Modeling Shared Resource Contention Using a Hybrid Simulation/Analytical Approach

Alex Bobrek Joshua J. Pieper Jeffrey E. Nelson JoAnn M. Paul Donald E. Thomas

Electrical and Computer Engineering Department
Carnegie Mellon University
{abobrek, jpieper, jnelson, jpaul, thomas}@ece.cmu.edu

Abstract

Future Systems-on-Chips will include multiple heterogeneous processing units, with complex data-dependent shared resource access patterns dictating the performance of a design. Currently, the most accurate methods of simulating the interactions between these components operate at the cycle-accurate level, which can be very slow to execute for large systems. Analytical models sacrifice accuracy for speed, and cannot cope with dynamic data-dependent behavior well. We propose a hybrid approach combining simulation with piecewise evaluation of analytical models that apply time penalties to simulated regions. Our experimental results show that for representative heterogeneous multiprocessor applications, simulation time can be decreased by 100 times over cycle-accurate models, while the error can be reduced by 60% to 80% over traditional analytical models to within 18% of an ISS simulation.

1. Introduction

To take advantage of billion transistor chips expected within the next five years, most current views of Systems-on-Chips (SoCs) involve multiple processing units, shared resources, and networks-on-chip. The challenge posed to the designers of such systems is to effectively juggle these SoC design elements, producing systems with high performance, but low power consumption, size, and cost. In these Programmable Heterogeneous Multiprocessor (PHM) systems, complex interactions between processing elements, scheduling strategies, memory access times, and communication system latencies will emphasize the design and programmability of the system as a whole, not just as a collection of individual programmable units. Due to the importance of these data-dependent element interactions, modeling them in a complete system at a high level will play an important role. This modeling will enable early system performance estimation and efficient traversal of the design space, both to optimize the performance of a programmable design and to optimize an architecture for a given fixed performance.

Perhaps the most important characteristic of PHMs is the system reliance on shared resources. Shared memory, the interconnect between processing elements, and I/O interfaces are all accessed by multiple processing elements, making the contention for shared resources a significant influence on the overall system performance and an important modeling consideration. However, most simulation of shared resource interaction is done at a cycle-accurate level. While the detail of such models leads to accurate simulation, they suffer from slow simulation speed as well as extensive up-front development time. At a higher level, analytical models decrease the model detail significantly by estimating performance based on statistical interactions and average behaviors. But, these average-based models do not perform well with applications exhibiting irregular shared resource access behavior, incorrectly estimating the amount of contention and making the analytical model of limited utility [2]. In general, while lowering the model detail improves simulation speed, it becomes increasingly difficult to generate accurate models of shared resource interaction.

We present an extension to the MESH [9] simulation framework for modeling shared resource contention that is faster and more abstract than a cycle-accurate simulation, yet of higher utility to the designer than purely analytical models. As such, this model is uniquely suited to be the first timed model the designer considers immediately after the system specification, thus enabling discovery of performance at high level. This paper's major contribution is a simulation kernel implementing a novel hybrid shared resource model. The hybrid model applies analytic contention models to groups of shared resource accesses derived from system simulation. Thus, our approach combines the ability of simulations to capture dynamic, data-dependent system behavior with the superior speed of analytical approaches.

2. Prior Work

Historically, shared resource contention models fall into either the simulation or analytical categories. Traditional simulation approaches operate at the cycle-accurate level, employing detailed contention models of shared resources at the cost of execution time [4] [5]. On the other hand, the analytical models, based on averages or statistical properties, abstract away much of the detail in a system in exchange for quick evaluation times [2].

There have been other attempts at modeling embedded systems higher than the instruction set level, but at a level lower than purely analytical approaches. Ptolemy [7] is one

1530-1591/04 $20.00 (c) 2004 IEEE

such simulation framework, focusing on simulating systems comprised of heterogeneous models of computation. Real-time models [11] attempt to find worst case execution times or verify software timing. Unlike these approaches, we look to capture data-dependent concurrent performance of software executing on hardware within PHM systems. Discrete event simulations have been applied to PHM systems as well [13], with all times described directly as physical numbers, in contrast to our approach as described in Section 3.

Our system simulation method draws upon many techniques from single processor software timing estimation to determine the physical timing of software. The object-based approach of [1] is similar to our technique for simulation above the instruction set level, but has several differences. In our approach, the annotations represent an abstract measure of computational complexity rather than physical times. Also, we allow annotations to be spaced arbitrarily far apart, instead of at the assembly or source code line level.

The work that most closely resembles the method described in this paper for combining a simulation and analytical approach is [3]. They develop a model of shared network lines for estimating large scale Internet performance at a transfer level, rather than a packet level. However, their approach is more limited since it is applicable only to network topologies and specifies only one very simple method of resolving contention. We consider the general case of any shared resource, while allowing analytical models to be interchanged for each individual shared resource within the simulation.

3. Simulation Methodology

The MESH simulation framework is based on a layered model composed of a theoretically unlimited number of dynamic logical threads (Th_L) running on top of a scheduling layer (U_E) that interfaces with a physical thread layer (Th_P) as seen in Figure 1a. Logical and physical threads both consist of an event set; in physical threads this set is totally ordered, in logical threads it is only partially ordered [10] [8]. These event orderings make logical threads uniquely suited for modeling of software (e.g. instructions may be executed out of order) where the physical threads are more suitable for hardware models (e.g. it is not possible to reorder gates within a chip). The scheduling layer resolves the partial ordering of events in logical threads to physical time. It also models system-state-aware scheduling algorithms that can affect the performance of a PHM system [9].

Logical threads are expressed by annotating arbitrary C code with *consume calls*, creating *annotation regions*, and thus indicating the computational complexity of software within that region. Values associated with consume calls can be derived from techniques such as profiling, designer experience, or software libraries. Consume call values are meant to indicate software computational complexity, not physical timing, and should not be confused with physical timing annotations such as Verilog's "# delay". Like the # delay, code that lies in annotation regions between consume calls is executed in zero virtual time. Unlike the # delay or the general

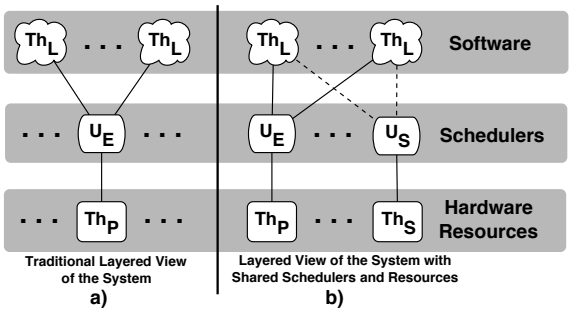

Figure 1. Adjustments to the Layered View

discrete event approach, after the code within the annotation region is executed, our simulation determines the physical timing of the region using the computational complexity values passed through the consume calls. Thus, the annotations dictate the finest unit of timing resolution available. Because of this, the spacing of annotations is the primary determinant of simulation accuracy and run-time.

Physical threads are described by a computational power (computation per unit time). The scheduling layer can use this power to determine when and where logical threads should execute, and the resulting advance in simulation time. Modeling the scheduling layer separately captures its importance to the system as a whole. By regulating the access of software threads to hardware resources, it provides a global system control flow across resources. Thus, using the layered model described above, the MESH simulator can model the behavior of heterogeneous processors executing parallel software threads.

Significantly, this model can be seen as equivalent to lower levels of modeling according to how annotations are placed. For example, an ISS simulation is approached if annotations occur after each assembly instruction. While it is possible to use such a relatively low-level modeling paradigm, MESH is designed to model PHM systems at a much higher level and is most suited to quick and early model development and design space exploration.

4. Shared Resource Modeling

Our proposed framework incorporates aspects of both simulation and analytical modeling methodologies, resulting in the ability to decrease simulation time with minimum impact on accuracy. It simulates parallel threads for a period of physical time determined by software annotations, temporarily ignoring contention for shared resources. All accesses to any shared resources encountered during these regions of time are grouped and sent to an analytical model which assigns time penalties to each competing logical thread. Simulation continues, with future logical to physical timing resolution including these penalties. The time penalties shift the execution time of any logical threads running on the penalized physical resource to a later physical time, effectively modifying the system state trajectory (performance over time) and modeling the degraded perfor-

```
1  while (active Th_L remain) { //main kernel loop
2    for each available Th_P {
3      invoke U_E to schedule a Th_L to run
4      execute this region (R) until annotation
5      resolve logical to physical timing for region R
6      insert region R into priority queue sorted by
         physical end times
7    }
8    grab annotation region R_top from top of priority
       queue (has lowest end time t_i)
9    while (R_top has unapplied penalty) {
10     add penalty to R_top end time t_i, zero out
         penalty, and re-insert R_top into queue
11     grab new R_top
12   }
13   remove R_top from queue
14   advance system time to t_i from region R_top
15   apply analytical model(s) for each shared
       resource from t_i-1 to t_i
16   assign penalties according to analytical model
17   if ( R_top received a penalty)
18      add penalty to R_top end time t_i, zero out
          penalty, and re-insert R_top into queue
19   else mark R_top's resource as available
20 }
```

Figure 2. Simulation kernel operation with shared resource modeling

mance due to a contended shared resource. We describe our approach in detail later in this section.

4.1 Layering

We extend our layered view from Figure 1a to include shared resources and models for contention resolution. In Figure 1b we introduce shared resource threads (Th_S) to contrast their behavior from the existing execution resource threads (Th_P). The function of each Th_P is to resolve the logical ordering of events in software (Th_L) into physical time based on the amount of computation the physical resource can perform in a given unit time. In contrast, the function of each shared resource thread (Th_S) is to apply time penalties to each Th_L that has accessed the Th_S. This is done through the application of an analytical model(s) associated with each shared resource thread. This fundamental difference between execution and shared resource types leads to their separation in Figure 1b.

To model access contention within shared resources we make changes to our scheduling model as well. The shared resources are managed by a new type of scheduler, the shared resource scheduler (U_S). Each annotation may now be a tuple, containing a value to pass to its U_E and possibly multiple additional values, one for each U_S; this is a major break from the discrete event approach. The shared resource schedulers are responsible for allocating these shared access requests onto shared resources much like execution schedulers allocate software thread computation onto physical resources. The key difference between the execution schedulers and shared resource schedulers is that the former arbitrates between the logical threads *prior* to resource access where the latter applies penalties *after* the resource access is completed; this permits us to consider annotation regions in groups across shared resources.

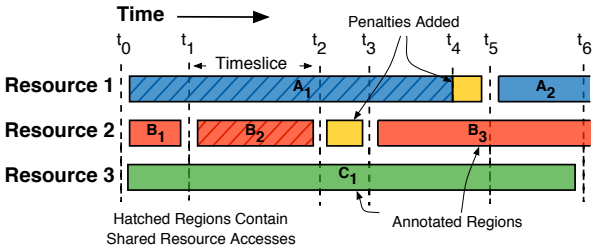

Figure 3. Timeline illustration of kernel operation

Since there is no entity that provides arbitration prior to the access for a shared resource (as does the execution scheduler), contention causes queueing delays within logical threads waiting for service. Shared resource schedulers implement post-access arbitration by applying penalties to logical threads exhibiting shared resource access contention. The amount of contention penalty is determined by an analytic contention resolution function provided by each shared resource model. Note that even though a logical thread can only be associated with one execution scheduler, the same thread can be associated with multiple shared resource schedulers, representing that a thread can access more than one type of shared resource (memory, communication medium, I/O devices, etc.).

4.2 Simulation Kernel Algorithm

The pseudocode in Figure 2 illustrates the simulation kernel with shared resource modeling included and Figure 3 depicts a sample run of the algorithm graphically over time. The kernel picks an available resource, invokes the appropriate U_E to schedule an eligible thread on it (line 3), and executes the thread until a user inserted annotation is encountered (line 4). The figure shows three threads named A, B, and C running on three resources with any thread eligible to run on any of the resources. In this simplified case threads do not switch resources, however, in general U_E schedulers can handle arbitrary scheduling schemes. At t_0 all resources are available and the scheduler maps the A thread onto Resource 1, the B thread onto Resource 2, and the C thread onto Resource 3. Computational complexity, specified by the annotations in the logical threads, is resolved to physical time by means of the computational power of each physical resource (line 5). For example, in Figure 3, the first annotation region of the B thread (B_1) executes from t_0 until t_1 in virtual physical time.

The physical end time of the executed annotated region of each logical thread (t_4 for A, t_1 for B, and t_6 for C) is then pushed onto a priority queue. The priority queue ensures that the earliest physical end time is always available on the top of the queue (line 6). We leave the explanation of lines 8 through 12 until later in the section. The earliest annotation is *committed* in line 13 by removing the annotation end time from the top of the queue and advancing the global simulation time to that point (line 14). In the figure, B_1's annotation region ends the earliest in physical time (t_1), so it is located on top of the priority queue where it

is retrieved and committed first. Shared resource accesses are analyzed in line 15, but since only thread A accessed the shared resource between t_0 and t_1, there is no contention and no penalties are applied in line 16. Since no penalty was applied, the resource is marked available in line 19.

The simulation time is now at t_1 and Resource 2 is available and eligible to execute a thread. During the next iteration of the kernel loop, B_2 is scheduled, executed, retrieved from the priority queue, and time advanced to t_2. Since annotation regions on different resources need not align in physical time, the simulator performs *timeslicing*, that is it considers only the time period between adjacent annotations' end times. The timeslice end times are represented by the dashed lines associated with physical time locations. If an annotation region with multiple shared resource accesses is broken into multiple timeslices, the shared resource accesses are proportionately divided among the timeslices. Once the quantity of shared accesses per timeslice per thread is known, this information is passed to the analytical model of each shared resource.

In the current iteration (slice t_1-t_2), both thread A_1 and B_2 contend for the shared resource. The analytical model assigns some queueing delay penalty to both contending threads (line 15) , resulting in the actual physical time for B_2 being extended as well as that for A_1. The assigned delay can vary for each contending thread. For instance, if a priority arbitration scheme is being modeled, the high priority thread may receive a lower average penalty. Here, since B_2 had a penalty applied, its resource is not marked available, and its physical end time is immediately re-inserted into the priority queue (line 18). The other end times are not updated instantly, instead penalties are accumulated until the next end time is encountered. This behavior will be illustrated when t_4 is reached.

After the penalties are assigned, the main kernel loop executes again. However, this time no resources are available so execution skips to line 8. Here the first region removed from the priority queue is B_2 at t_3, and it has no penalties remaining (they have been zeroed out once they have been added to region end time in line 18). Thus this region is committed, and shared resource analysis is performed. Since all of the region B_2's shared resource accesses were considered during time slice t_1 to t_2, the penalty time assigned to B_2 has no additional shared accesses contained within. Therefore there is no contention in timeslice t_2-t_3, so no penalties are applied.

The loop repeats, schedules B_3, executes it, then looks for the nearest physical end time. The nearest end time is thread A at t_4. Line 9 notes that this thread has unapplied penalty (assigned during t_1 to t_2 timeslice), applies it, and reinserts the end time (line 10 - 11). Notice that application of a penalty does not create a new timeslice at t_4 as was the case in the t_1 to t_2 timeslice. Line 9 then grabs the new nearest time, which happens to again be thread A, but now at t_5. Region A_1 is committed, and shared resource access analysis is performed over t_3-t_5, with no further contention or penalties being applied. Note that the physical time penalties effectively shift the execution of any threads on that physical resource later in time by the amount of the penalty. In Figure 3, penalties accrued during timeslice t_1-t_2 are added after regions A_1 and B_2. Therefore, the timing of a software region is not only dependent on the resolution of computational complexity into physical timing, but on penalties applied by the shared resource contention model as well.

4.3 Additional Features

For systems with numerous threads and resources, physical time misalignments of annotation regions can create a large number of small timeslices, increasing the runtime of the simulation. We combat this problem by introducing a parameter limiting the minimum timeslice size to a designer specified minimum. The algorithm avoids creation of timeslices smaller than the minimum by accumulating shared resource accesses of undersized slices and performing the analysis together with the next large timeslice. By moving the analysis of small timeslice shared resource accesses to a later time, the designer can choose to trade off small amounts of accuracy to keep the number of timeslices down.

In addition to the ability to limit the timeslice size, we provide the designer a full set of synchronization primitives commonly found in threaded programming libraries (mutexes, semaphores, condition variables). These allow the inter-thread data dependencies to be observed. If a synchronization primitive is encountered that requires blocking, the executing annotation region is shelved and the resource marked available until synchronization can be resolved. Since a blocked logical thread frees up the resource, the scheduler is capable of scheduling other tasks onto the resource. Once the event that a blocked thread is waiting on occurs, the shelved annotation region is allowed to continue. Since the simulator only knows the annotation region that the unblocking event occurred in, the restarted region is placed at the end of the unblocking event region's physical time. This is a pessimistic assumption and can cause errors with coarsely annotated threads requiring continuous synchronization. How to avoid fine grained annotation, or when to relax our assumptions in this case, is an area of future work.

5. Example

We explore the validity of MESH's hybrid simulation/analytical approach by considering two examples. In the first example, we will compare our approach to an analytical model used to estimate bus contention on a traditional homogeneous multiprocessor running an FFT application. This example will show our model's ability to adjust to changing program behavior over time while running significantly faster than cycle-accurate simulations. In the second example, we will apply our approach to a scenario more suitable to a SoC: a heterogeneous system executing various kernels from the MiBench [6] embedded benchmark suite. This example will show how execution of multiple programs across heterogeneous architectures necessitates the use of piecewise analytical models as opposed to traditional ones.

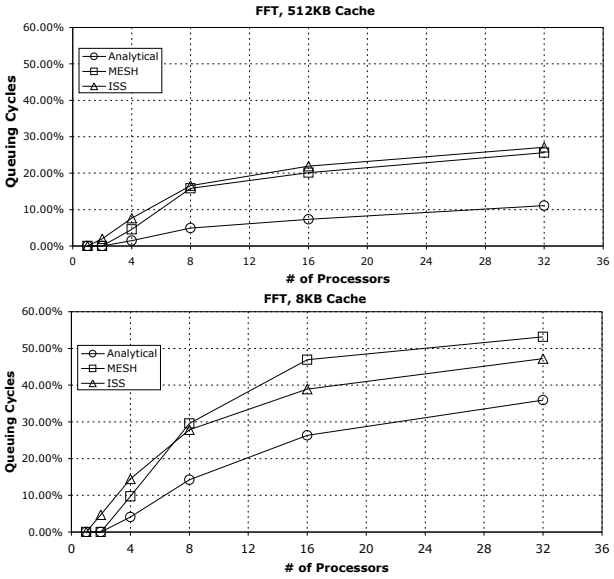

Figure 4. Modeling of SPLASH-2 FFT Benchmark

5.1 SPLASH-2 FFT

In the first example, we compare our hybrid approach to an analytical model developed by Chen and Lin in [2] when applied to the SPLASH-2 [12] FFT benchmark. The FFT application was chosen because it exhibited irregular shared bus behavior over time, causing the analytical model to have a large queuing cycle (number of cycles spent waiting due to bus contention) estimation error. In the other SPLASH-2 benchmarks the Chen-Lin model performs well, as does the corresponding MESH model. To focus on the benefits of our hybrid model, we used the same Chen-Lin model within our simulation kernel to apply penalties to timeslices. Thus, the only difference between the traditional Chen-Lin model (referred to as "analytical" in the figures) and the MESH hybrid model is that the MESH simulation performs a piecewise evaluation of the Chen-Lin model over periods of execution time where the traditional Chen-Lin model is applied in one step across the whole runtime of the program.

We placed annotations at every synchronization point (barrier statement) in the original SPLASH-2 FFT algorithm. This level of granularity is sufficient to capture the irregular behavior of shared resource accesses in this application and greatly improves the performance of the purely analytical model. In Figure 4 we compare the percentage of queuing cycles estimated by the purely analytical and MESH hybrid approaches to the baseline case, the cycle-accurate ISS simulation. As can be seen in the figure, the piecewise application of the Chen-Lin model through the MESH framework decreases the percent error of predicted queuing cycles for the 512KB cache case from an average of about 70% for a purely analytical model to an average of 14.5% for the hybrid MESH model. The 8KB cache average percent error is also decreased from 44% to 18%. Note that even though the MESH hybrid model raises the level of accuracy compared to the fully analytical model, the runtime

# of Procs.	FFT 512KB		FFT 8KB	
	MESH	ISS	MESH	ISS
32	0.12	12.29	0.13	13.08
16	0.09	10.32	0.08	11.16
8	0.06	9.95	0.07	10.43
4	0.07	10.77	0.07	10.32
2	0.08	12.03	0.08	10.93

Table 1. Simulation runtimes (in seconds) for the SPLASH-2 FFT Benchmark

of the MESH simulation is at least 100 times faster than a corresponding instruction set accurate simulation (Table 1). The MESH performance advantage would especially be evident with large and more complex models where the prohibitively large ISS simulation and model generation times would hamper rapid design exploration.

5.2 PHM SoC

Piecewise application of analytical models is especially useful when architectures or workloads are heterogeneous. Interleaving of applications on PHM systems generates irregular regions of high or low contention for shared resources depending on which applications are currently executing. Additionally, data dependencies or user interactions may result in available regions between execution of applications on SoCs, further unbalancing the usage of system-wide shared resources. Due to these reasons, purely analytical approaches, which are good at estimating performance of homogeneous multiprocessor systems running balanced workloads, are not well suited for PHM SoCs.

To demonstrate our approach's utility for PHM applications, we developed a PHM ISS for a shared bus 2 processor system [9], using the ARM and Renesas M32R processor simulators freely available in the GNU GDB distribution. To provide applications representative of future SoC workloads, we extracted several application kernels from the MiBench [6] benchmark suite. Results from the ISS were compared with our MESH model and with the purely analytical model as the bus access time was varied. We used the same Chen-Lin model from the first example, modified to work with 2 processors. Within the MiBench suite, we extracted several kernels from GSM encoding (telecomm), blowfish encryption (security), and mp3 encoding (multimedia) that are representative of their respective applications' behaviors. Unlike the SPLASH-2 FFT application used earlier, all these kernels have uniform levels of shared resource accesses across their runtimes, making purely analytical approaches accurate when considering each kernel individually. However, when these kernels are sporadically executed in a random fashion on two heterogeneous processors mimicking data-dependent behavior, the resulting contention patterns are irregular and unpredictable.

To illustrate decreasing performance of purely analytical models with unbalanced shared resource access loads, we kept the first processor busy (only 6% idle) while applying a light load to the second processor (90% idle). The idle periods here are used to introduce an extreme case of unbalance, or burstiness in shared resource accesses. Similar

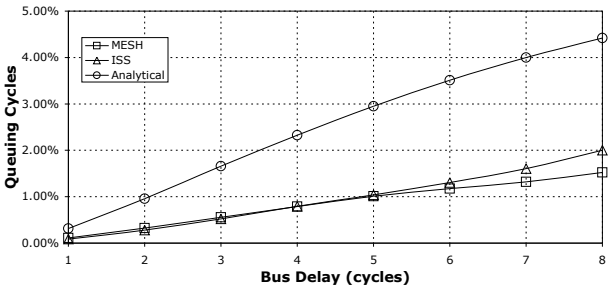

Figure 5. Queuing cycles predicted for various bus delays with second processor idle 90% of the time.

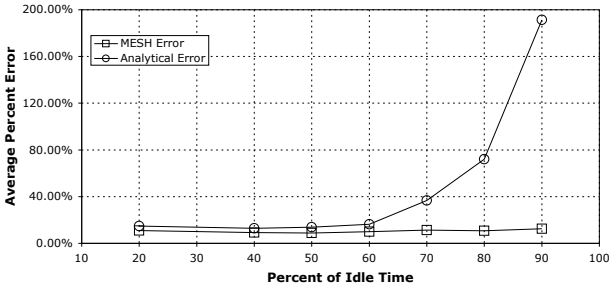

Figure 6. Degradation of the purely analytical model as shared resource access unbalance increases.

results can be gained with one application exhibiting much lower shared resource access rate than another application on the system. Figure 5 shows the percentage of queuing cycles given by each of the models as bus access time is varied. Because the analytical model is unable to recognize unbalanced workloads, it greatly overestimates the number of queuing cycles. To further quantify this behavior, Figure 6 shows the average error experienced by the MESH and the pure analytical models as the load on the second processor is varied. As can be seen, when application interactions exhibit relatively uniform shared resource access behavior, pure analytical models are acceptable. However, as one of the processors exhibits over 60% less shared resource accesses than the other, the purely analytical approach breaks down and is outperformed by the MESH hybrid model.

6. Conclusions

We present a hybrid approach for estimation of shared resource accesses in PHM SoC systems, combining elements of simulation and purely analytical modeling techniques. The MESH kernel implements this hybrid approach, evaluating analytical models in a piecewise fashion across the simulation runtime and applying time penalties to simulated regions. Through our examples, we show that the piecewise application of analytical models is superior when faced with systems exhibiting irregular shared accesses patterns, behavior commonly found in PHM systems. For these cases, simulation speed increased 100x versus instruction set simulation, with accuracy up to 80% better than the corresponding purely analytical models. The piecewise application presented in this paper is especially useful when analytical models assuming constant steady state system behavior are applied to systems with several distinct and unique modes of operation. The MESH kernel and the hybrid modeling technique present a first step towards a general method for abstractly but accurately modeling the shared resource contention within a PHM SoC.

7. Acknowledgments

This work was supported in part by ST Microelectronics, General Motors Collaborative Research Lab at Carnegie Mellon University, an NSF Graduate Research Fellowship, and the National Science Foundation under Grant 0103706. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] J. Bammi, W. Kruijtzer, L. Lavagno, E. Harcourt, and M. Lazarescu. Software performance estimation strategies in a system-level design tool. *CODES*, 2000.

[2] C. Chen and F. Lin. An Easy-to-Use Approach for Practical Bus-Based System Design. *IEEE Transactions on Computers*, August 1999.

[3] S. Gadde, J. Chase, and A. Vahadat. Coarse-grained network simulation for wide-area distributed systems. *Communication Networks and Distributed Systems Modeling and Simulation Conference*, 2002.

[4] S. Goldschmidt and H. Davis. Multiprocessor simulation and tracing using Tango. *Proceedings of the 1991 International Conference on Parallel Processing*, August 1991.

[5] R. Gupta and S. Liao. Using a programming language for digital system design. *IEEE Design and Test of Computers*, April-June 1997.

[6] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. MiBench: A free, commerically representative embedded benchmark suite. *IEEE 4th Annual Workshop on Workload Characterization*, December 2001.

[7] J. Davis II, M. Goel, C. Hylands, B. Kienhuis, E. Lee, et al. Overview of the Ptolemy Project. ERL Technical Report UCB/ERL No. M99/37, dept EECS, Berkeley, July 1999.

[8] E. Lee and A. Sangiovanni-Vincentelli. A Framework for Comparing Models of Computation. *IEEE Transactions on CAD, Vol 17, pp. 1217-1229*, December 1998.

[9] J. Paul, A. Bobrek, J. Nelson, J. Pieper, and D. Thomas. Schedulers as model-based design elements in programmable heterogeneous multiprocessors. *Design Automation Conference*, 2003.

[10] J. Paul and D. Thomas. A layered, codesign virtual machine approach to modeling computer sytems. *DATE*, 2002.

[11] K. Richter, M. Jersak, and R. Ernst. A Formal Approach to MpSoC Performance Verification. *IEEE Computer*, Volume 36, Number: 4, April 2003.

[12] S. Woo, M. Ohara, E. Torrie, J. Sing, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. *International Symposium on Computer Architecture*, June 1995.

[13] B. Zeigler, H. Praehofer, and T. Kim. *Theory of Modeling and Simulation.* Academic Press, 2000.