

# Using Scratchpad Memory for Stack Data in Hard Real-Time Embedded Systems

Sungjun Kim

Department of Computer Science

Columbia University

New York, New York 10027

Email: skim@cs.columbia.edu

**Abstract**—This paper proposes a scratchpad memory allocation technique for stack data to achieve predictability for hard real-time embedded systems. First, we suggest a profiling-based source code modification technique to make the whole stack area fit in the given scratchpad memory size. Furthermore, even though the whole stack area cannot fit in the scratchpad memory size, we propose another scratchpad memory allocation to achieve predictability when an end-to-end measurement-based WCET analysis is used. Lastly, we compared performance of our design with data caches.

**Keywords**—hard real-time systems; embedded systems; scratchpad memory; predictability;

## I. INTRODUCTION

Many embedded systems are hard real-time systems, which require all the tasks meet their deadlines. The upper bound of each task should be known to guarantee the requirements, which are often verified by worst-case execution time (WCET) analysis. The WCET analysis should provide safe but tight bounds of the tasks, but unpredictable timing behavior of underlying hardware requires large amount of pessimism to derive safe bounds, making the analysis result infeasible. Thus, designing the underlying hardware architecture predictable is significant to achieve feasible results.

Among the underlying hardware components, the predictability of cache depends on the WCET analysis technique. With static WCET analysis, using caches has been problematic. Since cache states are transparent to software, static analysis of its behaviors is hard. Furthermore, greedy approach to cache state to obtain WCET estimates doesn't lead to safe bounds. Cache-hit of an instruction fetch or data load/store may lead to longer execution time of later instructions (a.k.a timing anomalies) [1]. Timing anomalies of cache require large search space in static WCET analysis.

On the other hand, with measurement-based WCET analysis, the upper bounds of cache misses are found in [2] (for LRU, FIFO, and Pseudo-LRU). Especially, when the measurement is performed on end-to-end basis (from the beginning of the program to the end of it) like [3], LRU cache becomes predictable: the worst-case cache misses are close to the measured misses.

Still, since the LRU cache is expensive to implement, scratchpad memory is often desired as a design alternative.

On average, the scratchpad memory occupies less area by 34%, and consumes less power by 40% than cache [4].

Furthermore, when a data area only requires small memory space, using the LRU cache is overkill: just using scratchpad memory for the data area is enough. Stack areas in embedded programs are one example. This is because embedded programs often limit their stack depths into small sizes. (For e.g., uC/OS-II limits stack depths of its tasks in build-time. [5]) Furthermore, even though huge stack depths are used, we propose a method to reduce the depths by moving huge stack variables into different data areas. Provided the maximum stack depths of programs become smaller than the scratchpad memory size after moving the variables, the scratchpad memory can be allocated for the whole stack.

Still, even though the depths are not reduced enough, we introduce another scratchpad memory management technique for the stack area to achieve predictability and reasonable performance.

Our main contributions are as follows:

- We propose a strategy of the scratchpad memory management for the whole stack area to achieve predictability when certain conditions are met.
- When the above strategy cannot be used, we also propose another technique of the scratchpad memory management for the stack to still achieve predictability and reasonable performance.
- While using several embedded systems benchmarks, we evaluate the effectiveness of our approaches and compare its performance with caches'.

The next chapter reviews related works while the chapter III and the chapter IV present our technique and its qualitative analysis, respectively. The chapter V illustrates the quantitative evaluation result of our approach. Finally, the conclusion is explained in the chapter VI.

## II. RELATED WORK

An idea of using cache for the stack to achieve predictability is presented in [6]. Using FIFO replacement policy with direct-mapped scheme is proposed to handle the stack data. While the stack pointers are moving, the stack cache fills and evicts the stack data. The stack cache will work predictably on stack machines, but not on register machines.

This is because the stack machines can access the stack data only with push and pop operations, thus the stack pointers can designate the data to be cached. However, the register machines can reference anywhere in the activation records at once, thus the stack pointers cannot tell which data will be referenced. As another time-predictable cache, locked cache can be used to achieve predictability as proposed in [7]. However, unlike our approach, the locked cache needs additional instructions for locking and unlocking to the original source code.

As an alternative for caches, several static scratchpad memory allocations have been researched. [8] presented an approach to map the data allocation problem into 0-1 knapsack problem. This approach showed the optimal performance in the given modeling. [9] presented a scratchpad memory allocation when the scratchpad memory size is unknown in compile time. In [9], the program is divided into several regions and the data of regions are allocated to achieve portability and to improve average-case performance. Unlike the ACET-driven approaches, [10] introduced algorithms to reduce WCET. The allocations are done by integer linear-programming, greedy approach, and branch-and-bound algorithm. The above solutions are based on static allocation, and explicitly tell which variables in source codes are allocated in the scratchpad memory. On the other hand, our approach is combined with static and dynamic allocations.

Apart from static scratchpad memory allocation, there have been several dynamic scratchpad memory allocation of data. [11] proposes the scratchpad memory management unit (SMMU) to dynamically manage the scratchpad memory with its dedicated operations. Data sets are explicitly moved with the operations. Later, [12] and [13] extends the SMMU technique to achieve time-predictability and reduce WCET. In our approach, however, data sets are not explicitly moved with special operations, thus no additional instructions are needed to the original source codes.

### III. DESIGN

The purpose of our approach is, with a given scratchpad memory size, using the memory for stack while obtaining predictability and high performance. Our method of using scratchpad memory comprises of a run-time part (hardware management) and a compile-time part (software management).

#### A. Scratchpad Memory Management in Run-Time

When the maximum stack depth used by the tested program is smaller than the given scratchpad memory size, the scratchpad memory can be used for the whole stack area. In this case, just allocating the scratchpad memory for the stack is enough.

However, the scratchpad memory size is not enough, the memory is divided into several same-sized blocks. The

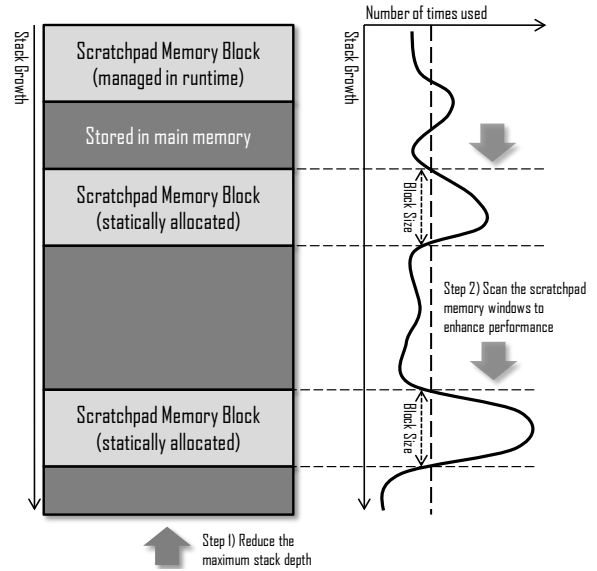


Figure 1. Scratchpad memory allocation and its management when the scratchpad memory size is smaller than the maximum stack depth used

mechanism is illustrated in Figure 1. Only the addresses of one block is dynamically allocated. This block works similar to a single-line, write-back cache to improve performance by exploiting locality. In this block, the read miss penalty is huge while the write miss penalty can be regarded as none, compared to cache hit. In both read or write misses, the contents of the block are saved to write buffer of main memory controller. However, in read misses, processors stall until the block is filled with the requested data, but don't stall and immediately write data to the block in the write misses because the previous contents of the block are already saved to the write buffer. (We assume the write buffer size is huge enough not to cause buffer overflow.)

Other than the dynamically allocated block, the addresses of all the other blocks are statically allocated and never change once allocated. Furthermore, the address windows of the blocks are referred simultaneously, thus no extra time is spent to access the scratchpad memory. More the memory is divided, more hardware implementation cost is required to refer the block addresses at the same time, however.

Compared to cache, the implementation cost of our design is lower because the replacement policy is simpler. While cache needs a data structure to record usages of cache blocks for its replacement policy, our design doesn't need such a data structure because only one block is replaced whenever a miss occurs.

Furthermore, provided a WCET analysis based on end-to-end measurements such as [3] is applied, this design provides predictability. After an end-to-end measurement, the number of the worst-case scratchpad block miss is just

one more than the measured value. This is because only one block is dynamically replaced and different initial states of the block can cause only one more miss for the first access, but not from the next accesses.

### B. Scratchpad Memory Allocation in Compile-Time

Whether the scratchpad memory will be used for the whole stack or will be allocated dynamically and statically is decided in compile-time. Furthermore, when dynamic and static allocation is used, the statically allocated addresses should be carefully selected to improve performance.

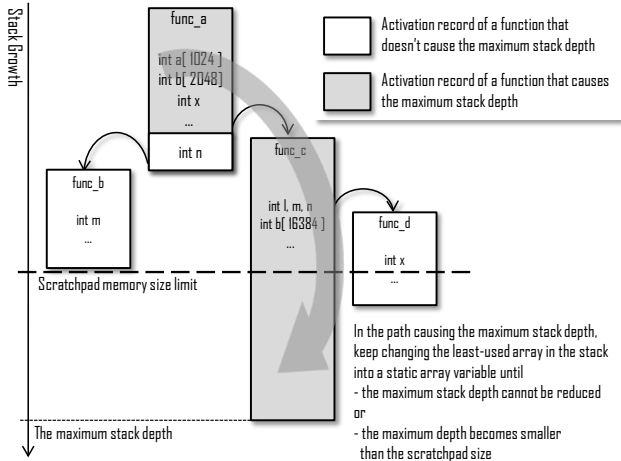


Figure 2. The greedy algorithm to reduce the maximum stack depth with profiling: dynamic call tree representation

1) *Source Code Modification – Reduce the Maximum Stack Depth:* Above all, provided the scratchpad memory size covers the maximum stack depth of the tested program, all the stack data will use the scratchpad memory. The timings of the stack data accesses becomes predictable. Therefore, the first step of our approach is reducing the maximum stack depth to fit it in the scratchpad memory. As usual suspects, huge stack variables such as *arrays* or *user – defined data structures* (e.g., *struct*) can be declared as *static variables* to reduce the maximum stack depth.

The algorithm to reduce the maximum stack depth is illustrated in Figure 2. At first, while profiling the tested program, the maximum stack depth is measured by probing the maximum stack pointer value. Then, provided the measured maximum stack depth is bigger than the scratchpad memory size, the stack depth needs to be reduced. We propose a greedy algorithm to reduce the maximum stack depth: from the function reaching the maximum stack depth to the root function in the tested program’s dynamic call tree, the least used array or user-defined data structure variable is redeclared as a static variable. Then, the program is profiled again to check the reduced maximum stack depth.

Provided the maximum stack depth becomes smaller than the scratchpad memory size, the stack depth reduction is no more needed. Otherwise, keep reducing the stack depth until no more reductions are possible or the maximum depth becomes smaller than the scratchpad memory size.

One limit of this technique is that it cannot be applied to *recursive functions*. This is because, after the stack variables are moved to the global variables, they cannot be privately used any more in recursively called functions. For those functions, the huge stack variables can be declared as *heap variables* instead. Using heap variables may degrade performance, however. The other limit is that the programs shouldn’t allow more than one entry point to its functions, thus statements such as *goto* are not allowed. Since the entry points designated by *goto* statements may need the array variables to be in the stack, moving them may destroy the program’s correctness.

When the maximum stack depth is reduced enough to fit in the scratchpad memory size, just allocating the whole stack area in the scratchpad memory is enough. The possible remaining scratchpad memory space can be used to allocate the other data areas as [8] or [10] do. Provided the maximum stack depth is not reduced enough yet, we propose a different technique, which is explained in the next chapter.

2) *Static Scratchpad Memory Allocation – Improve Performance:* When the memory is divided into several same-sized blocks, one is allocated dynamically while the others statically. The static blocks are allocated to the most frequently used stack areas, information of which are gathered by the profiler.

Figure 1 explains the algorithm to find those windows. The blocks are allocated from the most frequently used areas. While scanning the stack area by the window of the block size, find the most frequently accessed window and allocate a scratchpad memory block to the window. Repeat this procedure until all the static blocks are allocated.

## IV. QUALITATIVE ANALYSIS OF OUR APPROACH

### A. Predictability

Our approach provides predictability for end-to-end measurement based WCET analyses. First of all, provided the maximum stack depth is smaller than the given scratchpad memory size, our approach will guarantee that the scratchpad memory will be used for the whole stack.

Furthermore, then the scratchpad memory is used similar to a single-line cache, the upper bounds of the miss rates becomes tight. The worst-case misses are just one more than the measured values.

Similar to our design, LRU caches behave predictably when end-to-end measurement-based WCET analyses are used [2]. According to [2], their worst-case misses are increased only by a little more than the measured value. On the other hand, the upper bounds of miss rates of PLRU (Pseudo-LRU) caches are unbounded while those of

FIFO caches are proportional to their set-associativities. (For e.g., suppose 100 misses are measured in using 8-way set-associative FIFO cache. Then, the worst-case cache misses are a little more than 800.) Compared to LRU caches, however, the implementation cost of our design is lower because our technique is similar to a single-line cache.

### B. Performance Enhancement

Our approach will improve average-case performance as well. When the scratchpad memory is used for the whole stack area, only the fast on-chip memory will be accessed for the area. Furthermore, memory performance is still enhanced even when the scratchpad memory is allocated dynamically and statically. The management of the dynamic block is similar to a single-line cache, thus the performance enhancement will be similar to it. Furthermore, the static blocks are allocated to improve performance.

### C. Trade-offs Due to the Maximum Stack Depth Reduction

Since our main goal is achieving predictability, using the scratchpad memory for the whole stack area is the most desired design. However, one of the trade-offs of the stack depth reduction technique is that the total memory footprint can be increased. In other words, since lifetimes of stack variables can be mutually exclusive, using them as stack variables may require less total memory than using them as static variables. Furthermore, another trade-off is that the access to stack data can be less frequent. The performance enhancement due to our technique can become less beneficial.

## V. EXPERIMENTAL RESULT

### A. The Experiment Setup and Methodology

As our profiler, a single-core ARM processor simulator, *Simit – ARM v.2.1*, was used [14]. Furthermore, we combined a cache simulator, *Dinero IV* [15], with the ARM simulator to compare our technique with cache.

For the hard real-time embedded benchmarks, we used *MiBench v.1.0* [16] and *autopilot* program of *PapaBench* [17]. C programming language was used for those benchmarks, which are compiled by *GNU ARM* compiler, combined with *uClib* standard library, a specialized library for embedded systems. We implemented a prototype of our design technique with the simulator.

We selected *MiBench* to test small, real-time embedded programs which mainly perform computation-intensive programs. Among *MiBench* benchmark programs, we selected *JPEG encoder/decoder* program and automotive benchmark set, which is used for air bag controllers, engine performance monitors and sensor systems. In the automotive benchmark set, *Basimath* computes several mathematical calculations, *bitcount* computes bit manipulation, *qsort*

runs quicksort algorithm, and *susan* performs image recognition algorithm [16]. We also select *PapaBench* to test system-level, hard real-time program.

For our experiment methodology, we profiled the stack data access patterns of the benchmark programs at first, then applied our stack depth reduction technique and analyzed the trade-offs due to the reduction. Provided the maximum stack depth is already small enough or reduced enough with our technique, we propose to allocate the scratchpad memory for the whole stack area. Otherwise, the scratchpad memory is divided into several blocks, which are managed dynamically and statically. For the programs using dynamic and static scratchpad memory allocation, we compared our scratchpad memory performance with caches. Since we are handling the stack area, the cache was used only for stack data as well.

The compared memory types are as follows:

- LRU cache: Fully associative, write-back cache.
- Scratchpad memory with statically allocated blocks and one dynamically allocated block: This is our technique.
- Scratchpad memory with all the statically allocated blocks: To see the performance benefit of the dynamically allocated block, we tested the scratchpad memory using our static allocation technique only.

While dividing the scratchpad memory into 2, 4, and 8 blocks, the compared LRU cache is also divided as the same manners (2, 4, and 8 way fully associative cache, respectively).

To compare memory performance, read misses are counted because write miss penalties are assumed to become negligible by using write buffers between the on-chip memories and main memory controllers.

### B. The Stack Data Access Patterns of the Tested Benchmarks

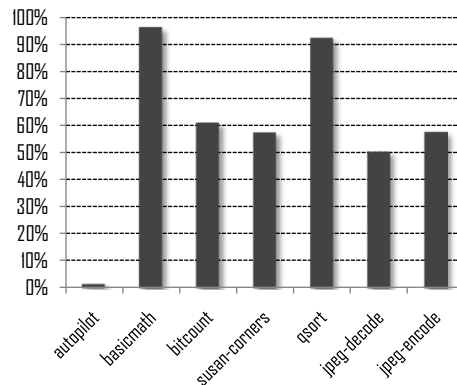


Figure 3. Call stack usage (%): proportion of data memory accesses over call stack over total data memory accesses.

We found that majority of the tested programs frequently use stack data. Figure 3 illustrates data accesses to call stacks

are frequent, even sometimes comprising of up to 92% of total data accesses. Therefore, using scratchpad memory for the stack is worth to improve performance of the most of the tested programs.

Only *autopilot* uses stack data rarely. This is because *autopilot* frequently uses global data, which communicate with its physical environment. Therefore, the memory accesses of global data are more often than those of stack data.

Program name	Size used	Benchmark Set
autopilot	128B	PapaBench
basicmath	468B	MiBench
bitcount	296B	MiBench
susan-corners	361KB	MiBench
quicksort	16MB	MiBench
jpeg-decode	2.1KB	MiBench
jpeg-encode	2.8KB	MiBench

Figure 4. Memory sizes used for call stack: the maximum stack depths are measured.

We also found that some tested programs use small stack sizes. Figure 4 shows that *autopilot*, *basicmath*, and *bitcount* use less than 0.5KB for the stack data. Therefore, for those tested programs, a small size of scratchpad memory can manage the whole stack area even without applying any algorithms.

Other programs need relatively huge memory size for their stacks. For those programs, we applied the stack depth reduction technique.

### C. The Effect of the Stack Depth Reduction

Program name	Original	After the change
susan-corners	361KB	788B (-360KB)
quicksort	16MB	728B (-16MB)

(a) The change of the maximum stack depths

Program name	Original	After the change
susan-corners	8.23KB	369.9KB (+361.6KB)
quicksort	8.1KB	7.3MB (+7.1MB)

(b) The change of the memory footprints (.bss)

Program name	Original	After the change
susan-corners	57.5%	10.2%
quicksort	92.3%	6.6%

(c) The change of the stack usage frequencies

Figure 5. The effects due to the source code modification

*susan - corners* and *quicksort* have large size of arrays in their stacks, thus we could reduce the maximum stack depth. However, even though *jpeg - encode* and *jpeg-decode* use huge stack area, the maximum stack depth cannot be reduced much because the depth becomes bigger

due to huge function parameters. Therefore, for those two benchmark programs, the stack depth reduction algorithm was not useful.

Figure 5 illustrates effects of the maximum stack depth reduction algorithm for the programs using large sizes of arrays (*susan-corners* and *quicksort*). In this result, the stack depth are reduced at the maximum. For both programs, the used stack data sizes are reduced so that less than 0.8KB of the scratchpad memory is enough to manage the whole stack area. Furthermore, for *susan - corners*, the decreased maximum stack depth is similar to the increased memory footprint, thus the trade-off of the memory footprint is not noticeable. On the other hand, for *quicksort*, the decreased maximum stack depth was more than the increased memory footprint. This is because small number of registers cannot handle large amount of stack variables (an array in this program) so that compiler reserves a huge stack area to handle them, but the reservation is not necessary when the array is moved to static variable. Therefore, the memory footprint is rather reduced in *quicksort*. However, as another trade-off, the stack data of the two programs are used less frequently, as described in Figure 5(c).

### D. Performance Comparison with Cache

Since maximum stack depths of *jpeg - encode* and *jpeg-decode* can't be reduced by our source code modification algorithm, we applied our dynamic and static scratchpad memory allocation to those programs. While comparing our design with caches, we focused on evaluating performance sacrifice to achieve predictability. As illustrated in Figure 6(b), when all the scratchpad memory blocks are allocated statically, performance degrade was huge. The miss rate was at least twice more than our technique. On the other hand, when one dynamic block is allowed to the scratchpad memory, the performance of two blocked scratchpad was better or similar to that of two-way LRU caches. This is because set associativity doesn't affect huge performance influences on the two-way cache. Other than that case, the other LRU caches performed better while increasing the scratchpad's blocks and cache's set associativity.

## VI. CONCLUSION

In this paper, to achieve predictability, we propose a simple but effective scratchpad memory allocation technique for the stack. With our strategy, the stack area can be managed only by scratchpad memory for several cases. Furthermore, even though the stack area is huge so that the scratchpad cannot handle it alone, we provide a scratchpad memory allocation algorithm to achieve predictability and reasonable performance.

## REFERENCES

- [1] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker, "A definition

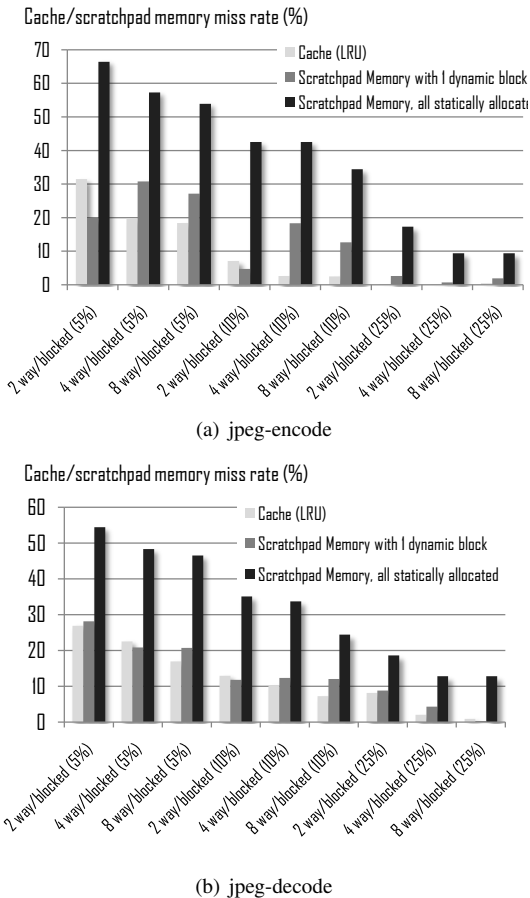


Figure 6. Performance comparison of our design with cache: Read miss rates are measured. %s of x-axis indicate the scratchpad memory sizes proportional to the maximum stack depths.

and classification of timing anomalies,” in *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, F. Mueller, Ed. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2006. [Online]. Available: <http://drops.dagstuhl.de/opus/volltexte/2006/671>

- [2] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand, “Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems,” *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 28, pp. 966–978, July 2009. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1669804.1669808>
- [3] S. A. Seshia and A. Rakhlin, “Game-theoretic timing analysis,” in *Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD ’08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 575–582. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1509456.1509584>
- [4] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, “Scratchpad memory: design alternative for cache on-chip memory in embedded systems,” in

*Proceedings of the tenth international symposium on Hardware/software codesign*, ser. CODES ’02. New York, NY, USA: ACM, 2002, pp. 73–78. [Online]. Available: <http://doi.acm.org/10.1145/774789.774805>

- [5] R. Co., “How to Get a uC/OS-II Application Running,” [http://ftp1.digi.com/support/documentation/0220047\\_e.pdf](http://ftp1.digi.com/support/documentation/0220047_e.pdf).
- [6] M. Schoeberl, “Time-predictable cache organization,” in *Proceedings of the 2009 Software Technologies for Future Dependable Distributed Systems*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 11–16. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1524877.1525241>
- [7] X. Vera, B. Lisper, and J. Xue, “Data cache locking for tight timing calculations,” *ACM Trans. Embed. Comput. Syst.*, vol. 7, pp. 4:1–4:38, December 2007. [Online]. Available: <http://doi.acm.org/10.1145/1324969.1324973>
- [8] O. Avissar, R. Barua, and D. Stewart, “An optimal memory allocation scheme for scratch-pad-based embedded systems,” *ACM Trans. Embed. Comput. Syst.*, vol. 1, no. 1, pp. 6–26, 2002.
- [9] N. Nguyen, A. Dominguez, and R. Barua, “Memory allocation for embedded systems with a compile-time-unknown scratch-pad size,” in *Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, ser. CASES ’05. New York, NY, USA: ACM, 2005, pp. 115–125. [Online]. Available: <http://doi.acm.org/10.1145/1086297.1086313>
- [10] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen, “Wcet centric data allocation to scratchpad memory,” dec. 2005, pp. 10 pp. –232.
- [11] J. Whitham and N. Audsley, “The Scratchpad Memory Management Unit for Microblaze: Implementation, Testing, and Case Study,” University of York, Tech. Rep. YCS-2009-439, 2009.
- [12] —, “Implementing time-predictable load and store operations,” in *Proceedings of the seventh ACM international conference on Embedded software*, ser. EMSOFT ’09. New York, NY, USA: ACM, 2009, pp. 265–274. [Online]. Available: <http://doi.acm.org/10.1145/1629335.1629371>
- [13] —, “Studying the applicability of the scratchpad memory management unit,” *Real-Time and Embedded Technology and Applications Symposium, IEEE*, vol. 0, pp. 205–214, 2010.
- [14] W. Qin, “Simit-ARM 2.1,” <http://simit-arm.sourceforge.net/>.
- [15] J. Edler and M. D. Hill, “Dinero IV,” <http://pages.cs.wisc.edu/~markhill/DineroIV/>.
- [16] 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 Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 3–14. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1128020.1128563>
- [17] F. Nemer, H. Cass, P. Sainrat, J. P. Bahsoun, and M. D. Michiel, “Papabench: a free real-time benchmark,” in *WCET’06*, 2006, pp. –1–1.