High Level Cache Simulation for Heterogeneous Multiprocessors

Joshua J. Pieper[1], Alain Mellan[2], JoAnn M. Paul[1], Donald E. Thomas[1], Faraydon Karim[2]

[1]Carnegie Mellon University
[jpieper, jpaul, thomas]@ece.cmu.edu

[2]STMicroelectronics
[alain.mellan, faraydon.karim]@st.com

ABSTRACT

As multiprocessor systems-on-chip become a reality, performance modeling becomes a challenge. To quickly evaluate many architectures, some type of high-level simulation is required, including high-level cache simulation. We propose to perform this cache simulation by defining a metric to represent memory behavior independently of cache structure and back-annotate this into the original application. While the annotation phase is complex, requiring time comparable to normal address trace based simulation, it need only be performed once per application set and thus enables simulation to be sped up by a factor of 20 to 50 over trace based simulation. This is important for embedded systems, as software is often evaluated against many input sets and many architectures. Our results show the technique is accurate to within 20% of miss rate for uniprocessors and was able to reduce the die area of a multiprocessor chip by a projected 14% over a naive design by accurately sizing caches for each processor.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Modeling Techniques

General Terms

Performance, Design

1. Introduction

There is a growing consensus that heterogeneous multiprocessing on a single chip will become more prevalent. For these chips, it is unlikely that all the processing elements will use a common, or homogeneous cache structure. Indeed, one recently proposed architecture, the Hyperprocessor framework [9], enables integration of many heterogeneous programmable components. Each Hyperprocessor instance may contain hundreds of interacting processors, caches, memories, I/O devices, and custom hardware, all of which can be customized to the application. Cache structures are especially difficult to simulate at a high level [3] [11], with few current tools to accurately simulate cache structures above the instruction set level. We tackle this problem by developing a method for simulating cache structures at a high level quickly and accurately.

DAC 2004, June 7–11, 2004, San Diego, California, USA.

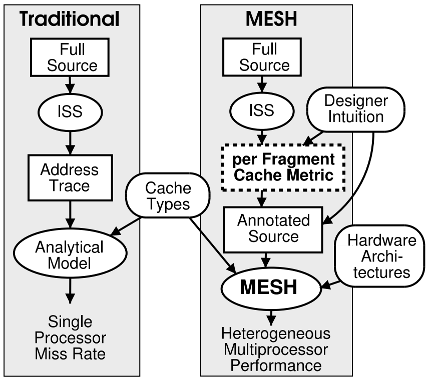

Figure 1: High-Level Cache Simulation Flows

Figure 1 shows the traditional method for high-level cache modeling and our newly proposed method, part of the MESH (Modeling Environment for Software and Hardware) framework [4]. In the traditional method, the modeled software is executed on an instruction set simulator (ISS), generating an address trace, which is then analyzed to determine the performance of the uniprocessor system. The address trace contains no information regarding potential parallelization, or data-dependencies within the code. This limits it to modeling uniprocessor non data-dependent systems. In systems that have data dependent execution, data dependent parallelization, or multiple hardware architectures to explore, the address trace will have to be generated for every possible input and architecture combination.

In contrast, we propose to use an ISS or designer intuition to generate short, concise metrics that describe the memory behavior of individual program fragments in the concurrent application set. These metrics are annotated into the original source code, then this source code and the hardware architecture are executed in the MESH high level performance simulation framework. Since the cache behavior for each program fragment is represented, the cache structure can be an input into the simulator for each processor in the system. The simulator can then find the cache behavior for each processor when executing each application fragment. Using this information, it can determine the interactions between concurrently executing software and discover the net performance of the heterogeneous processor system.

The proposed per fragment metric, highlighted in the figure, is a derivative of stack distance histograms. The stack distance histogram has many desirable properties for embedded system simulation, but requires too much data per fragment for back-annotation. Our contributions are two techniques that alleviate this limitation, logarithmic binning and adaptive average compression. Actual simulation time is very fast, with most of the cache modeling work occurring during a one-time annotation phase, resulting in a 20 to 50 times simulation speedup over trace based approaches. This is important for embedded systems, as software is often evaluated against many input sets and many architectures for which the one-time annotation cost is amortized over many simulations. Later in our example section, we show how this ability allowed MESH to reduce the total chip area needed for a multiprocessor system by a projected 14% by identifying which caches needed to be large and which did not, all with a reduced detail model.

2. The Hyperprocessor Framework

We begin by discussing the Hyperprocessor framework as its heterogeneous nature motivated this work. The Hyperprocessor is an attempt to streamline the design of modern heterogeneous Systems-on-Chip (SoC), by providing the designer with a simple, abstract view of the system (i.e. the programming model) and the technology to build an architecture that facilitates the mapping of the programming model.

The Hyperprocessor[9] manages task-level parallelism, where *tasks* are equivalent to actors in coarse-grain dataflow models. The top-level data- and control-flow is expressed in a high-level sequential program, very similar to assembly language. Each "instruction," or task, is a block of code, or a function performed by a hardware block. The data dependencies of the tasks are expressed through registers. This enables concurrent scheduling and, unlike a dataflow model, allows complex control-flow constructs to be implemented with much fewer resources. The result is a clean, 2-level hierarchical programming model that has the potential to blend hardware and software blocks seamlessly, and that raises the abstraction level for the SoC programmer, creating a new chip-level programmer's view.

The programming model is an abstraction of the underlying architecture, which uses principles very similar to a superscalar micro-architecture. At the task-level, a Control Processor fetches and decodes the task-level "instructions," and then performs renaming within the limits of the available renaming registers. The renaming eliminates false dependencies, giving more opportunities to increase the coarse-grain parallelism. A Task Dispatcher checks the data availability and dispatches eligible tasks to the Processing Units according to scheduling policies. A Universal Register File implements the data exchange between the processing units.

Each slave processor may be unique in both its microarchitecture and its interface to a memory subsystem, including caches. When exploring Hyperprocessor architectures at a high level using current techniques, the software must be profiled for every possible cache configuration to obtain the number of cache misses. Our technique reduces the amount of profiling that must be done before design exploration starts and reduces the amount of storage necessary per software application during the exploration process.

3. Prior Work

Several approaches have been proposed to represent the cache behavior of a program above the trace level. The simplest of these is the cache-miss rate or average cache misses per access. While simple and easy to use, this metric is simply an average performance value, closely tied to a specific cache architecture and requires re-profiling for every configuration considered. More advanced techniques attempt to reduce the dependence of the metric on the cache architecture.

3.1 High-Level Cache Modeling

In [11] both the program behavior and the cache structure are represented by 3D surfaces created by analyzing traces. The surface is a measure of both spatial and temporal locality. These surfaces contain too much data to be easily understandable by the designer and their accuracy is very limited. Cache Miss Equations [6] are another technique to represent the cache access patterns of software. Potential cache misses are represented as a set of linear equations whose solutions describe exactly when and where misses occur. When applied to this problem, their inability to handle data-dependent computation and the large amount of data needed prevent them from being widely used.

Many simplified analytical forms have been derived that can describe memory patterns [1]. In general, these formulas are very promising as metrics in our simulation methodology. Access patterns are distilled into a few numbers that represent key memory behavior characteristics. However, they are usually designed to model a single cache, and can be difficult to extend to multiple levels of cache hierarchy.

Stack distance histograms are another approach to modeling memory behavior originally used to quickly simulate many caches from a given address trace. They describe how many unique references occur in between two references to the same memory location. Using this information, a stack histogram will produce identical results as simulation for any fully associative cache and as shown in [8] can approximate any sized set-associative as well with reasonable accuracy. Their use of the inclusion property of caches makes them well suited to simulating multiple levels of hierarchy. They may in the future even be useful for simulating the differing register sets of microprocessors. Recent compiler techniques [3] have been able to generate approximate histograms, so this metric may be extendable to source level analysis. However, they do not account for cache line sizes, and a new histogram must be calculated every time the cache line size changes.

3.2 Stack Distance Histograms

Stack distance histograms are created by the stack simulation algorithm, which uses the inclusion property to quickly simulate many size caches with only one pass through an address trace. *Inclusion* [8] is a property of caches that is obeyed for two caches of size x and y where $x > y$ if every address that results in a hit in y is also a hit in x. It holds for caches that have the same block size, use the same set-mapping function and have certain restrictions placed on the set mapping and eviction functions (most common cache architectures meet these restrictions).

The stack simulation algorithm can be implemented as a single linked list, where the head of the list contains the most recently referenced cache line, the second item contains the second most recently referenced line, etc. When an address is encountered in the trace, a search is conducted through this list to see when the line was last referenced, then the item is removed and placed at the head. The item's previous distance is recorded and added to a histogram. Each bin in the histogram represents the number of cache lines that were referenced at that distance, and thus measures temporal locality in an address trace.

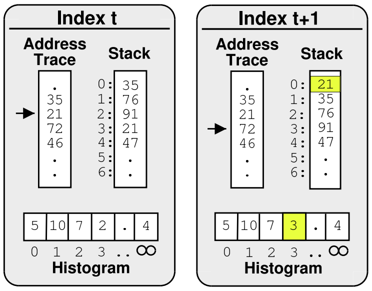

Figure 2: Stack Simulation Algorithm

For example, consider the address trace in Figure 2. It contains an entry for every cache line as it is accessed in program order. At time index t, the address trace is at the indicated position and the current stack state is as shown. The histogram contains the references that have been computed up until this point in time. Advancing to cache line at time index $t+1$, cache line 21 is found at position 3, then removed and re-inserted at the head of the list. Position 3 is the *distance* since last reference, so this value is recorded in the *distance histogram* at the bottom of the figure. The infinity bin is used for the first time a cache line appears in the address trace, and thus does not yet exist in the stack.

Using the histogram, any fully associative cache of size s can quickly be simulated by summing the bins from 0 to $s-1$. In this example, at index $t+1$, the number of hits for a cache that could hold three lines would be the sum of bins 0 through 2, or 22. An approximate formula for set-associative caches is derived in [8].

4. Proposed Metric

Stack distance histograms have many of the properties we require for our per-fragment metric. They can be evaluated quickly, handle multiple levels of hierarchy naturally, and can be used in automated way without requiring a full target toolchain. Their downfall is the large amount of space required to store the information for one fragment. There exists an entry in the histogram for every temporal distance referenced within a fragment, which on average equates to the working set size. For our largest example, Sphinx3 (to be described later), some fragment's histograms would need to contain 200,000 elements. This size is not feasible when the elements need to be back-annotated for possibly thousands of program fragments. Our contributions are two techniques, logarithmic bins and adaptive compression that reduce the size required to store the data inside a histogram with an acceptable loss of accuracy.

4.1 Logarithmic Bins

The first technique we use is calculating the histogram such that each bin sums all the distances from 2^n to 2^{n-1}. Since cache hardware is most efficiently implemented in powers of two, this seems like a reasonable approximation to make. For all fully associative caches of size 2^n, the number of misses will still be exact. For set-associative caches, this introduces more error as the set associative approximation must be made with truncated information.

The top of Figure 3 shows how the logarithmic bins relate to the full histogram for a sample fragment from one of our benchmarks. The original histogram has three distinct regions of memory references, one between 0 and 2, which correspond to stack distances between 0 bytes and 64 bytes, another between 5 and 6 (512 bytes through 1024 bytes in the original histogram), and a final region between 11 and 14 (32768 bytes through 524288 bytes). Stack distance is normally indicated in terms of cache lines, which for all of our experiments are 32 bytes. The logarithmic bins capture the basic features of these three regions, without retaining all of the detail. When simulating this program fragment, these logarithmic bins can easily be used to calculate a miss rate for many different cache sizes.

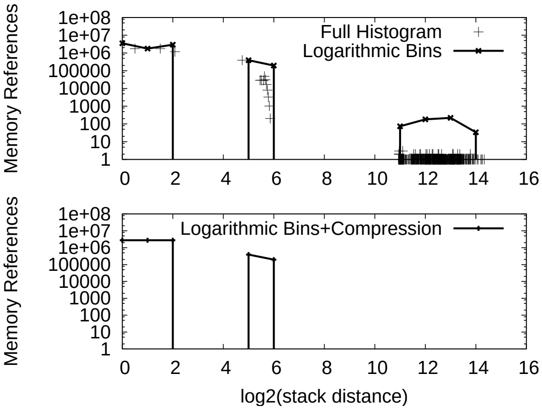

Figure 3: Stack Distance Histograms for Sample Sphinx3 Fragment

4.2 Compression

While the logarithmic binned histograms capture the relevant details of the base stack distance histogram, they still can require large amounts of space to store depending upon the workload size. For example, Sphinx3 has a workload of approximately 4 megabytes and requires 19 bins to store all of its histogram for many fragments. Each bin requires approximately 4 bytes to store, which equates to 76 bytes per annotation. For small fragment sizes, or finely instrumented software, this is clearly excessive. A method of reducing the amount of data further is required.

Our technique for compression, adaptive average compression, accomplishes this based on the observation that for most program fragments the histogram contains significant amount of empty space. Further, when there is non-empty space, the resulting distance values are often relatively flat and clustered together. To take advantage of these properties, our compression technique describes a program fragment's histogram as ranges of logarithmic bins and an associated average number of references per bin. For example, in the bottom of Figure 3, the histogram has been represented as 4 regions, bins 0-2, 3-4, 5, and 6. In each region the number of references per bin is assumed to be constant and equal to the average from the original logarithmic binned version.

We select a new set of regions for each program fragment based on what provides the least amount of modeling error. Most programs exhibit fewer references as the distance increases, which means the same absolute error will result in a greater percentage error for large cache sizes. We combat this by weighting our error calculation such that large stack distances are more likely to be allotted a region than small distances. In the fragment of Figure 3, bins 11-14, while containing some references, are represented as 0 in the compressed graph because even after the extra weight assigned to them they still contribute little to the memory behavior of the program fragment.

4.3 Assumptions

In developing this combination of stack distance histograms, logarithmic binning, and adaptive average compression, we must make several assumptions that limit the accuracy and retargetability of our technique. First, we will ignore the effects of compulsory misses and cache interference from multiple tasks executing on a single processor. These can greatly affect performance, but the former is unaffected by cache structure and while the latter is a difficult problem, it is negligible in the Hyperprocessor's task-based system and in other approaches that utilize many processors. Second, data-dependent behavior is averaged *within* each annotation region. For this to hold, the size of the annotation region must be small enough to capture system-level data dependencies. Third, we will only look at data accesses and ignore instruction accesses. For an embedded multiprocessor it is very likely that each processor will have a dedicated storage region for its instructions or an effective level 1 instruction cache. Fourth, reads and writes are treated equally.

Also, due to our selection of stack distance histograms, we must ignore variations in cache line size and assume that cache blocks map to sets independently. The first assumption is reasonable, often the cache line size is fixed by the processor, and even then it has only a second order effect on miss rate. We have used a 32 byte line size for all our experiments. The second assumption tends to cause the models to overestimate cache misses since real programs show high spatial locality. We will show with our later results that the accuracy loss is tolerable and in many cases unnoticeable. Finally, LRU replacement is assumed for all simulated caches.

5. Experiment Methodology

In this section we will first describe our modifications to the MESH framework to incorporate our new metric, next the methods we used to collect data for our experiments, and finally the specific benchmarks that we tested with.

5.1 MESH Simulation Framework

The MESH performance simulation framework facilitates modeling of heterogeneous systems on chips using reduced detail level models. It enables early design space exploration for performance in complex heterogeneous architectures where only part of the complete application or processor set is known. This is accomplished by decoupling the application, scheduling, and physical resources into separate interacting layers. These layers are analogous to the layers in the Hyperprocessor platform and are one reason why MESH was chosen to model this new architecture. Detailed descriptions of the methodology of MESH can be found in [2] [4].

In MESH, software threads are expressed by annotating arbitrary C code with *consume calls*. These consume calls create *annotation regions* between the calls, indicating the computational complexity of software within that fragment. Values associated with consume calls can be derived from techniques such as profiling, designer experience, or software libraries and do not directly relate to physical timing. The system's physical timing is determined by resolving the consume call complexity to the available physical resources in the system.

When applying the MESH annotations to cache design space exploration, or exploration of architectures with multiple differing caches, the main limitation is the dependence of computational complexity on the cache structure. For each cache and software combination that is considered, a separate computational complexity number must be derived. This requires that multiple copies of the source code be maintained, each with different valued annotations. No matter what method is used to derive the complexity number, it is difficult to maintain and store separate values for every cache structure under consideration.

To enable heterogeneous cache simulation, we extend our consume call structure to include memory behavior effects independent of cache structure, thus leaving the remainder of the structure to define the computing requirements of software. At simulation time, the memory system performance is calculated using the proposed metric and cache parameters such as size and associativity.

5.2 Experiment Design/Benchmarks

For our experiments, we use as a baseline a full address traces taken from an instruction set simulator (ISS). The ISS was instrumented to write the current address to a file every time a memory location was read or written. The exact *miss ratio*, or ratio of cache misses to memory references, was determined by running a full cache simulator with these traces. The simulator can be configured to emulate any combination of cache size, set size, and block size. For each combination, a separate run of the cache simulator was used to determine the miss ratio for that selection of address trace and cache parameters.

When analyzing the proposed metric, we used two different methods depending upon the aspect of the metric being tested. In the first case, we calculated a single metric for the entire application run, then used our model with the single metric and cache parameters to determine the miss ratio. In the second case, the metric is calculated for each program fragment, which are then modeled against a cache, and the resulting miss counts are added to obtain a total miss ratio. The first type of test allowed us to determine the accuracy of single metrics, while the second version is more detailed and shows the effects of local truncation errors averaged over entire application runs. To compare the accuracy of the metrics, we always measure the percentage error in the miss rates from that calculated by the proposed metric model to that of the trace simulator.

We employ several embedded benchmarks to evaluate our approach. The MiBench [7] suite is used as well as the Sphinx3 [10] speech recognition engine ported to run on the Hyperprocessor framework. From MiBench we use GSM, a cell phone audio compression algorithm and blowfish, a stream encryption cipher. They both have a small working set and exhibit regular data access patterns. Sphinx3 in contrast, has a very large working set and a data-dependent amount of dynamic parallelism. When annotating these benchmarks, we delimited fragments such that every function call and return would cause a new fragment to be created. Thus the granularity is somewhat higher than basic-block, but lower than function call level. However, the cache metric itself can be applied to any size annotation region.

6. Results

We first verify that each of our individual techniques, logarithmic binning and adaptive compression, are justifiable using directed experiments. After this, we then show that the techniques are applicable to larger systems, including single processors and finally heterogeneous multiprocessors.

Table 2: Compression Accuracy Comparison

		Block Average				Adaptive Average		
		w=2	w=3	w=4	w=5	n=5	n=4	n=3
Full	Sphinx3	11.7%	19.7%	16.7%	27.5%	6.2%	8.3%	12.8%
	Blowfish	10.8%	18.1%	17.3%	16.8%	8.7%	13.1%	14.5%
	GSM	18.2%	35.0%	26.4%	44.7%	13.8%	15.5%	34.0%
4-Way	Sphinx3	15.9%	18.7%	20.7%	32.0%	18.2%	19.6%	19.6%
	Blowfish	16.7%	7.1%	28.3%	19.1%	11.9%	13.7%	23.4%
	GSM	15.3%	41.3%	48.9%	58.6%	18.3%	23.1%	36.7%
Direct-Mapped	Sphinx3	14.3%	14.9%	20.5%	25.1%	12.7%	13.3%	13.2%
	Blowfish	3.5%	5.2%	7.8%	10.4%	2.6%	3.3%	8.6%
	GSM	32.3%	35.0%	37.6%	44.6%	27.1%	27.2%	25.9%
	Bytes/Annotation	40	28	20	16	22	18	14
	Average Error	15.3%	21.0%	24.8%	30.8%	13.3%	15.2%	20.9%

Table 1: Set-Associative Accuracy Comparison for Logarithmic Binning

	Average Error	
Experiment	Full Histogram	Logarithmic Binning
Sphinx3 Direct-Mapped	8.9%	12.7%
Sphinx3 2-Way	6.4%	11.1%
Sphinx3 4-Way	4.0%	8.6%
Sphinx3 8-Way	2.5%	6.3%
Blowfish Direct-Mapped	21.1%	16.1%
Blowfish 2-Way	21.7%	14.5%
Blowfish 4-Way	21.1%	11.6%

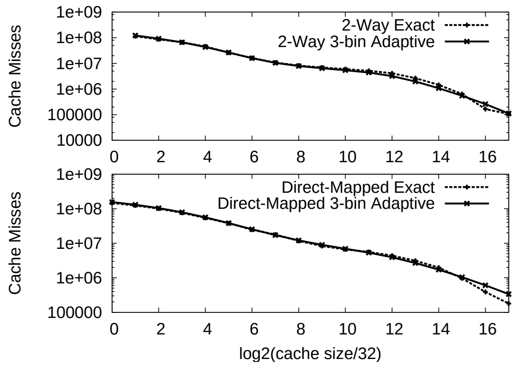

Figure 4: Miss Rate vs. Cache Size for Sphinx3

6.1 Metric Validation

First, we verify the validity of logarithmic binning as applied to stack distance histograms. We compare the accuracy of the logarithmic binning technique to the accuracy of the base stack distance histograms using our first experimental technique, whole program metric characterization. In Table 1, error was averaged for all power of two cache sizes less than the working set size. The average error after logarithmic binning is very similar to that of the original histogram, with some benchmarks showing slightly more, and some benchmarks showing slightly less error. Since the entirety of the error in the original stack distance approach is due to the set mapping independence assumption, it is reasonable to conclude that this is the source of the majority of the error after logarithmic binning as well.

To verify the adaptive average compression we compare it and a simple block average to trace based results. In the simple block average there is one fixed region size for all program fragments. When describing these techniques for the fixed block average, w is the width of each region and inversely proportional to the number of regions. In the adaptive case, n is the number of regions used per fragment. Using a similar methodology as when evaluating the logarithmic binning, we performed experiments comparing the accuracy of these approaches. The major difference between the two experiments is that now we use our second experimental technique and perform the binning and compression for every annotation region in the program, then sum the results, as opposed to using one histogram for the program as a whole. This increases the average error, but shows the actual results that a full uniprocessor simulation of each benchmark would. Table 2 gives our results for various benchmarks and cache configurations. The adaptive block averages perform significantly better than simple block averages and also require the fewest number of bytes to store. Large direct-mapped caches are more inaccurate. We address this issue in section 6.2.

6.2 Single Processor Results

After validating the metrics in isolation, we now perform simulation experiments of entire systems to determine all the interactions between metric error, truncation error, and application data dependencies. In Table 3 we ran each of the back-annotated benchmarks on a single processor ARM system for an entire input data-set. The total number of cache misses from the trace simulation and the model's prediction are shown, along with the percentage error. Figure 4 shows the number of predicted misses as cache size is varied for the Sphinx3 benchmark. The most error is shown for large cache sizes in the direct-mapped cache, where our model predicted twice as many misses as actually occurred. This error is a result of the stack histogram approach itself, and is not a result of our proposed compression techniques. We expect this to be minimal source of error in real systems, as data caches are rarely direct mapped.

To compare simulation complexity, the trace simulation, annotation calculation, and metric simulation CPU times are shown in Table 4. These show that the stack histogram plus compression metric performs more work at annotation time, but then enables very fast simulation. The long annotation time for sphinx is attributable to its large working set; stack histogram calculation complexity is proportional to both the trace length and working set size. In all examples, the proposed metric simulation time is more than an order of magnitude less than full trace simulation. This is important for embedded systems, as software is often evaluated against many input sets and many architectures; the one-time annotation cost is amortized over many simulations.

6.3 Multiprocessor System

Our multiprocessor system is the Sphinx3 benchmark ported to the Hyperprocessor architecture. The particular Hyperprocessor configuration we simulated is a bus-based shared

Table 3: Single Processor Cache Misses x 10^6

Benchmark	Cache	Exact	5-bin Adaptive	Error
Sphinx3	Direct 8k	11.7	11.7	0.7%
	Direct 64k	5.58	5.55	0.7%
	4-way 8k	7.58	7.10	6.3%
	4-way 64k	5.10	4.60	9.8%
GSM	Direct 512b	16.1	27.2	68.7%
	Direct 2k	1.73	7.58	337.0%
	4-way 512b	3.31	3.15	4.9%
	4-way 2k	0.808	0.519	35.8%
Blowfish	Direct 1k	25.9	25.1	3.3%
	Direct 4k	4.27	10.6	148.8%
	4-way 1k	24.2	21.0	13.1%
	4-way 4k	3.75	4.22	12.7%

Table 4: CPU Time (seconds) Comparison

	Trace-Based	Metric-Based	
	Simulation	Profiling + Annotation	Simulation
GSM	72.4	157.0	0.24
Blowfish	27.6	72.5	0.52
Sphinx3	48.8	8266.5	1.87

memory system where all of the slave processors execute the ARM instruction set. The MESH high level models use a penalty-based shared resource framework to model the interconnect [2]. In Figure 5 our experiments show how the application scales in the presence of different bus bandwidths. We ran the cycle accurate simulator and MESH with all processors using an 8k fully associative cache. The performance limit can be seen to change as the bus bandwidth (measured in processor cycles per cache line) varies. While MESH's absolute accuracy is only within 20% for this example, it does track the correct trends as both bus bandwidth and the number of processors is scaled. These trends allow the designer to discover and prune the design space early in the design process [4].

Next, we tried a configuration that could not be evaluated in the ISS, but was easy to implement in MESH—heterogeneous cache structures. Since caches can take up a large portion of the design budget of a processor, we decided to reduce the cache size of some of the processors. In the right column of Table 5 we started with a configuration consisting of all 8k fully associative caches. Then we reduced the cache size of all but one processor to 1k, these results are shown in the middle column. Finally, the configuration with all 1k caches is shown in the left column. We see that the architecture with only one large cache has performance that is almost identical to that of the architecture containing all large caches, indicating that the other caches were not the bottleneck in the system. For at least one embedded ARM processor, the cache system consumes 60% of the area budget of the entire processor core [5]. If the cache sizes are scaled proportionally between the described chip and this example, this experiment showed that 14% of the original chip area was not necessary. Since MESH was able to discover this during early design space exploration, possibly many iterations of expensive instruction set simulation can be eliminated.

7. Conclusions

We present a technique for representing memory access behavior of program fragments that allows high level simulation of cache structures by moving runtime simulation complexity into a one-time annotation phase. We evaluated the technique using single processor benchmarks from MiBench and a speech recognition example on the next generation Hyperprocessor framework. The experiments show that our technique has a simulation time one to two orders of magnitude faster than trace based approaches and can accurately model single processor cache configurations. In combination with a high level interconnect model it can accurately distinguish between heterogeneous multiprocessor architectures and quickly evaluate alternative designs. For our sample heterogeneous example, the technique was able to reduce the total chip area by a projected 14% through intelligently sizing the caches for each processor in the system.

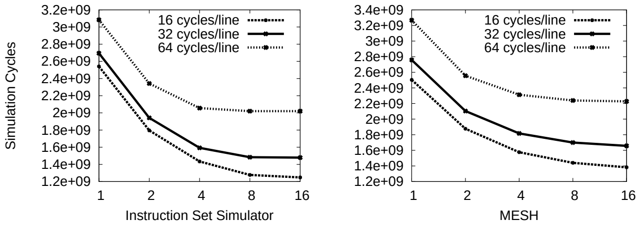

Figure 5: Scalability: Performance vs. # of Processors for 8k Fully Associative Cache

Table 5: Heterogeneous Cache Architecture Performance (Simulation Cycles x 10^9)

# of Processors	All 1k	One 8k	All 8k
1	3.021	2.758	2.758
2	2.355	2.110	2.103
4	2.068	1.824	1.817
8	1.947	1.705	1.700
16	1.899	1.659	1.658

8. Acknowledgments

This work was supported in part by ST Microelectronics, General Motors, 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.

9. REFERENCES

[1] A. Agarwal, M. Horowitz, and J. Hennessy. An Analytical Cache Model. *ACM Transactions on Computer Systems*, 7(2):184–215, 1989.

[2] A. Bobrek, J. Pieper, J. Nelson, J. Paul, and D. Thomas. Modeling Shared Resource Contention Using a Hybrid Simulation/Analytical Approach. *DATE*, 2004.

[3] C. Cascaval and D. Padua. Estimating Cache Misses and Locality Using Stack Distances. *Proceedings of the 17th Annual International Conference on Supercomputing*, pages 150–159, 2003.

[4] A. Cassidy, J. Paul, and D. Thomas. Layered, Multi-Threaded, High-Level Performance Design. *DATE*, 2003.

[5] L. Clark et al. A Scalable Performance 32b Microprocessor. *IEEE International Solid-State Circuits Conference*, pages 230–231, 2001.

[6] S. Ghosh et al. Cache Miss Equations: A Compiler Framework for Analyzing and Tuning Memory Behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, 1999.

[7] M. Guthaus et al. Mibench: A free, commerically representative embedded benchmark suite. *IEEE Workshop on Workload Characterization*, Dec. 2001.

[8] M. D. Hill and A. J. Smith. Evaluating Associativity in CPU Caches. *IEEE Transactions on Computers*, 38(12):1612–1630, 1989.

[9] F. Karim, A. Mellan, B. Stramm, T. Abdelrahman, and U. Aydonat. The Hyperprocessor: a Template Architecture for Embedded Multimedia Applications. *WASP*, 2003.

[10] K. Lee, H. Hon, and R. Reddy. An overview of the SPHINX speech recognition system. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 38:35–44, 1990.

[11] E. S. Sorenson and J. K. Flanagan. Cache Characterization Surfaces and Prediction Workload Miss Rates. *IEEE Workshop on Workload Characterization*, pages 129–139, December 2001.