian[2], Aviral Shrivastava[3]
[1]Synopsys, Inc., [2,3]Arizona State University
Corresponding author[2]: mkhayati@asu.edu

***Abstract*—Scratchpad memories (SPMs) provide a time-predictable alternative to caches but requires explicit management in the code. When the call stack is stored in the SPM, stack frames need to be evicted to and restored from the main memory to avoid stack overflow. We propose a technique to find optimal locations in the code to perform stack management operations such that the worst-case execution time (WCET) is minimized.**

I. Introduction

In embedded systems with hard real-time constraints, task must finish execution before their deadlines to ensure correct system behavior [1]. It is, therefore, important to determine the worst-case execution time (WCET) of a task. Scratchpad memories (SPMs) are promising alternatives to caches in such systems, because caches often make WCET analysis complicated and pessimistic [2]. SPMs are raw onchip SRAMs explicitly controlled by executing direct memory access (DMA) instructions at the runtime. This explicit control provides the much needed time predictability and facilitates static analyses to obtain a safe WCET estimates.

Since SPMs are typically only a few megabytes in most embedded processors, programs running on an SPM-based processor may face stack overflow issue. Most previous stack data management techniques either divide the stack data into two parts (one to be stored in the main memory and the other in the SPM) and accessed by using two stack pointers, or allocate the SPM space for only selected stack variables. In contrast, we keep once stack in the SPM. The stack frames are evicted to main memory temporarily before calling a function and then brought back later before it returns. These management operations happen only at selected call sites, and we select the optimal set of call sites to perform such operations to minimize the WCET. Please refer to [2] for more details.

II. Our Approach

Our stack management works in two steps. First, we select the optimal set of call sites to perform stack management, using the ILP formulation. Then, we perform code transformation as shown in Fig. 1.a at selected call sites. All stack frames stored in the SPM at the moment are evicted before the call and restored after the return. Fig. 1.b illustrates that the evicted stack frames are stored in a virtual stack in the main memory. The depth of the virtual stack in the main memory at a program location depends on its execution context (function call history). An execution context can be represented as a unique path on a call graph. We use a special control flow graph (CFG) that explicitly exposes every possible execution path in a program, called *inlined CFG* [2], [3] to calculate the virtual stack depths at compile-time. Let $G = (V, E, v_s, v_t, F, fn)$ be an inlined CFG. V is the set of basic blocks. The set of edges is defined as E. v_s and v_t represent the starting and terminal basic blocks respectively. F is the set of functions in the program, and mapping $fn : V \rightarrow F$ states that $fn(v)$ is the function that v belongs to. We take an inlined CFG G, the SPM size, stack frame sizes, and the loop bounds as input[1]. W_v is the WCET of the code starting from the basic block v to the final basic block. The objective is to minimize the WCET of the whole program.

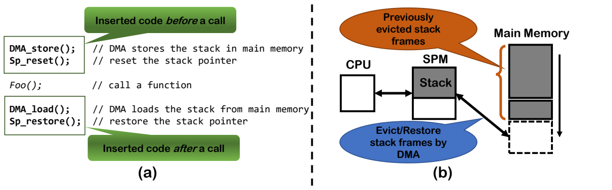
$$minimize \qquad W_{v_s} \tag{1}$$

Let n_v be the execution frequency of the basic block v, and t_v be the time it takes to execute instructions in v once in the worst case. Then, v contributes to the WCET with the sum of its computation cost ($n_v t_v$) and any management cost C_v (defined later). For W_v to be an upper bound of the WCET starting from v, W_v has to be greater than or equal to the sum of the contributions of v and its successor w as follows.

$$\begin{aligned} \forall(v,w) \in E, \qquad & W_v \geq W_w + n_v \cdot (t_v + C_v) \\ & W_{v_t} = n_{v_t} \cdot (t_{v_t} + C_{v_t}) \end{aligned} \tag{2}$$

Let $\mathcal{C} \subset V$ be the set of all basic blocks containing a function call. Also, let $\mathcal{R} \subset V$ be the set of all basic blocks that immediately follow a return, which is the first caller basic blocks after returning from the callee.

A basic block $v \in C$ has a management cost only if a management code block is inserted at v, which is denoted by a binary decision variable M_v. The management cost at $v \in R$ depends on the management operation before its



```
DMA_store();    // DMA stores the stack in main memory
Sp_reset();     // reset the stack pointer

Foo();          // call a function

DMA_load();     // DMA loads the stack from main memory
Sp_restore();   // restore the stack pointer
```

Inserted code *before* a call
Inserted code *after* a call
(a)
Previously evicted stack frames
Main Memory
CPU
SPM
Stack
Evict/Restore stack frames by DMA
(b)

Fig. 1. a) Code modification to perform management operations, which b) move stack frames between SPM and main memory using DMA transfers.

[1] We remove all back edges, assuming that G is reducible to make G acyclic

2380-6923/19/$31.00 ©2019 IEEE
DOI 10.1109/VLSID.2019.00127

corresponding function call ($M_{cl(v)}$) because any evicted stack frame must be restored before returning to the caller function.

$$\forall v \in \mathcal{C}, \qquad C_v = \begin{cases} mo_c(S_v) & \text{if } M_v = 1 \\ 0 & \text{if } M_v = 0 \end{cases} \tag{3}$$

$$\forall v \in \mathcal{R}, \qquad C_v = \begin{cases} mo_r(S_v) & \text{if } M_{cl(v)} = 1 \\ 0 & \text{if } M_{cl(v)} = 0 \end{cases} \tag{4}$$

where S_v is the variable to calculate the stack size at v (defined later), and $mo_c(x)$ and $mo_r(x)$ are the management overhead to evict and restore the stack, respectively, when the size of the stack is x bytes. These are linear equations that evaluate the sum of the time to execute the additional code for management, a constant DMA setup time, and a transfer time proportionate to x. Please refer to [2] for how we linearize the `if-then-else` conditions between variables.

The stack size at basic block v depends on i) the stack size at the parent function who called $fn(v)$ and ii) whether the stack had been evicted before the call to $fn(v)$. If the stack was evicted before $fn(v)$ was called, the stack at v will only have the stack frame of $fn(v)$. Let $p(v)$ denote the basic block in which a call to $fn(v)$ appears. The starting function ($fn(v_s)$), e.g. `main`, has no parent function. In the following, $V_{\texttt{main}}$ denotes the set of all basic blocks in `main`.

$$\forall v \in V_{\texttt{main}}, \quad S_v = sz_{fn(v_s)} \tag{5}$$

$$\forall v \notin V_{\texttt{main}}, \quad S_v = \begin{cases} sz_{fn(v)} & \text{if } M_{p(v)} = 1 \\ S_{p(v)} + sz_{fn(v)} & \text{if } M_{p(v)} = 0 \end{cases} \tag{6}$$

$$\forall v \in V, \quad S_v \leq \mathit{SPMSIZE} \tag{7}$$

where sz_f denote the stack frame size of function f and $\mathit{SPMSIZE}$ is the size of the SPM. Eq. (6) says that stack size at node v (S_v) is equal to i) the stack frame size of function $fn(v)$ if the stack of caller was evicted before (management operations are inserted) or ii) the summation of previous stack size and $fn(v)$'s frame size if the caller's stack remains in the SPM. Eq. (7) ensures that the stack size is always smaller than the size of the SPM. The solution for the above ILP is an optimal set of call sites, represented by M_v variables.

III. Evaluation

We compare our approach with two previous approaches: SSDM (Smart Stack Data Management) heuristic by Lu *et al.* [4] and a WCET-optimizing stack frame allocation technique by Liu and Zhang [5]. Our benchmarks are from Mälardalen WCET suite, compiled for ARMv4 ISA. As Lu *et al.*'s approach assumes cache access also, we use standard cache analysis technique [6] to predict cache miss/hit in the worst case, assuming 2-way set associative caches with LRU replacement policy. All memory accesses other than stack data accesses are assumed to take zero latency.

Please refer to [2] for detailed architectural parameters and assumptions on memory access latency. Fig. 2 shows the reduction in WCETs (calculated by static analysis) and in the number of cycles spent in memory access out of the WCETs. We use two memory configurations, (1) $min + 0.5 \times (max - min)$ and (2) $min + 0.7 \times (max - min)$, where min is the largest stack frame size and max is the maximum stack depth.

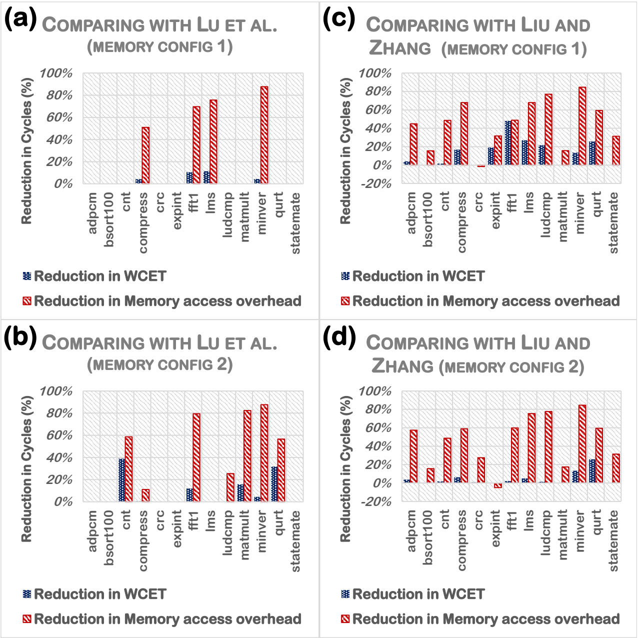


Fig. 2. WCET and memory access overhead reduction

Fig. 2.a and b show that SSDM finds exactly or almost the same solution as our technique for benchmarks with simple call patterns without nested calls, e.g. `bsort100`, `expint`, and `statemate`. Often times, however, SSDM suffers from its greedy characteristics and gets stuck in local optima. In `cnt`, `matmult`, and `qurt`, SSDM leads to significantly longer WCETs for the larger SPM size. This is because quite counter-intuitively, the larger memory made the SSDM heuristic method to get stuck into a local optimum. Fig. 2.c and d show that the technique by Liu and Zhang [5] has a very little overhead when most stack frames can be entirely allocated in the SPM (e.g. `bsort100`, `crc`, `expint`, and `matmult`) but greatly suffers from the long latencies of offchip main memory accesses in most cases. Results show that our approach can greatly reduce the memory access overhead in most benchmarks, which result in the WCET reduction of up to 48%.

References

[1] A. Shrivastava *et al.*, "Time in Cyber-physical Systems," in *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/-Software Codesign and System Synthesis*. ACM, 2016, p. 4.

[2] Y. Kim, "WCET-Aware Scratchpad Memory Management for Hard Real-Time Systems," Ph.D. dissertation, Arizona State University, 2017.

[3] Y. Kim *et al.*, "WCET-aware Dynamic Code Management on Scratchpads for Software-Managed Multicores," in *RTAS*. IEEE, 2014, pp. 179–188.

[4] J. Lu, K. Bai, and A. Shrivastava, "SSDM: Smart Stack Data Management for Software Managed Multicores (SMMs)," in *Proceedings of the 50th Annual Design Automation Conference*, 2013, pp. 149:1–149:8.

[5] Y. Liu and W. Zhang, "Scratchpad Memory Architectures and Allocation Algorithms for Hard Real-Time Multicore Processors," *Journal of Computing Science and Engineering*, pp. 51–72, June 2015.

[6] C. Cullmann, "Cache Persistence Analysis: Theory and Practice," *TECS*, vol. 12, no. 1s, p. 40, 2013.