Modeling Cache Effects at the Transaction Level

Ardavan Pedram*, David Craven, and Andreas Gerstlauer

Department of Electrical and Computer Engineering
University of Texas at Austin,
Austin, Texas, USA
ardavan@mail.utexas.edu,{dcraven,gerstl}@ece.utexas.edu

Abstract. Embedded system design complexities are growing exponentially. Demand has increased for modeling techniques that can provide both accurate measurements of delay and fast simulation speed for use in design space exploration. Previous efforts have enabled designers to estimate performance with Transaction Level Modeling (TLM) of software processors but this technique typically does not account for the effect of memory latencies. Modeling latency effects of a cache can greatly increase accuracy of the simulation and assist designers in choosing appropriate algorithms. In this article, we show the implementation of a cache model and its integration into a processor TLM. We demonstrate a method for extracting information about memory accesses from the final binary and abstracting them into cache model accesses. Our methodology is tested on a common embedded processor application with two algorithms exhibiting different cache behaviors. Our experiments show that the cache model can achieve results comparable to a cycle-accurate ISS, but with very little overhead compared to native, host-compiled code execution.

Key words: Cache; Transaction Level modeling; System-level design

1 Introduction

Modern System-on-Chip (SoC) designs are becoming more complex as the capacity of chips is increasing dramatically. Deriving an accurate model for several candidate designs has always been a problem in the limited time given to the vendors to finalize their product.

Traditional Instruction Set Simulators (ISS) provide cycle-accurate precision of the functional and timing behavior, but their simulation speed is prohibitively slow. Thus, these simulators are often unacceptable for exploring the design space in the limited time available.

Increasing the level of abstraction in system modeling can increase the simulation speed by two or even three orders of magnitude. Transaction Level Modeling (TLM) [9] is the most commonly used and accepted approach, for abstracting

* This research was partially sponsored by NSF grant CCF-0702714. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

the system behavior both in communication and computation [15]. With this approach the simulation speed is dramatically increased, enabling a larger design space to be covered.

Modern processors contain at least one extra level of memory hierarchy e.g. cache between the CPU and the memory to take advantage of data locality and hide the memory latency. Loading and storing data from the cache incurs much less delay than direct communication with memory. However, due to their dynamic nature, accurate estimates of cache hit/miss rates and associated cache access delays are impossible to obtain statically. Yet, cache effects can have a significant influence on overall software execution performance [11].

Existing high-level, abstract processor TLMs for native, host-compiled execution of computation [15] currently do not take the behavior of a cache into account for execution-delay estimation. As it stands, the simulator back-annotates functions with a simple time-estimate based on the number of cycles the processor takes to complete a given instruction. Each memory access is assumed to take an approximate fraction of the execution delay between CPU and memory.

In order to accurately model a processor, the simulator must also model the behavior of a cache. Two different algorithms of the same application may have different cache access behaviors, which will cause one of them to outperform the other. In the current environment, not only could this not be determined from the TLM estimated delays, the model actually predicts the opposite relative performance for such algorithms because of their higher complexity.

In this paper, we describe our approach to integrating the cache model with a transaction-level model of the processor. We will present the overhead in simulation time and the increase in simulation precision compared to a normal processor TLM simulation.

1.1 Problem Definition

We use a standard system-level design language (SLDL) [5] to obtain TLM simulation results for an ARM processor. Furthermore, we utilize the SWARM ISS [4] to obtain cycle accurate information on execution. Our work is based on an existing processor and operating system model for accurate yet native, host-compiled software execution [15]. However, to the best of our knowledge none of the existing high-level processor modeling approaches provide a behavioral model of a cache for simulation-time estimation.

A cache simulator is provided and its implementation is discussed as a SLDL channel. Needed addresses are obtained from the final target binary and symbol table. The generated addresses are used in back annotated cache calls. The cache model dynamically updates its status as the simulation is run and returns appropriate delay for each access.

We demonstrate our approach using matrix multiplication as a test application. Matrix multiplication is the foundation of many embedded applications like Discrete Cosine Transform (DCT), and is convenient because it may be implemented with different algorithms with known different cache access behavior [10, 8]. We have implemented two matrix multiplication algorithms: a naïve matrix multiplication which exhibits poor cache utilization, and a cache aware matrix

multiplication that utilizes the cache to hide memory latencies. We show how the results for the TLM and TLM+cache model differ, and we use SWARM as our reference point for accuracy.

1.2 Outline

The rest of the paper is organized as follows: In Section 2 we discuss the related work. Section 3 describes the design steps taken to refine the application down to its target binary and to use that information for back-annotating cache calls into the TLM code. In Section 4, we give a brief explanation of our cache model and its interfaces. Section 5 introduces our application and presents our results. Finally, Section 6 discusses future work, and we will conclude our paper with a summary in Section 7.

2 Related Work

Traditionally, embedded software is validated using virtual prototyping environments that rely on instruction set simulation of processors [2]. ISS, can provide up to cycle-accurate results, but at the expense of slow simulation speeds, especially in multi-processor contexts. We use the Software ARM (SWARM) simulator [4] as such a cycle-accurate ISS reference to compare our approach against.

To provide fast virtual platform models, there are approaches for native, host-compiled software execution running in a model of the OS and the software environment. Our work extends previous approaches on such high-level, abstract processing modeling [15] based on the SpecC [5] SLDL. Our approach and results are, however, applicable to other processor models [3, 14] built on top of other C-based, event-driven SLDLs such as SystemC [6].

There have been similar attempts for high-level modeling of processors with caches for accurate simulation-time measurements. InterDesign Technologies uses a high speed CPU model for hardware/software co-simulation. Their product, FastVeri [1], converts software code into a virtual CPU model in SystemC. To keep cycle accuracy, FastVeri also back-annotates software code with delays from their instruction and data cache emulation. Their process for cache model integration is similar to ours in that the C source code is decomposed into basic blocks. Their approach, however, is proprietary and not easily extensible towards standard system-level design flows.

There are two possible methods for modeling a cache: mathematically or behaviorally. A mathematical cache model involves the derivation and use of cache miss equations for specific code patterns, and their evaluation to produce static delays [13]. While this is fast to simulate, it is necessarily limited to specific algorithms that match code patterns for which such an equation has been derived [7, 12]. Being based on static off-line analysis, mathematical models inherently introduce errors into the model, and cannot be extended to handle dynamic effects such as context switching.

3 Methodology

As previously mentioned, current processor models provide no method of modeling memory access latencies. A back-annotation provides execution delays by

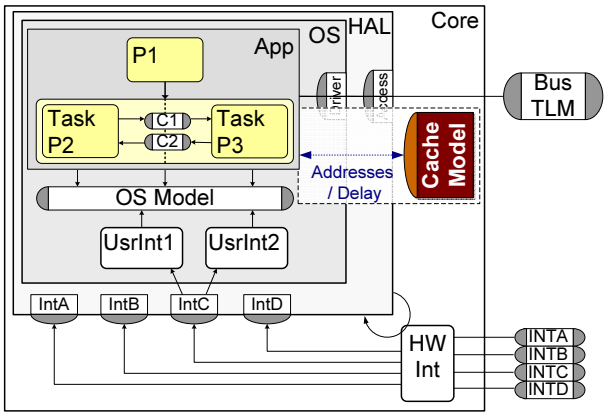

Fig. 1. Processor TLM organization with cache model.

inserting function calls into user tasks to update the simulation time with static delays . Delays can be obtained through estimation or ISS-based measurements. No runtime analysis or memory latency modeling occurs. We base our work on a processor model that has user tasks wrapped in a hierarchy of behaviors. This hierarchy includes a CPU as an execution unit for user code, a simplified model of an OS consisting of a task scheduler and drivers, a HAL for the OS and finally the hardware core [15]. In order to enable modeling of memory access latency within the TLM simulation, it is necessary to augment this processor model to include a model for a cache.

3.1 Cache Model Integration

As a hardware component, the cache fits most appropriately in the core level of the processor TLM, as shown in Fig. 1. We note that the OS model is very simple, and because it consists of only a task scheduler it makes no memory accesses that reflect real-world behavior. As a result, we cannot model memory activity by the OS itself. User tasks, by contrast, may make many accesses to memory for computation purposes, and we can observe that algorithms selected in the specification exactly reflect those in the resulting code generated for execution on the target platform.

Due to the likelihood of memory accesses that are data-dependent, such as array indexing, it is necessary to model memory behavior at run time when that data is available and the address being accesses can be computed. To accomplish this, we follow the general back-annotation approach and introduce function calls to update the cache model as necessary.

3.2 Base Address Acquisition

Currently, we support access to static global variables/arrays data in our solution. Addresses for our cache model are computed at execution time of the TLM by code that is back-annotated into user behaviors.To compute these addresses, two pieces of information are needed. First, we need the static base

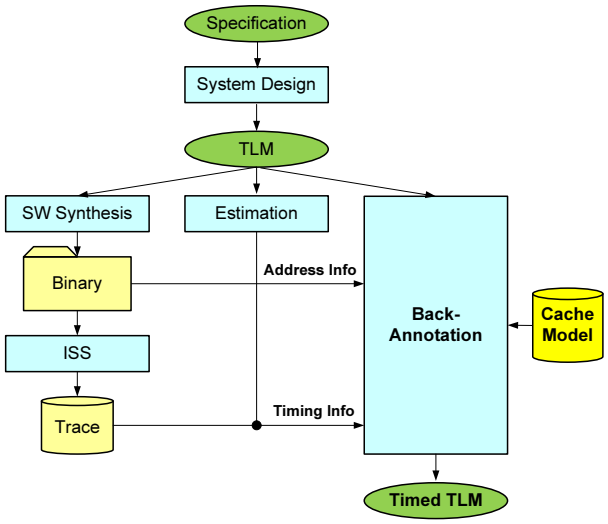

Fig. 2. Design flow.

address from which the memory offset is calculated, which is available prior to execution. The dynamic offset from this base address is required (in the case of accessing arrays), and this is available only at runtime. Calls to the cache model secondly pass the memory address being accessed and the cache model will return a modeled latency based on whether the data was available in the cache or not.

Obtaining the base addresses requires intimate knowledge of the memory layout of generated code. As shown in Fig. 2, the needed information about memory layout can be obtained from the target binary. This binary is generated starting from the user specification by proceeding through the system-level refinement process down to a TLM of the system. A software synthesis process generates target C code that will run on the final system. This code is then cross-compiled for the target processor to obtain the final target binary. At this point the symbol table with base addresses for global variables has been created. The binary is suitable for simulation on an instruction set simulator to verify results or to obtain accurate information about the execution delay of the various blocks of code.

3.3 Back Annotation

The second step for creating our cache-augmented TLM is to insert the cache model into the basic TLM, and back-annotate it with the proper cache model calls. As seen in Fig. 2, the back annotation process pulls address information from the synthesized binary, basic block timing information either from an estimation tool or from a cycle accurate simulation, and injects API calls to the cache model.

```
MatrixC[i][j] += sum;
/*__BACKANNOTATED: enqueue cache access*/
accesslist[index]= MatrixC_Base +
(sizeof(int)*(i*MAT_WID+j));
accesslist[index+1]= MatrixC_Base +
(sizeof(int)*(i*MAT_WID+j));
index+=2;
/**/

…

/*__BACKANNOTATED: accumulate BB delays */
cache_delay =
cache_port.cache_call(accesslist, index);
cumulative = BASE_DELAY + cache_delay;
waitfor ( cumulative );
/**/
```

Fig. 3. Back annotated code for a basic block.

A cache channel is instantiated in the core behavior of the TLM. Back annotation also inserts memory address computations into user code wherever a global variable appears in a statement. Cache calls are inserted to update the model and obtain the memory latency as shown in Fig. 3. Currently, this process is done manually, but we plan to automate this process in the future. The code snippet shows an example of a back-annotated memory address computation and cache model update for a basic block of code.

4 Cache Model Implementation

This section details the process of implementing and integrating the cache model into the processor model. The purpose of integrating the cache model into the TLM is to dynamically update and simulate the behavior of a cache in terms of hit and miss delay. Because memory accesses may be data-dependent, and because the OS model may context-switch between tasks, our cache is maintained dynamically at runtime.

4.1 Cache Model

We use a behavioral modeling approach, which involves the use of data structures that emulate the state of a cache at any point in time. These structures are updated at runtime as memory is accessed. This method allows the user to observe the state of the cache at all times and can model any memory access pattern. Its drawback is a performance penalty due to additional overhead for each memory access to maintain the cache state. We chose to dynamically model a behavioral cache for greater flexibility. Our cache model consists of less than 200 lines of SpecC code and can easily support various cache organizations.

There are two options in the SpecC language for implementing a cache model: as a behavior or as a channel. A behavior consists of a main method, and is

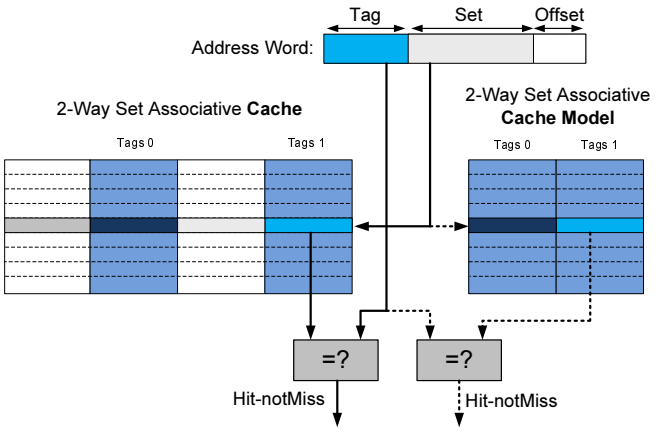

Fig. 4. An example of a typical 2-way set associative cache(left) and our tags matrix representation of the same configuration(right).

associated with an active thread while a channel is a passive component [5]. Since the cache state only needs to be updated when it is being called by a behavior, we chose to build the cache as a SpecC channel. This channel implements an interface that defines *cache_init*() and *cache_call*() functions available for cache communication.

The *cache_init*() function is used to create and initialize the cache model. It is called before the OS is instantiated. It allows the user to configure the cache model based on the number of sets, associativity, and cache line size. The *cache_call*() function provides behaviors a means of updating the cache state and of obtaining cache access delays. To reduce function call overhead, its interface accepts an array of cache accesses containing an ordered sequence of memory accesses with which the cache will be updated and for which a total delay is returned.

4.2 Address Tags Matrix Implementation

Since our cache model only needs to take into account occurrences of cache hits or misses to calculate the delay, we model only functional behavior of the cache, e.g. whether a particular address does or doesn't exist within the cache. For our purposes, we are not concerned with the data that would be stored. As such, our model is simply a matrix of address tags where the rows represent the number of sets and the columns represent the associativity. Fig. 4 shows an example of a typical two-way set associative cache on the left, and our implementation of it using the tags matrix on the right. As shown in the figure, the tags, set, and offset are determined by the address bits. As described, the cache model on the right does not hold any data.

In a typical cache, there are three types of misses which differ by cause: compulsory, conflict, and capacity [4]. A compulsory miss occurs on a first access, a conflict miss happens when the cache is not associative enough, and a capacity

miss means the cache size itself is not large enough. When a miss occurs, a replacement policy is needed to appropriately evict an old tag from the cache and replace it with the new tag. Our model implements the most commonly used algorithm, the Least Recently Used (LRU) replacement policy. In this method, the tag that was the least recently used within a set is evicted from the cache and replaced with the new, most recently used tag. To implement this functionality, we append a current time stamp to each element as it is placed in the address tags matrix. In this way, when a tag needs to be removed from the cache, the element with the smallest time stamp is the least recently used. We define the time stamp variable as an unsigned long long integer, which takes values up to 2^{64}, to ensure that it does not roll over to zero as it increments. This means that the cache must be accessed more than 2^{64} times for a single application before the LRU replacement policy is corrupted.

4.3 Cache Access Implementation

A delay time in cycles is calculated for each address that is passed to the cache model through the address vector. To ensure correct functionality, and therefore correct delay times of the cache, we must constantly maintain the state of the cache model. The *cache_call*() function generates an address tag, a set number, and an offset from the address of each cache access. It then searches the specified set in the tags matrix for a matching address tag. If the tag is found it updates the LRU cache status and returns an appropriate hit delay. If the tag is not found in the matrix, the cache algorithm employs the LRU policy to evict one of the tags in the set, updates the state of the cache, and returns a larger cache miss delay.

5 Experimental Results

Matrix-matrix and matrix-vector multiplication are the core of many applications based on solving linear systems, filtering, and media processing. Some examples of these applications include least squares, FIR and DCT filters. Matrix-matrix multiplication is also considered the core computational step for many matrix based computations such as matrix decomposition and inversion for solving linear and non-linear differential equations.

The main concern in matrix multiplication is the memory latency. If the algorithm does not take advantage of the data locality in different levels of the memory hierarchy, it will suffer a performance penalty waiting for data.

We chose to simulate two algorithms, a typical naïve matrix multiplication algorithm and a cache-aware, blocked algorithm. For each algorithm, we simulated both small size matrices that would fit entirely in the cache and large size matrices that could not. Results were obtained from the SWARM instruction set simulator, the TLM simulation, and our back-annotated TLM with cache model. We will discuss results for the small and large data set separately, comparing speed and precision for the three simulation approaches.

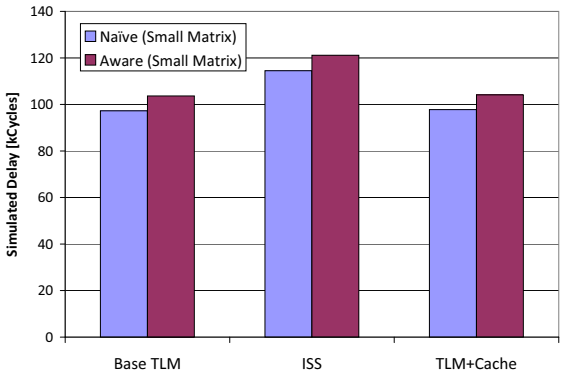

Fig. 5. Small matrix (16×16) Simulated delays.

5.1 Accuracy

For our experiments, we obtain a base cycle delay from the ISS that does not take into account cache delays and back-annotate this into a basic timed TLM. We compare this against our extended TLM+Cache that reports simulation delay as the sum of this base delay and the delay computed by our cache model. Table 1 summarizes the results gathered for all simulated cases.

To match undocumented target system and simulator characteristics, we need to characterize and calibrate the miss penalty assumed in the reference ISS model. The ISS produces the precise number of cache misses, the number of real cycles, which includes cache miss penalties, and the number of logical cycles, which does not include cache miss delays. We estimate miss penalty by subtracting logical cycles from real cycles and dividing by the number of misses, using large problem sizes with many cache misses to minimize error. From this we obtained a miss penalty of 10 cycles.

For the small data set, we expect the two algorithms to perform similarly. The cache-aware algorithm should hold no advantage for data accesses and may run slightly slower due to increased code complexity over the naïve implementation. As shown in Fig. 5, the difference in performance between the two algorithms remains the same for the the ISS considering cache delays, base TLM cycles, and our cache modeling TLM because both algorithms have approximately the same memory behavior due to the small matrix size.

Table 1. Accuracy Results

Simulation	**Simulated Delay (Cycles)**			
	Naive Larger	**Aware Large**	**Naive Small**	**Aware Small**
Base TLM	285,999,032	330,511,500	97,276	103,641
ISS	472,986,462	334,785,093	114,500	121,129
TLM+Cache	469,066,892	395,387,630	97,766	104,131

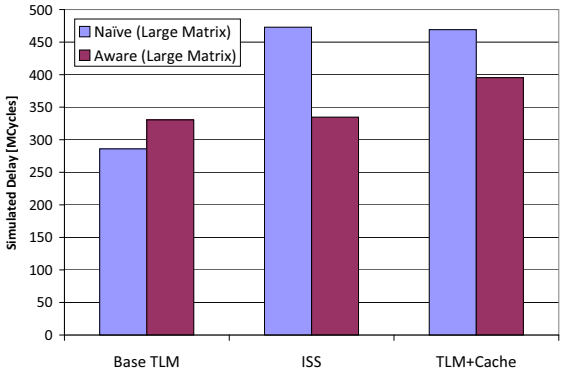

Fig. 6. Large matrix (256×256) simulated delays.

The large matrix size problem should greatly favor an algorithm that takes advantage of the memory hierarchy. We expect that the cache aware algorithm will amortize the cost of extra instructions by hiding the memory latency penalty. The naïve algorithm exhibits a high cache miss rate and should be heavily penalized in any simulation that takes this into account.

The results in Fig. 6 show that the ISS strongly favors the cache aware algorithm with almost a 30% increase in speed. The cache aware algorithm suffers in the base TLM because of its higher number of instructions. We expected our cache-modeling TLM to eliminate this gap to match the expected relative performance of the two algorithms as seen in the ISS. The cache-modeling TLM correctly reflects the advantage of the cache aware algorithm seen in the ISS.

5.2 Simulation Time

For design space exploration, it is necessary to obtain a variety of simulation results for many different possible implementations. This makes the time to execute the simulator important, as it can be a limiting factor.

As seen in Fig. 7, we incur some overhead over the TLM, but even for large matrix sizes we are no worse than an order of magnitude slower. This penalty is worthwhile considering that the TLM model is very fast, requiring fractions of a second to complete. The ISS, however, is four orders of magnitude slower than the TLM model, and requires more than two hours for a large matrix size. It was frequently the case that we completed back-annotation by hand for our TLM with cache and obtained results before the ISS could finish execution. By contrast, our TLM+Cache simulator completes this simulation in just over two seconds.

6 Future Work

Future efforts to integrate cache effects into TLM will need to focus on a few important areas of improvement. These next steps include automation, expansion to track more variables, and additional more realistic testing to determine conditions most favorable to this method.

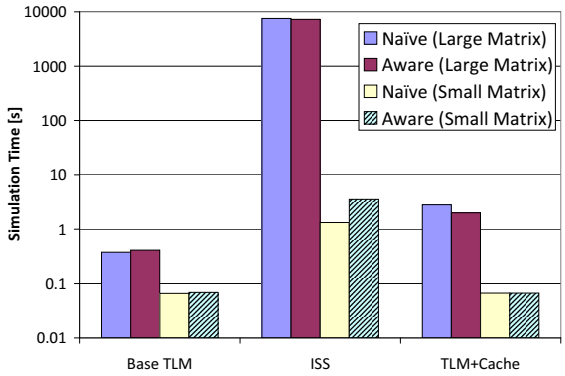

Fig. 7. Model simulation times.

First, for back-annotation techniques to be useful as a modeling tool, they require implementation as an automated tool. Because we have available detailed information from the target binary during our process, future work may also choose to utilize this information to replace the current estimation tools with one that makes use of the additional detail.

Second, while our results show that tracking only global memory accesses is effective, ideally all variable accesses should be tracked. Hence the ability to trace accesses to stack variables would be desirable and could be achieved by integrating with the OS model to track the stack pointer of each task.

Third, additional testing is necessary to ensure that this technique applies well to a variety of algorithms on a variety of target platforms. We plan to apply the approach to a variety of industrial-strength MPSoC application, e.g. to evaluate performance of different DCT implementations in standard image or video processing algorithms. Larger models with multiple threads could allow us to determine the effects of cache pollution on performance, but to do so may require enhancement of the OS model to allow finer granularity preemptive multitasking than is currently possible. More control over the configuration of the ISS cache model would permit testing the technique against a variety of cache types and sizes. Expanding the set of test cases may also help demonstrate cases in which this technique is most helpful.

7 Summary and Conclusions

In this paper, we showed the implementation and integration of a configurable cache model into the system TLM in order to achieve higher precision for simulation. Our process requires us to refine the specification all the way down to the target processor binary. We gather critical information from the binary code and back-annotate it as input parameters for cache calls in the application code of the TLM.

We selected matrix multiplication as test application because of its well-known behavior under different variations. We chose a cache-aware, blocked ma-

trix multiplication and a naïve algorithm, two algorithms with vastly different cache access behavior, to see how our cache model reflects these behaviors in the simulated cycle counts.

Our experimental result show that the TLM including the cache model does not have a significant simulation time overhead compared to a normal TLM. On the other hand, it is three orders of magnitude faster for large matrix sizes than executing on the ISS.

The TLM with cache accurately shows that extra complexity of cache-aware algorithms is amortized by the reduction in memory access penalty. Our method of back-annotation and cache modeling allows us to model delays with 100% fidelity compared to the ISS while maintaining execution speeds similar to TLM, facilitating rapid, early design space exploration.

References

1. D. Araki, N. Ito, T. Shinsha, and Y. Mori. High speed hardware/software co-verification with cpu model generator from software code. Technical report, Inter-Design Technologies Inc., 2006.
2. L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri. MPARM: Exploring the multi-processor SoC design space with SystemC. *VLSI Signal Processing*, 41(2):169–182, 2005.
3. A. Bouchhima, I. Bacivarov, W. Yousseff, M. Bonaciu, and A. A. Jerraya. Using abstract CPU subsystem simulation model for high level HW/SW architecture exploration. In *ASP-DAC*, Shanghai, China, Jan. 2005.
4. M. Dale. *SWARM Instruction Set Simulator*. `http://www.cl.cam.ac.uk/~mwd24/phd/swarm.html`.
5. A. Gerstlauer, R. Dömer, J. Peng, and D. D. Gajski. *System Design: A Practical Guide with SpecC*. Kluwer, 2001.
6. F. Ghenassia. *Transaction-Level Modeling with Systemc: Tlm Concepts and Applications for Embedded Systems*. Springer, 2006.
7. S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: an analytical representation of cache misses. In *ICS*, Vienna, Austria, 1997.
8. K. Goto and R. A. v. d. Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3):1–25, 2008.
9. T. Grotker. *System Design with SystemC*. Kluwer, 2002.
10. F. G. Gustavson, A. Henriksson, I. Jonsson, B. Kågström, and P. Ling. Superscalar GEMM-based level 3 BLAS - the on-going evolution of a portable and high-performance library. In *PARA*, London, UK, 1998.
11. J. Hennessy and D. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, 2003.
12. I. Hur and C. Lin. Modeling the cache effects of interprocessor communication. In *PDCS*, Cambridge, MA, Nov. 1999.
13. Y. Hwang, S. Abdi, and D. Gajski. Cycle approximate retargettable performance estimation at the transaction level. In *DATE*, Munich, Germany, Mar. 2008.
14. H. Posadas, J. A. Adamez, E. Villar, F. Blasco, and F. Escuder. RTOS modeling in SystemC for real-time embedded SW simulation: A POSIX model. *Design Automation for Embedded Systems*, 10(4), Dec. 2005.
15. G. Schirner, A. Gerstlauer, and R. Dömer. Abstract, multifaceted modeling of embedded processors for system level design. In *ASP-DAC*, Yokohama, Japan, Jan. 2007.