Fast, Accurate Design Space Exploration of Embedded Systems Memory Configurations

Jason D. Hiser, Jack W. Davidson
Department of Computer Science
University of Virginia
Charlottesville, VA 22904
{hiser,jwd}@virginia.edu

David B. Whalley
Department of Computer Science
Flordia State University
Tallahasse, FL 32306
whalley@cs.fsu.edu

ABSTRACT

The memory hierarchy is often a critical component of an embedded system. An embedded system's memory hierarchy can have dramatic impact on the overall cost, performance, and power consumption of the system. Consequently, designers spend considerable time evaluating potential memory system designs. Unfortunately, the range of options in the memory hierarchy (e.g., number, size, and type of caches, on-chip SRAM, DRAM, EPROM, etc.) makes thorough exploration of the design space using typical simulation techniques infeasible. This paper describes a fast, accurate technique to estimate an application's average memory latency on a set of memory hierarchies. The technique is fast—two orders of magnitude faster than a full simulation. It is also accurate—extensive measurements show that 70% of the estimates were within 1 percentage point of the actual cycle count while over 99% of all estimates were within 10 percentage points of the actual cycle count. This fast, accurate technique provides the embedded system designer the ability to more fully explore the design space of potential memory hierarchies and select the one that best meets the system's design requirements.

Categories and Subject Descriptors

B.m [**Hardware**]: Miscellaneous, C.4 [**Performance of Systems**]: Performance attributes

General Terms

Algorithms, Measurement, Performance, Design, Experimentation.

Keywords

Memory Hierachy, Performance Estimation, Embedded Systems.

1. INTRODUCTION

Embedded system designers often must tailor a system's memory hierarchy to meet strict performance/cost constraints. Unfortunately, the wide range of options in the memory hierarchy (e.g., number, size, and type of caches, on-chip SRAM, DRAM, EPROM, etc.) leads to an exponential number of possibilities for memory hierarchy configuration. Changing the size of the cache may effect how much on-chip SRAM (also known as Scratchpad memory) is required. With hundreds or thousands of possibilities, it becomes prohibitively expensive to do a full simulation for each possible memory configuration. Such simulations typically take hours, and thousands of simulations may take weeks or months and are often impractical given the fast time-to-market constraints of many embedded systems. Worse yet, the program itself may evolve during the development of an ASIC processor. An extra memory reference in a critical loop could change what is the most appropriate configuration.

To produce designs that meet performance and cost constraints, system designers need fast, effective methods to determine the cost and relative performance of many different possible memory configurations. Figure 1 contains an example of a graph that a designer may use to evaluate the cost and performance of different memory configurations. The figure shows the cost (kilobytes of cache and on-chip SRAM) and the total memory latency for the *adpcm decode* benchmark when run for 80 different memory configurations. A designer can use this information to choose the memory configuration which best meets the design criteria. For example, a designer may know that the program needs to execute in less than five million memory cycles, and consume no more than 25 KB of cache and SRAM space. The figure shows that only configurations 60–76 meet those requirements.

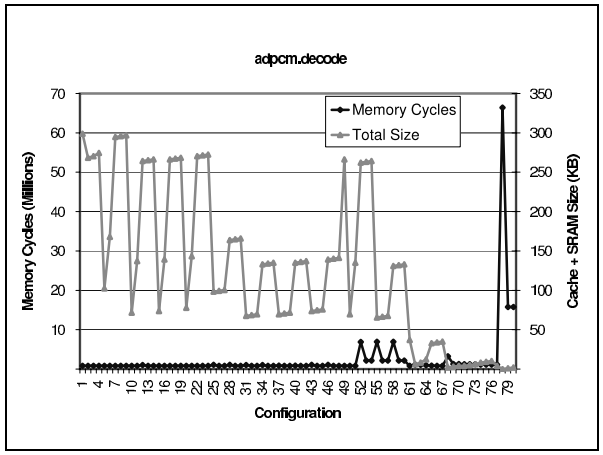


Figure 1: Memory access time and cost graph.

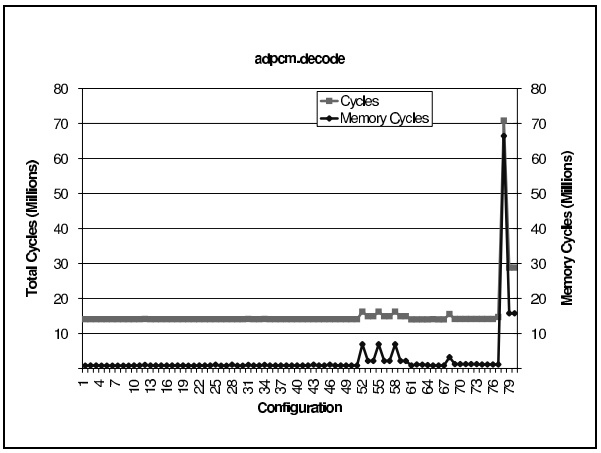
Note that Figure 1 plots the total time the program took to access memory for each configuration. The total memory access time is plotted because the goal is to estimate the performance of the

SAC'07, March 11-15, 2007, Seoul, Korea.

memory hierarchy, not the performance of the surrounding core. Furthermore, the performance of the chip is often tied directly to the memory system's performance.

The graph in Figure 2 plots actual total memory access time and total cycles to execute the program, again for the *adpcm decode* benchmark. This graph represents data from a two-issue out-of-order processor. See Section 4 for details of the processor and memory configurations used. The figure shows that the two measurements have the same trend, and thus processor performance is tied directly to the memory system performance.

Figure 2: Cycle count vs. memory latency.



This paper makes the following contributions. It describes the MCDSE process (memory configuration design space exploration) a *fast*, *accurate* process for exploring the design space of memory hierarchy configurations for embedded systems. MCDSE couples an existing algorithm for assigning program variables to *memory partitions* and a novel cache performance estimation technique to compute total memory access time curves for the memory configuration design space.

In this work we refer to a memory partition as a component of the memory hierarchy in which data is stored on a permanent basis. On-chip SRAM, and DRAM are examples of a memory partition. A cache, although part of the memory hierarchy is not considered a memory partition. Each memory partition is considered homogeneous and compiler-visible. This model is distinct from the models used by work that selects portions of main memory to convert to faster SRAM based on program access characteristics [2, 4].

The MCDSE process is fast—two orders of magnitude faster than using detailed simulation to evaluate a candidate memory hierarchy. MCDSE is also accurate—extensive measurements show that 70% of the estimates were within 1 percentage point of the actual cycle count and over 99% of all estimates were within 10 percentage points of the actual cycle count.

The MCDSE process is easily incorporated into existing compilation and simulation frameworks. Furthermore, the speed of the technique allows it to be employed in an iterative design environment where hundreds or thousands of possible memory hierarchies may need to be considered to find one that meets cost/performance/resource design constraints.

The remainder of the paper is organized as follows: Section 2 describes how the MCDSE process can be integrated into a design process. Section 3 describes how the total memory access time is estimated, while Section 4 discusses the experimental setup and results with a focus on the accuracy of the technique. Section 5 discusses related work in this area, and Section 6 summarizes our findings.

2. THE DESIGN PROCESS

Figure 3 illustrates how the memory system design space exploration technique can be used to accelerate the hardware design process.

The inputs to the hardware design process are the target application codes, any hardware restrictions, and the performance goals. After determining an appropriate processor core, the designer is faced with selecting the most appropriate memory system configuration.†

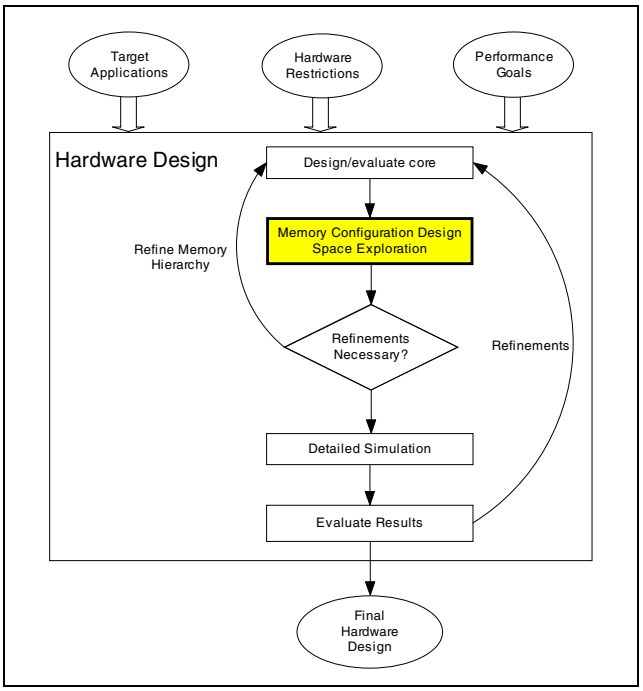


Figure 3: Design process.

The designer uses the MCDSE process (shaded box) to quickly eliminate memory configurations that do not meet the design constraints. For example, MCDSE produced the data in Figure 1 allowing the designer to focus on just 16 candidate memory configurations. From these few candidate memory configurations, the designer may select some or all to be evaluated using cycle-accurate simulation. Using the results of the detailed simulation, the designer may complete the design process or make changes to the core or memory system and repeat the process.

† We assume that the designer has some methodology for selecting an initial core that is appropriate for the application. As the figure shows, the MCDSE process does support subsequent changes to the core architecture.

3. MEMORY CONFIGURATION DESIGN SPACE EXPLORATION

Figure 4 expands the shaded box of Figure 3. The first step in the MCDSE process is to collect a profile of the variable accesses in the program. The profile contains information about which variables (global, stack, and heap) are accessed and where in the target code they are accessed.

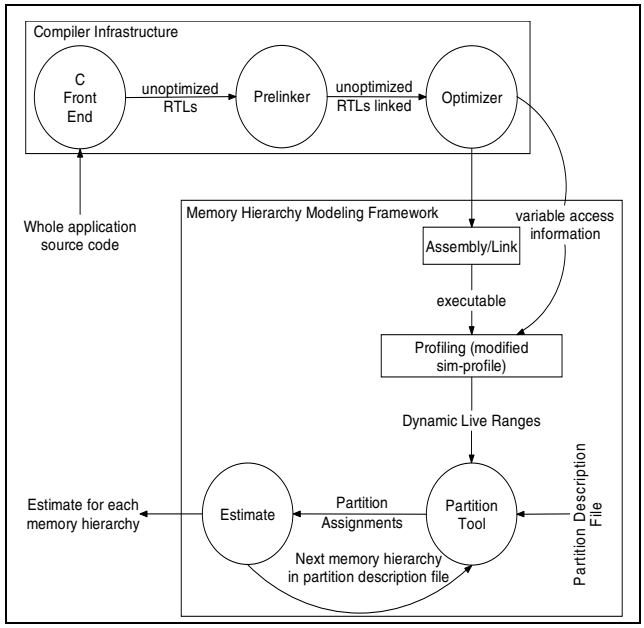


Figure 4: Block diagram of the compilation infrastructure.

To collect the profile, an existing compiler and simulator were modified. The compiler was modified to collect static information about variable accesses. Using interprocedural analysis, the compiler classifies variable accesses according to region (global, stack, and heap) and also records where in the program the variables were accessed. This information is made available to the simulator.

The simulator (based on SimpleScalar [7]) was modified to use the compiler collected information to monitor each load and store instruction to a particular variable and compute "dynamic" live ranges for each variable. A dynamic live range for a variable v is a triple (s, e, n), where s is the start cycle, e the end cycle, and n is the number of accesses to v between s and e. The dynamic live ranges are maintained in a hash table indexed by variable. When an address is accessed, the simulator determines which variable is accessed and updates the appropriate live range list.

We were able to use a fast functional simulator (sim-profile) because obtaining the profile does not require a cycle-accurate simulation. Furthermore, the profile information only needs to be collected once (unless the application or processor core is changed). Consequently, the simulator's runtime was sufficiently fast for our purposes (profile simulations took less than 25 minutes per benchmark to complete.)

The dynamic live range information, a description of the candidate memory hierarchies, and the program text itself are then used to assign variables to memory partitions. The choice of partitioning algorithm is arbitrary although the accuracy of MCDSE could be affected by the use of a poor algorithm. For the results reported in this paper, a previously published algorithm that has been shown to produce excellent results was used [11].

3.1 Estimating Cache Hit Rates

Once partition assignments are made, the data profile can be used to estimate the access patterns to each cache. In this work, the cache hit rate for each cache in the memory hierarchy is calculated first. To calculate hit rates, the estimation phase calculates an *effective cache size* (*ESize*) for each cache, *C*, in a top-down fashion, as follows:

$$ESize_C = Size_C \times \left(1 + \left(1 - \frac{\sum_{\forall v_1, v_2 \in P_c} tcf(v_1, v_2)}{Assoc_C \times Accesses_C}\right)\right)$$

where $tcf(v_1, v_2)$ is defined as the total conflict factor between variable v_1 and v_2 assigned to the same memory partition. We use the calculation of *tcf* as defined in our previous publication [11].

P_C is the partition cached by *C*. The sum of the *tcf*s is divided by the cache's associativity, C_{assoc}, to account for the reduced conflicts from higher associativity. The sum is also divided by $Accesses_C$, the accesses to cache *C*, to convert the summation into a percentage estimating the amount of conflict.

Next, using the data profile sorted by access count, the estimation phase counts the accesses to variables until the sum of the variables' size exceeds the effective cache size. Figure 5 gives an example. In the figure, the accesses to variables 1, 2, and 3 are included in the summation. Also, part of the accesses to variable 4 are included.† Call this sum *S*. The estimate for *C*'s hit rate is defined as:

$$HitRate_C = \frac{S}{LineSize_C \times Accesses_C} + 1 - \frac{1}{LineSize_C}$$

where $LineSize_C$ is *C*'s line size, and $Accesses_C$ is the number of accesses to *C*. The idea is that most of the *S* accesses will be hits and most of the remaining accesses will be misses. However, we also want to take into account that while traversing an array, which is common in embedded codes, about $1/LineSize_C$ accesses will probably be misses.

Once $HitRate_C$ is calculated, the estimation of misses can be passed down to lower level caches as cache accesses, and the calculation is repeated (omitting the variables cached by the upper level caches, such as Var 1–3 in the example) to calculate hit rates for lower level caches.

3.2 Estimating Average Access Times

Once the estimated hit rate of all caches have been computed, the estimated average access time (*EAAT*) for each cache is calculated.

† The estimation phase calculates the percent of the variable's size that fits within the cache, say X%. It then includes X% of the accesses in the sum.

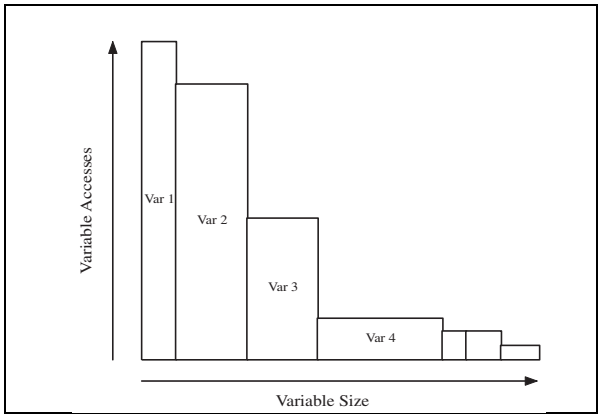


Figure 5: Cache hit rate visualization.

Since the access time of a first level cache depends on access times for second level caches, the calculation is done in a bottom-up fashion. The *EAAT* for a cache is:

$$EAAT_C = t_C \times HitRate_C + t_m \times (1 - HitRate_C)$$

where t_C is the time to access *C*, and t_m is the time to satisfy a miss (either the *EAAT* of a lower level cache, or by the access time of the backing partition.)

Next, once an *EAAT* has been calculated for each cache, the average access time for each partition can be calculated. $EAAT_P$ is the minimum of the access time to the partition or the minimum *EAAT* of the caches that satisfy requests for variables in partition *P*.

$$EAAT_P = \min\left\{\begin{matrix}\forall EAAT_C \mid C \text{ caches } P \\ AccessTime_P\end{matrix}\right\}$$

Lastly, the total access time is calculated:

$$TotalAccessTime = \sum_{\forall p \in partitions} Accesses_P \times EAAT_P$$

This calculation provides an estimate of how long the processor will spend satisfying memory requests if the program were run on the training input with the given memory configuration. Of course, we are interested in the time satisfying memory requests for the program run on actual data. Section 4.2 discusses the accuracy of the MCDSE estimation when compared to cycle-accurate simulations using real data.

4. EVALUATION

4.1 Experimental Setup

For the experiments reported in this paper, we used a compiler targeted to SimpleScalar PISA instruction set [5, 7]. PISA is a variant of the MIPS R4000 instruction set, commonly used in many embedded applications.

SimpleScalar was configured to use a two-issue out-of-order processor in all experiments [6]. The CPU has two integer execution units, and one floating-point execution unit, along with a memory unit for each partition. All on-chip caches and SRAM are modeled as having a one cycle latency, while an off-chip (level-2) cache is modeled using a 10 cycle latency. All caches were direct mapped caches. DRAM is modeled as having 100 cycle latency. All these parameters can be adjusted to reflect the operating parameters of the target system and memory components.

Table 1 lists the 80 memory hierarchies used to evaluate the accuracy of the MCDSE technique. Additional experiments and measurements using other configurations were performed and yield similar results. Due to space limitations this data cannot be presented.

The configurations range from memory hierarchies with 0–4kb of on-chip SRAM, 0–32kb on-chip cache, and 0-256kb of off-chip, L2 cache. Some configurations, such as configuration 50, may have 0kb of on-chip cache, but still access an off-chip cache. Note that different configurations have significantly different cache and SRAM sizes. Thus cycle times may be significantly different between configurations and two configurations may not be directly comparable. However, since the goal of the work is to estimate the cycle count accurately, it is not important to compare configurations. Instead, it is important that the estimates trend in the same fashion as the cycle-accurate measurements.

Table 2 describes the benchmarks that are used in this work, and also the inputs used for profiling and for the evaluation. We selected a benchmark set that had a wide range of qualities that made the benchmarks suitable for testing our algorithm. Some benchmarks, such as *dijkstra*, use significant amounts of dynamic memory allocation. Others, such as *pegwit* and *mpeg2.decode*, have large, complex code segments, which would make static analysis complicated. All of the benchmarks are long-running and most have been used in prior memory partitioning research [17, 3].

4.2 Experimental Results

To evaluate the accuracy of the MCDSE cache hit rate estimates, one could compare the estimates to those obtained via cycle-accurate simulation. However, some cache hit rates are more important than others. Consider Figure 6a. Since many more accesses are to the SRAM (indicated by the wide arrow between the SRAM and CPU), the hit rate of the cache is relatively unimportant. In Figure 6b however, the cache hit rate is very important as it strongly determines the performance of the system. Furthermore, it may not be necessary to accurately estimate the cache hit rates when it is possible to representatively estimate the change in cache hit rate caused by changes in partition assignments or cache configuration.

Figures 7–9 show the results of evaluating our estimation technique for each of the five benchmarks. Each graph plots the estimated memory cycles for the benchmark and the actual memory cycles used. The actual memory cycles used was obtained by running a cycle-accurate simulation using the evaluation input. Notice that the data are plotted using different scales. The data must be plotted using different scales because the estimation phase has no way to estimate the run time of the program on reference input, and the reference runs execute longer than the training runs.

The key observation is that the trends are the same. The figures show that the estimate curve tracks very closely to the actual curve. On 500Mhz UltraSparc-IIe's with 1GB of RAM the curves corresponding to the actual memory cycles took over 200 hours to cal-

Table 1: Component sizes (in kb) for configurations.

config num	SRAM	L1	L2	config num	SRAM	L1	L2	config num	SRAM	L1	L2	config num	SRAM	L1	L2
1	4	32	256	**21**	4	8	128	**41**	1	4	128	**61**	4	32	0
2	4	2	256	**22**	0	8	256	**42**	2	4	128	**62**	4	2	0
3	4	4	256	**23**	1	8	256	**43**	0	8	64	**63**	4	4	0
4	4	8	256	**24**	2	8	256	**44**	1	8	64	**64**	4	8	0
5	4	32	64	**25**	0	32	64	**45**	2	8	64	**65**	0	32	0
6	4	32	128	**26**	1	32	64	**46**	0	8	128	**66**	1	32	0
7	0	32	256	**27**	2	32	64	**47**	1	8	128	**67**	2	32	0
8	1	32	256	**28**	0	32	128	**48**	2	8	128	**68**	0	2	0
9	2	32	256	**29**	1	32	128	**49**	4	0	256	**69**	1	2	0
10	4	2	64	**30**	2	32	128	**50**	4	0	64	**70**	2	2	0
11	4	2	128	**31**	0	2	64	**51**	4	0	128	**71**	0	4	0
12	0	2	256	**32**	1	2	64	**52**	0	0	256	**72**	1	4	0
13	1	2	256	**33**	2	2	64	**53**	1	0	256	**73**	2	4	0
14	2	2	256	**34**	0	2	128	**54**	2	0	256	**74**	0	8	0
15	4	4	64	**35**	1	2	128	**55**	0	0	64	**75**	1	8	0
16	4	4	128	**36**	2	2	128	**56**	1	0	64	**76**	2	8	0
17	0	4	256	**37**	0	4	64	**57**	2	0	64	**77**	4	0	0
18	1	4	256	**38**	1	4	64	**58**	0	0	128	**78**	0	0	0
19	2	4	256	**39**	2	4	64	**59**	1	0	128	**79**	1	0	0
20	4	8	64	**40**	0	4	128	**60**	2	0	128	**80**	2	0	0

Table 2: Benchmark and input descriptions.

Benchmark	Description	Profile Input	Evaluation Input	Inst. Count	Source
CRC 32	Compute 32-bit CRC used as the frame check sequence in ADPCM	small.pcm—1 MB pcm file	large.pcm—26 MB pcm file	4.8M	[10]
dijkstra	Dijkstra's shortest path algorithm	input.dat—20 shortest paths	input.data—50 shortest paths	285M	[10]
adpcm.encode	Adaptive, Differential, Pulse-Code Modulation Speech Encoder	clinton.pcm	clinton.pcm	6.6M	[13]
pegwit	public key decryption and authentication	91k pgptest.plain, encoded	637k pgpref.plain, encoded	89M	[10]
mpeg2.decode	Converts a compressed bitstream into an ordered set of uncompressed pictures	test.m2v—8k m2v file	mei16v2.m2v—35k m2v file	159M	[10]

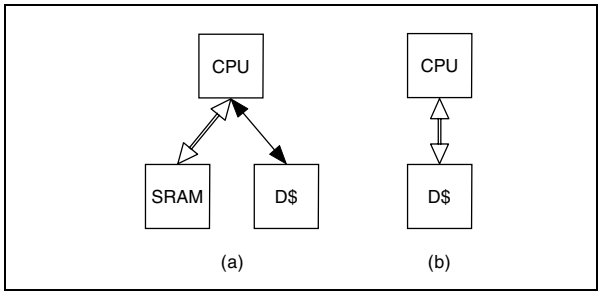


Figure 6: Example memory configurations.

culate while the estimate curves were produced in under two hours of elapsed time—a two order of magnitude reduction in elapsed time.

To measure how closely the estimated memory access time matches the actual memory access time, both the estimated values and the actual values were scaled so they were in the 0–1 range. The scaling effectively makes each point a percentage of the maximum value. On this adjusted data set, the mean absolute error (MAE) and root mean square error (RMSE) were calculated. The MAE and RMSE are the equivalent of the average and standard deviation for this type of data set.

Table 3 contains the error estimates. The percent errors average only 1.5%. The small RMSE indicates that the errors are relatively consistent, and few are off by more than the MAE. Figure 10 contains a histogram of the percentage errors. Each bar represents how many estimates deviate from the actual memory access time by more than 1, 3, 5, 10, and 20 percentage points, respectively. The average bar shows that over 70% of the data points deviate by less than 1 percentage point and 99% of the estimates deviate by less than 10 percentage points. The *CRC* benchmark does the best with all estimated memory cycles deviating by less than 3 percentage points.

Some points, however, have significant error. In particular, configuration 68 for the *dijkstra* and *pegwit* benchmarks shows errors of 11.4 percentage points and 8.4 percentage points respectively.

This increased error stems from configuration 68 itself and the dynamic nature of these benchmarks. Configuration 68 has no SRAM, no second-level cache, and a very small first-level cache. Since a program that has much dynamic memory allocation and large variables makes estimating conflict in the cache extremely difficult, the MCDSE estimation model has difficulty predicting the cache hit rate. Small errors in the estimated cache hit combined with a large discrepancy between the cost of a cache hit and miss (such as having no L2 cache) yield these larger errors in estimated memory access times.

An embedded system designer may be less interested in the amount of error and more interested in the *fidelity* of the memory

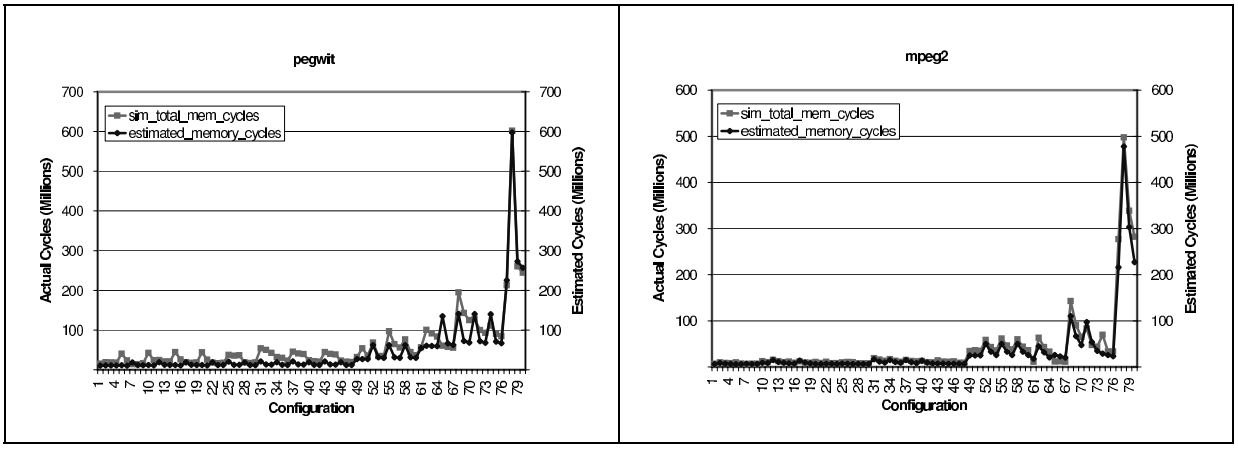


Figure 7: Estimated and actual memory cycles for *pegwit* and *mpeg2*.

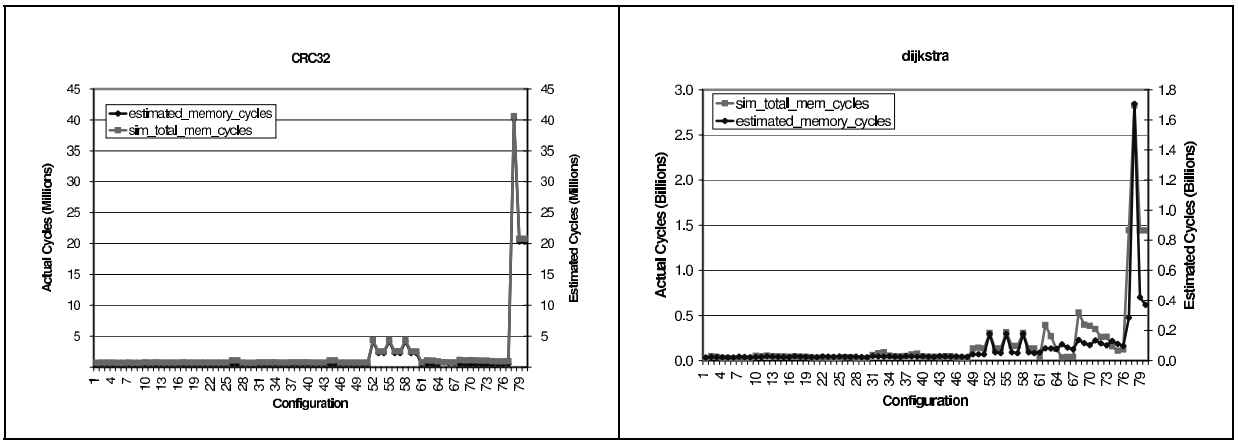


Figure 8: Estimated and actual memory cycles for *CRC* and *dijkstra*.

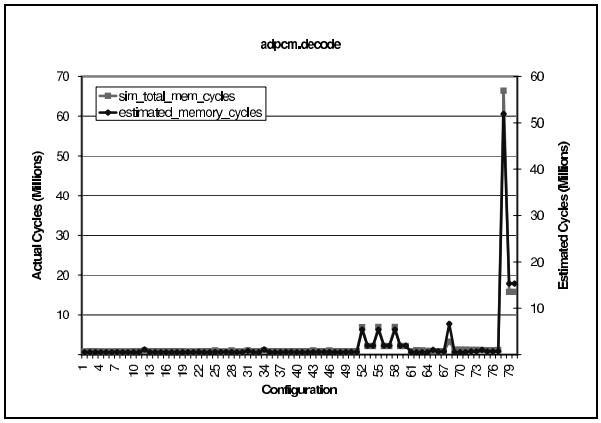


Figure 9: Estimated and actual memory cycles for *adpcm decode*.

performance estimates. The fidelity of the estimates is defined as the percentage of time the estimates of a memory configuration's performance can be used to select the memory configuration that actually provides the best performance.

Consider Figure 11a which shows a magnified view of a section of a graph plotting the design space estimates of memory cycles expended and the actual memory cycles expended as determined by a cycle-accurate simulator. Comparing the estimates and choosing the one with the lowest cycle count results in the selection of the memory configuration H_1 that, in fact, does provide the best performance (in terms of cycles). Figure 11b shows a similar graph where choosing the estimate with the lowest cycle count (E_2) results in the selection of the memory configuration H_4 that, in fact, does not provide the best performance (in terms of cycle count).

Table 3: MAE and RMSE.

Benchmark	MAE	RMSE
CRC32	0.27%	0.05%
dijkstra	2.66%	0.73%
pegwit	2.61%	0.41%
mpeg2.decode	1.39%	0.27%
adpcm.encode	0.45%	0.15%
Average	1.50%	0.30%

To measure the fidelity of the MCDSE estimates, a pairwise comparison of all estimates was performed to determine the number of times the comparison of two estimates would result in the selection of a memory configuration that, in fact, was not the memory configuration that provided the best actual performance. Thus, for n configurations, there are $n(n-1)/2$ comparisons. An estimator has 100% fidelity if the estimator, when used to compare two memory

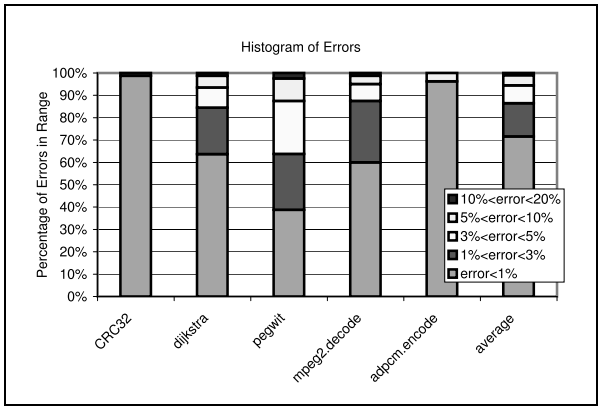


Figure 10: Error histogram.

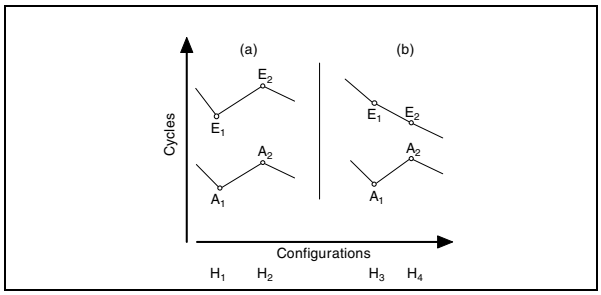


Figure 11: Example of correct and incorrect comparisons.

configurations, always selects the configuration that has the best actual performance.

Table 4 presents the results of this pairwise comparison of the 80 estimates for the five benchmarks. The table shows the breakdown of correct, incorrect, and total comparisons per benchmark in columns 2, 3, and 4, respectively. The fidelity ranges from 73% to 90% across the benchmarks. The average is 80.7%.

However, savvy system designers understand that estimates have some amount of inherent error. A designer may see that two memory configurations were estimated to perform very similarly. Knowing that there is some inherent error allows the designer to fall back to detailed simulations when two memory configurations are estimated to have similar performance. To determine how well this approach may work for a designer, the previous calculation was redone, but comparisons in which the estimates differ by less than the MAE for that benchmark were rejected. Table 4 shows that, on average, 94.9% of the time, a designer is able to select the best candidate memory hierarchy using this scheme. Column 5 shows the average for each benchmark.

5. RELATED WORK

Due to the importance of the problem addressed in this paper, there has been much research in this area. Jacob, et al. describe cache modes that address memory hierarchy design for generic workloads [12]. Such models are not applicable to ASIC-based systems because in an ASIC, the workload is fixed by the application and need not perform well on a wide range of workloads like a desktop machine. Other work focuses on statically predicting the cache behavior for each kernel loop of a program [21, 9]. Unfortunately, some benchmarks, *dijkstra* for example, cannot be analyzed by static techniques because of function calls, pointers, and dynamic allocation.

Benchmark	Correct Compares	Incorrect Compares	% Correct	Restricted % Correct
CRC	2,441	719	77.2	95.1
dijkstra	2,459	611	80.7	90.8
pegwit	2,611	549	82.6	96.6
mpeg2	2,827	333	89.5	97.1
adpcm	2,317	843	73.3	94.8
Average	2,549	611	80.7	94.9

Table 4: Per benchmark memory hierarchy comparisons.

Other work aims to solve the memory configuration evaluation problem in a different way, by speeding up simulation time. This can be done by simulating multiple caches in one simulation run, taking advantage of cache properties such as the inclusion property or how associativity affects caches [20, 22]. These techniques, however, still require many simulations to fully examine the spectrum of possible memory hierarchies and may still take extensive time to complete. Also, it is difficult to reason about some properties when multiple first-level caches are backed by lower-level caches (such as having first-level instruction and data caches backed by a level 2 cache.) Other work only addresses part of the problem, how to choose first-level cache parameters (size, associativity, line size, etc.) or how to choose the number of memory partitions assuming there is a first-level cache [18, 19, 14, 1]. Unfortunately, a single simulation run, no matter how many cache parameters are evaluated, cannot take into account varying SRAM size. Varying the partition size changes which variable accesses are serviced by a cache, and can significantly change the access patterns. Thus, to get a picture of how the entire memory hierarchy works, a chip designer may still need a very large number of detailed simulations.

Still other work aims to speed up simulation by taking samples of the memory trace and replaying just those samples to the memory subsystem [8, 23, 16, 15]. These techniques also yield accurate results. Although such techniques can be significantly faster than full simulation, sampling a small fraction of memory references can still be expensive and replaying them for each candidate memory configuration would be more expensive than an approach that combines simulation with analytical modeling.

6. SUMMARY

The memory system is a critical component of many embedded systems. Increasingly designers are using partitioned memory architectures to meet cost/performance constraints. Unfortunately, the range of options (e.g., number, size, and type of caches, on-chip SRAM, DRAM, EPROM, etc.) complicates choosing an appropriate memory configuration.

To address this problem, this paper has presented an approach for quickly exploring the design space of possible memory configurations for an embedded system. The technique is fast and accurate. It is two orders of magnitude faster than detailed simulation. It is also accurate. For five commonly used embedded benchmarks, we found that the estimations show the same trends as the applica-

tion. The mean absolute error of the scaled data is 1.5% of the maximum estimate, on average. Furthermore, we see that over 70% of all estimates were within 1 percentage point of the actual, while over 99% of all estimates were within 10 percentage points.

7. ACKNOWLEDGEMENTS

This research was supported in part by the National Science Foundation under grants CNS–0551560, CNS–0524432, CNS–0305144, CNS–0072043, CCR–0208892, CCR–0312493, and CNS–0615085.

8. REFERENCES

[1] ABRAHAM, S., AND MAHLKE, S. Automatic and efficient evaluation of memory hierarchies for embedded systems. In *Proceedings of the 32th Annual International Symposium on Microarchitecture* (1999), pp. 114–125.

[2] ANGIOLINI, F., BENINI, L., AND CAPRARA, A. Polynomial-time algorithm for on-chip scratchpad memory partitioning. In *CASES '03: Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems* (New York, NY, USA, 2003), ACM Press, pp. 318–326.

[3] AVISSAR, O., AND BARUA, R. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Transactions on Embedded Computing Systems 1*, 1 (2002), 6–26.

[4] BENINI, L., MACII, A., AND PONCINO, M. A recursive algorithm for low-power memory partitioning. In *ISLPED '00: Proceedings of the 2000 International Symposium on Low Power Electronics and Design* (New York, NY, USA, 2000), ACM Press, pp. 78–83.

[5] BENITEZ, M. E., AND DAVIDSON, J. W. A portable global optimizer and linker. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation* (1988), pp. 329–338.

[6] BROOKS, D., TIWARI, V., AND MARTONOSI, M. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture* (2000), ACM Press, pp. 83–94.

[7] BURGER, D., AUSTIN, T. M., AND BENNETT, S. Evaluating future microprocessors: The SimpleScalar tool set. Technical Report CS-TR-1996-1308, University of Wisconsin, Madison, 1996.

[8] CONTE, T. M., HIRSCH, M. A., AND HWU, W.-M. W. Combining trace sampling with single pass methods for efficient cache simulation. *IEEE Transactions on Computers 47*, 6 (1998), 714–720.

[9] GHOSH, S., MARTONOSI, M., AND MALIK, S. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems 21*, 4 (1999), 703–746.

[10] GUTHAUS, M. R., RINGENBERG, J. S., ERNST, D., AUSTIN, T. M., MUDGE, T., AND BROWN, R. B. Mibench: A free, commercially representative embedded benchmark suite.

[11] HISER, J. D., AND DAVIDSON, J. W. EMBARC: An efficient memory bank assignment algorithm for retargetable compilers. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools* (2004), ACM Press, pp. 182–191.

[12] JACOB, B. L., CHEN, P. M., SILVERMAN, S. R., AND MUDGE, T. N. An analytical model for designing memory hierarchies. *IEEE Transactions on Computers 45*, 10 (October 1996), 1180–1194.

[13] LEE, C., AND STOODLEY, M. University of toronto dsp benchmark suite. World Wide Web, *http://www.eecg.toronto.edu/extasciitildecorinna/*.

[14] LIN, H., AND WOLF, W. Co-design of interleaved memory systems. In *Proceedings of the 8th International Workshop on Hardware/Software Codesign* (2000), ACM Press, pp. 46–50.

[15] LIU, L., AND PEIR, J. Cache sampling by sets. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on 1*, 2 (1993), 98–105.

[16] PATEL, S. L. J. H., AND IYER, R. K. Accurate low-cost methods for performance evaluation of cache memory systems. *IEEE Transactions on Computers 37*, 11 (1988), 1325–1336.

[17] SAGHIR, M. A. R., CHOW, P., AND LEE, C. G. Exploiting dual data-memory banks in digital signal processors. In *Proceedings of the 2006 International Conference on Architectural Support for Programming Languages and Operating Systems* (1996), pp. 234–243.

[18] SHIUE, W.-T., AND CHAKRABARTI, C. Memory exploration for low power, embedded systems. In *Proceedings of the 36th ACM/IEEE Conference on Design Automation* (1999), ACM Press, pp. 140–145.

[19] SINGH, J. P., STONE, H. S., AND THIEBAUT, D. F. A model of workloads and its use in miss-rate prediction for fully associative caches. *IEEE Transactions on Computers 41*, 7 (July 1992), 811–815.

[20] WANG, W.-H., AND BAER, J.-L. Efficient trace-driven simulation methods for cache performance analysis. *ACM Transactions on Computer Systems 9*, 3 (1991), 222–241.

[21] WOLF, M. E., MAYDAN, D. E., AND CHEN, D.-K. Combining loop transformations considering caches and scheduling. In *MICRO 29: Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture* (Washington, DC, USA, 1996), IEEE Computer Society, pp. 274–286.

[22] WU, Z., AND WOLF, W. Iterative cache simulation of embedded cpus with trace stripping. In *Proceedings of the 7th International Workshop on Hardware/Software Codesign* (1999), ACM Press, pp. 95–99.

[23] XU, R., AND LI, Z. A sample-based cache mapping scheme. In *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (New York, NY, USA, 2005), ACM Press, pp. 166–174.