Analytical Design Space Exploration of Caches for Embedded Systems

Arijit Ghosh and Tony Givargis
Department of Information and Computer Science
Center for Embedded Computer Systems
University of California, Irvine, CA 92697
{arijitg,givargis}@ics.uci.edu

Abstract

The increasing use of microprocessor cores in embedded systems, as well as mobile and portable devices, creates an opportunity for customizing the cache subsystem for improved performance. Traditionally, a design-simulate-analyze methodology is used to achieve desired cache performance. Here, to bootstrap the process, arbitrary cache parameters are selected, the cache sub-system is simulated using a cache simulator, based on performance results, cache parameters are tuned, and the process is repeated until an acceptable design is obtained. Since the cache design space is typically very large, the traditional approach often requires a very long time to converge. In the proposed approach, we outline an efficient algorithm that directly computes cache parameters satisfying the desired performance. We demonstrate the feasibility of our algorithm by applying it to a large number of embedded system benchmarks.

Keywords

Cache Optimization, Core-Based Design, Design Space Exploration, System-on-a-Chip

1. Introduction

The growing demand for embedded computing platforms, mobile systems, general-purpose handheld devices, and dedicated servers coupled with shrinking time-to-market windows are leading to new core based system-on-a-chip (SOC) architectures [1][2][3]. Specifically, microprocessor cores (a.k.a., embedded processors) are playing an increasing role in such systems' design [4][5][6]. This is primarily due to the fact that microprocessors are easy to program using well evolved programming languages and compiler tool chains, provide high degree of functional flexibility, allow for short product design cycles, and ultimately result in low engineering and unit costs. However, due to continued increase in system complexity of these systems and devices, the performance of such embedded processors is becoming a vital design concern.

The use of data and instruction caches has been a major factor in improving processing speed of today's microprocessors. Generally, a well-tuned cache hierarchy and organization would eliminate the time overhead of fetching instruction and data words from the main memory, which in most cases resides off-chip and requires power costly communication over the system bus that crosses chip boundaries.

Particularly, in embedded, mobile, and handheld systems, optimizing of the processor cache hierarchy has received a lot of attention from the research community [7][8][9]. This is in part due to the large performance gained by tuning caches to the application set of these systems. The kinds of cache parameters explored by researchers include deciding the size of a cache line (a.k.a., cache block), selecting the degree of associativity, adjusting the total cache size, and selecting appropriate control policies such as write-back and replacement procedures. These techniques, typically, improve cache performance, in terms of miss reduction, at the expense of silicon area, clock latency, or energy.

Traditionally, a design-simulate-analyze methodology is used to achieve optimal cache performance [10][11][12][13]. In one approach, all possible cache configurations are exhaustively simulated, using a cache simulator, to find the optimal solution. When the design space is too large, an iterative heuristic is used instead. Here, to bootstrap the process, arbitrary cache parameters are selected, the cache sub-system is simulated using a cache simulator, cache parameters are tuned based on performance results, and the process is repeated until an acceptable design is obtained.

Central to the design-simulate-analyze methodology is the ability to quickly simulate the cache. Specifically, cache simulation takes as input a trace of memory references generated by the application. In some of the efforts, speedup is achieved by stripping the original trace to a provably identical (from a performance point of view) but shorter trace [14][15]. In some of the other efforts, one-pass techniques are used in which numerous cache configurations are evaluated simultaneously during a single simulation run [16][17]. While these techniques reduce the time taken to obtain cache performance metrics for a given cache configuration, they do not solve the problem of design space exploration in general. This is primarily due to the fact that the cache design space is too large. Figure 1(a) depicts the traditional approach to cache design space exploration.

Our approach uses an analytical model of the cache combined with an algorithm to directly and efficiently

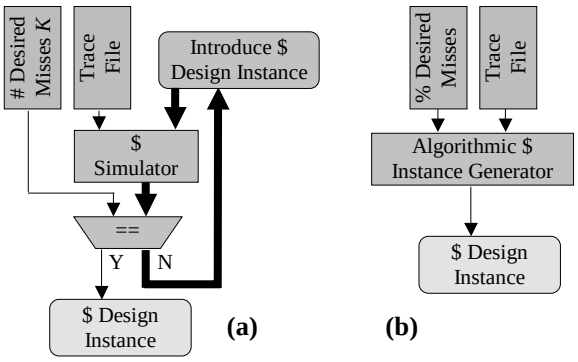

Figure 1: Design space exploration: (a) traditional approach, (b) proposed approach.

compute a cache configuration meeting designers' performance constraints. Figure 1(b) depicts our proposed analytical approach to cache design space exploration. In our approach, we consider a design space that is formed by varying cache size and degree of associativity. In addition to the trace file, our algorithm takes as input the design constraint in the form of the number of desired cache misses. The output of the algorithm is a set of cache instances that meet the constraint.

The remainder of this paper is organized as follows. In Section 2, we outline our technical approach and introduce our data structures and algorithm. In Section 3, we present our experiments and show our results. In Section 4, we conclude with some final remarks and future direction of research.

2. Technical Approach

2.1 Overview

In the following approach, we consider a design space that is obtained by varying caches depth *D* and the degree of associativity *A*. Cache depth *D* gives the number of rows in the cache. In other words, $log_2(D)$ gives the bit-width of the index portion of the memory address. Degree of associativity *A* is the number of storage available to accommodate data/instruction words mapping to the same cache row (a.k.a., cache line/block). Our objective is to obtain a set of optimal cache pairs (*D*, *A*) for a given number *K* of desired cache misses. Note that by using the cache depth *D*, degree of associativity *A*, and line size L_{size}, we obtain the total cache size by computing $D \times A \times L_{size}$. Also, note that the *K* desired caches misses are assumed to be those beyond the cold misses, as cold misses cannot be avoided.

In our approach, we do not consider the cache line size as a varying parameter. In part, our decision is due to the fact that a change in the cache line size would require redesign of processor memory interface, bus architecture, main memory controller, as well as main memory organization. Thus, changing of cache line size requires a more encompassing design space exploration. Likewise, we have assumed fixed least recently used replacement and write-back policies, as these are the most common and often optimal choices.

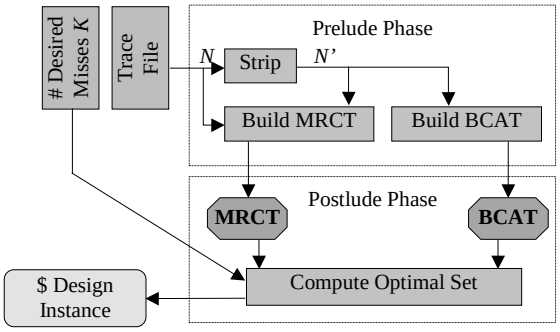

Figure 2: Block diagram of proposed approach.

Our approach can be divided into two logical phases, namely, the prelude algorithms and the postlude algorithm. During the execution of the prelude algorithms we process the trace file and construct two key data structures. One of these data structures is a *Binary Cache Allocation Tree* BCAT. The other data structure is the *Memory Reference Conflict Table* MRCT. During the execution of the postlude algorithm, and while operating on the BCAT and MRCT data structures, we compute the optimal cache pairs (*D*, *A*), which are guaranteed to result in a miss rate of less than *K*. A block diagram of our analytical approach is shown in Figure 2. We next describe in detail the two phases of our approach and define further the purpose of the key data structures.

2.2 Prelude Phase

Recall that a trace of *N* instruction/data memory references is obtained after simulating the target application on a processor whose cache is being optimized. We reduce this trace of *N* memory references into a set of *N*' unique references, where $N' \leq N$. In other words, the original trace contained repetitions of these *N*' memory references. As part of a running example, consider the trace shown in Table 1 along with the stripped version shown in Table 2.

B_3	B_2	B_1	B_0
1	0	1	1
1	1	0	0
0	1	1	0
0	0	1	1
1	0	1	1
0	1	0	0
1	1	0	0
0	0	1	1
1	0	1	1
0	1	1	0

Table 1: Original trace.

ID	B_3	B_2	B_1	B_0
1	1	0	1	1
2	1	1	0	0
3	0	1	1	0
4	0	0	1	1
5	0	1	0	0

Table 2: Stripped trace.

Our trace contains 10 4-bit references. Of those, there are 5 unique references. We have assigned a numeric identifier to each of the unique references as shown in Table 2. (At times, we may simply refer to a particular reference using its numeric identifier.)

To compute the BCAT data structure, we first transform the stripped trace into an array of zero/one sets. The array of zero/one sets contains a pair of sets for each address bit. Specifically, for index bit B_i, we compute a pair of sets called zero Z_i and one O_i. The set Z_i contains the identifier of all references that have a bit value of 0 at B_i. Likewise, the set O_i contains the identifier of all references that have a bit value of 1 at B_i. For our running example, shown in Table 1, the zero/one sets are given in Table 3.

	2.2.1.1.1.1	O
$\mathbf{B_0}$	{2,3,5}	{1,4}
$\mathbf{B_1}$	{2,5}	{1,3,4}
$\mathbf{B_2}$	{1,4}	{2,3,5}
$\mathbf{B_3}$	{3,4,5}	{1,2}

Table 3: Zero/one sets.

The choice of a BCAT data structure is due to the fact that it fully captures how references are mapped onto a cache of any possible organization. To construct this tree, we use the array of zero/one sets given earlier. We use these sets because the set intersection operation nicely defines how references are allocated to each cache location. For example, in a cache of depth 4 (i.e., 4 indexed rows), using B_0 and B_1 as the index bits, we can compute the following cross intersections: $L_{00}=Z_0 \cap Z_1=\{2,5\}$, $L_{01}=Z_0 \cap O_1=\{3\}$, $L_{10}=O_0 \cap Z_1=\{\}$, and $L_{11}=O_0 \cap O_1=\{1,4\}$. Here sets L_{00}, L_{01}, L_{10}, and L_{11} contain the reference identifiers mapped onto the 4 cache slots. Likewise, for a cache of depth 8, using an additional index bit B_2, we cross intersect each of these 4 sets with Z_2 and O_2 to obtain the 8 new sets and so on. The new sets form the nodes of our binary tree. We stop growing the tree further down when we reach a set with cardinality less than 2. Algorithm 1 recursively builds a BCAT data structure as described here.

Algorithm 1

Input: array of zero Z and one O sets
Output: data structures *BCAT*
BCAT.root $\Leftarrow (Z_0,O_0)$
call **build-tree**(*BCAT.root*, 1)
begin **build-tree**(node $n=(Z,O)$, l)
  if $|Z| >= 2$ then
    $n.left \Leftarrow (Z \cap Z_l, Z \cap O_l)$
    call **build-tree**(*n.left*, $l + 1$)
  if $|O| >= 2$ then
    $n.right \Leftarrow (O \cap Z_l, O \cap O_l)$
    call **build-tree**(*n.right*, $l + 1$)
end **build-tree**

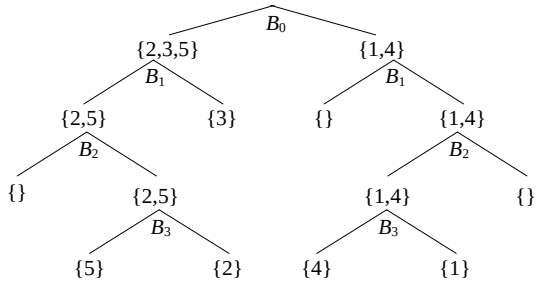

Figure 3: BCAT data structure.

The complete BCAT data structure for our running example is shown in Figure 3.

Next, we look at the MRCT data structure. The choice of an MRCT data structure is due to the fact that it captures, for each occurrence of a reference, a set of references that may cause a conflict. In other words, the MRCT data structure is an array of size N' (number of unique references) of sets, where each set corresponds to one unique reference. Moreover, each set is composed of sets, each corresponding to an occurrence (excluding the first cold occurrence) of that unique reference in the original trace. To clarify, consider the MRCT data structure of our running example shown in Table 4.

ID	Conflict Sets
1	{{2,3,4}, {2,4,5}}
2	{{1,3,4,5}}
3	{{1,2,4,5}}
4	{{1,2,5}}
5	{}

Table 4: MRCT data structure.

Here, the reference "1011" has 3 occurrences. The first occurrence of "1011" is ignored as it will always be a cold miss. The second occurrence of "1011" can potentially be a miss due to a conflict with references "1100", "0110", or "0011" (i.e., the set {2,3,4}). The last occurrence of "1011" can potentially be a miss due to a conflict with references "0100", "1100", or "0011" (i.e., the set {2,4,5}). So, the set of sets for reference "1011" contains two sets, namely {{2,3,4}, {2,4,5}}. Algorithm 2 builds an MRCT data structure as described here.

Algorithm 2

Input: memory references $R_1 \ldots R_N$
Input: unique reference $U_1 \ldots U_{N'}$
Output: memory reference conflict table T
for $i \in \{1 \ldots N'\}$
  $T_i \Leftarrow S_i \Leftarrow \varnothing$
for $j \in \{1 \ldots N\}$ do
  for $i \in \{1 \ldots N'\}$ do
    if $R_j = U_i$ then
      $T_i \Leftarrow T_i \cup S_i$
      $S_i \Leftarrow \varnothing$
    else
      $S_i \Leftarrow S_i \cup R_j.identifier$

2.3 Postlude Phase

Let us now compute a set of cache depth D and degree of associativity A pairs that would result in K or less misses. We start by looking at the BCAT data structure of our running example, shown in Figure 3. Note that each level of the tree corresponds to a particular cache depth. For example, level one of the tree (root being level zero) corresponds to a cache of depth two. At this level, the nodes of the BCAT tree capture the reference instances that would map to the two cache rows, namely any reference identified as one of {2,3,5} would map to the first cache row and any reference identified as one of {1,4} would map to the second cache row. In essence, for a cache of depth two with zero desired misses, we would need to set the degree of associativity A equal to the maximum cardinality of the two sets {2,3,5}, and {1,4} (i.e., $A=max(|\{2,3,5\}|,|\{1,4\}|)=3$). A similar approach can be taken to compute the degree of associativity A of a cache with depth four. Here, the degree of associativity A is set to the maximum of the cardinality of the sets {2,5}, {3}, {}, and {1,4} corresponding to the nodes at level two of the BCAT, and so on for the any other cache depth.

Clearly, the above approach is too conservative and produces caches that are *ideal*, in other words, caches that result in exactly zero misses (not counting cold misses). However, when the desired number of cache misses is greater than zero, we need to compute the minimum degree of associativity A that would satisfy our constraint. The MRCT data structure is used to accomplish this. Once again, for any particular cache depth, we look at the corresponding BCAT level. For each node at that level we determine the number of misses (described below) that would occur if the degree of associativity A was set to 1, 2…A_{zero}. Where A_{zero} is the degree of associativity required to have zero misses at that node. Consequently, we choose the smallest A that results in the sum of the misses of the individual nodes to be less than the desired number of misses K.

Lets us now compute the number of misses at a particular node given a particular degree of associativity A. Let us assume that the set of references mapping to this node is S. For each member of S we refer to the corresponding conflict sets C_1, C_2, … from the MRCT data structure. We count as a miss each time the cardinality of the intersection of the set S with C_i is larger than or equal to A. To illustrate, let us look at the rightmost node at level two of our BCAT example with S={1,4} and assuming A=1. From the MRCT data structure we obtain the conflict sets of the first element (i.e., 1), namely, C_1={2,3,4} and C_2={2,4,5}. Since the cardinality of the intersection of S and C_1 is one, we increment our miss count at that level. Likewise since the cardinality of the intersection of S and C_2 is one, we increment our miss count at that level for a second time. We repeat the same for the second element in S (i.e., 4). Note that a miss count is associated with each degree of associativity A under consideration (i.e., 1, 2…A_{zero}). We stop to consider a particular degree of associativity A when its miss count goes beyond the desired number of desired misses K. The complete procedure to compute the set of optimal cache instances is presented in Algorithm 3.

Algorithm 3

Input: data structures *BCAT* and *MRCT*
Input: desired number of cache misses K
Output: optimal pairs $(D,A)_1$, $(D,A)_2$…
for $i \in \{1…|BCAT.depth|\}$
  $min_i \Leftarrow 1$
  for each node n in *BCAT*
    for $j \in \{1…|n.S|\}$
      $count_j \Leftarrow 0$
for $i \in \{1…|BCAT.depth|\}$
  for each node n at level i of *BCAT*
    for each element $e \in n.S$
      for each set $C \in MRCT_e$
        for $j \in \{min_i…|n.S|\}$
          if $|C \cap n.S| \geq j$
            $count_i \Leftarrow count_i + 1$
            if $count_i > K$
              $min_i \Leftarrow min_i + 1$
for $i \in \{1…|BCAT.depth|\}$
  $(D,A)_i \Leftarrow (2^i, min_i)$

2.4 Final Remarks

The data structure and algorithms described above are presented in a manner to illustrate the logic and intuition behind our analytical cache optimization technique. Here, we comment on issues to be considered in an actual implementation (such as the one used to obtain the results in our experiments section).

- Stripping of a trace amounts to sorting the references and thus could take as long as $N \times log(N)$ steps. However, using a hash table can substantially improve the performance of this step of the algorithm.
- In Algorithm 2, the building of the MRCT data structure can be performed during the stripping of the trace with no additional added time complexity using a hash table.
- The extensive use of sets in our technique is due to the fact that sets are efficient to represent, store, and manipulate on a computer system using bit vectors. In addition, the use of sets allows for execution of the algorithm on a cluster of machines by utilizing a distributed set library, enabling the processing of very large trace files.
- The implementation of Algorithm 1 and Algorithm 3 can be combined. Specifically, the BCAT does not need to be calculated in its entirety. Instead, a depth first traversal of the tree can be performed. This also would reduce the

space complexity of the algorithm from exponential down to linear.

Finally, we note that the space complexity of our analytical approach is of the order of the size of the trace file. In designing embedded systems, this is not likely to be a limitation as most embedded systems execute a small kernel of the code most of the time.

3. Experiments

For our experiments, we used 12 typical embedded system applications that are part of the PowerStone benchmark applications [4]. The applications included a Unix compression utility called *compress*, a CRC checksum algorithm called *crc*, an encryption algorithm called *des*, an engine controller called *engine*, an FIR filter called *fir*, a fax decoder called *g3fax*, a sorting algorithm called *ucbqsort*, an image rendering algorithm called *blit*, a POCSAG communication protocol called *pocsag*, and a few other applications.

We first compiled and executed the benchmark applications on a MIPS R3000 simulator. Our processor simulator is instrumented to output separate instruction and data memory reference traces. The size of the traces *N*, the number of unique references *N*' and the maximum number of misses are reported for all the data traces in Table 5 and all the instruction traces in Table 6. In these tables, the maximum number of misses is obtained by simulating the traces on a cache simulator configured to be direct mapped with the cache depth set to one.

Benchmark	Size *N*	Unique References *N*'	Max. Misses
adpcm	18431	381	17066
bcnt	456	162	376
blit	4088	2027	4072
compress	58250	8906	48924
crc	2826	603	2787
des	20162	2241	20149
engine	211106	225	166599
fir	5608	146	5521
g3fax	229512	3781	211576
pocsag	13467	515	11569
qurt	503	84	489
ucbqsort	61939	1144	59215

Table 5: Data trace statistics.

Benchmark	Size *N*	Unique References *N*'	Max. Misses
adpcm	63255	611	63255
bcnt	1337	115	1337
blit	22244	149	22244
compress	137832	731	137832
crc	37084	176	37084
des	121648	570	121648
engine	409936	244	409936
fir	15645	327	15645
g3fax	1127387	220	1127387
pocsag	47840	560	47840
qurt	1044	179	1044
ucbqsort	219710	321	219710

Table 6: Instruction trace statistics.

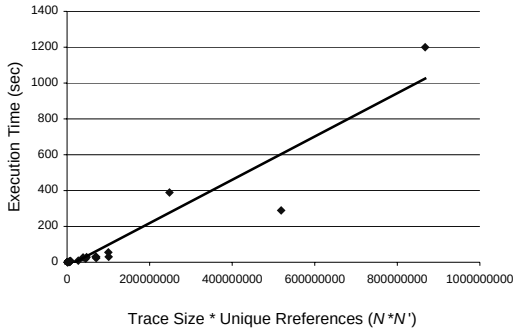

Figure 4: Execution efficiency of proposed approach.

We have ran these traces through our analytical algorithm for various values of desired number of cache misses *K*. Specifically, we have set *K* to one of 5%, 10%, 15%, and 20% of the maximum number of misses, which is shown in the last columns of Table 5 and Table 6. A traditional simulator has been used to verify the results generated by our algorithms. For brevity, we have presented the optimal cache instances for only one of the benchmarks, namely the data trace of *adpcm*, as computed by our algorithm in Table 7.

Cache Depth *D*	Degree of Associativity *A*			
	Desired Cache Misses *K* as a Percentage			
	5%	10%	15%	20%
2	115	114	109	106
4	115	115	109	106
8	60	57	54	53
16	33	28	27	26
32	16	14	13	13
64	9	8	7	6
128	4	4	4	3
256	3	3	2	2
512	2	2	1	1
1024	1	1	-	-

Table 7: Optimal cache instances.

In this table, the inner entries are the degree of associativity *A* necessary to ensure the desired number of cache misses. For example, for a cache of depth 512, a direct mapped cache would be sufficient to ensure less than 15% misses, while a two way set associative cache would be needed to assure less than 5% misses.

Our algorithm was executed on a Pentium III processor running at 1.0 GHz with 256 MB of memory. The average time taken to produce results for data and instruction traces is shown in Table 8 and Table 9.

In Figure 4 we have plotted the execution time on the vertical axis versus the size of the trace *N* multiplied by the number of unique references *N*' on the horizontal axis. It is easy to see that the time complexity of the algorithm is on the average linear with respect to the product of these two figures. In other words, it is faster than quadratic considering that the number of unique

references *N'* is much smaller than the number of total references *N*.

Benchmark	Time (sec)
adpcm	2.9
bcnt	0.11
blit	6.8
compress	290
crc	0.80
des	19
engine	28
fir	0.67
g3fax	1200
pocsag	3.2
qurt	0.090
ucbqsort	23

Table 8: Algorithm run time: data traces.

Benchmark	Time (sec)
adpcm	27
bcnt	0.13
blit	2.0
compress	30
crc	5.1
des	31
engine	56
fir	2.3
g3fax	390
pocsag	8.2
qurt	0.20
ucbqsort	31

Table 9: Algorithm run time: instruction traces.

4. Conclusion

We have presented an analytical approach to the design space exploration of caches that avoids exhaustive simulation. Our approach uses an analytical model of the cache combined with algorithms to directly and efficiently compute a cache configuration meeting designers' performance constraints. In our approach, we consider a design space that is formed by varying cache size and degree of associativity. For a given memory reference trace, our algorithm takes as input the design constraint in the form of the number of desired cache misses and outputs a set of optimal cache instances that meet the constraint. We have shown the feasibility of our algorithm by experimenting with 12 embedded applications, which are part of the PowerStone suite of benchmarks.

Our future direction of research will focus on incorporating additional design flexibility such as replacement policies, write policies, line size, bus architecture, multi-level cache hierarchies and other system-on-a-chip artifacts.

5. References

[1] International Technology Roadmap for Semiconductors. http://www.itrs.net.

[2] C. Kozyrakis, D. Patterson. A New Direction for Computer Architecture Research, IEEE Computer, pp. 24-32, 1998.

[3] F. Vahid, T. Givargis. The Case for a Configure-and-Execute Paradigm. International Symposium on Low Power Electronics and Design, 1999.

[4] A. Malik, B. Moyer, D. Cermak. A Lower Power Unified Cache Architecture Providing Power and Performance Flexibility. International Symposium on Low Power Electronics and Design, 2000.

[5] P. Petrov, A. Orailoglu. Towards Effective Embedded Processors in Codesigns: Customizable Partitioned Caches. International Workshop on Hardware/Software Codesign, 2001

[6] K. Suzuki, T. Arai, N. Kouhei, I. Kuroda. V830R/AV: Embedded Multimedia Superscalar RISC Processor. IEEE Micro, vol. 18, No. 2, pp.36-47, 1998.

[7] P. Petrov, A. Orailoglu. Towards Effective Embedded Processors in Codesigns: Customizable Partitioned Caches. International Workshop on Hardware/Software Codesign, 2001.

[8] C. Su, A.M. Despain. Cache Design Trade-offs for Power and Performance Optimization: A Case Study. International Symposium on Low Power Electronics and Design, 1995.

[9] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, S. Dwarkadas. Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures. International Symposium on Microarchitecture, 2000.

[10] Y. Li, J. Henkel. A Framework for Estimating and Minimizing Energy Dissipation of Embedded HW/SW Systems. Design Automation Conference, 1998.

[11] S.J.E. Wilton, N.P. Jouppi. CACTI: An Enhanced Cache Access and Cycle Time Model. IEEE Journal of Solid State Circuits, vol. 31, no. 5, 1996.

[12] T. Sato. Evaluating Trace Cache on Moderate-Scale Processors. IEEE Computer, vol. 147, no. 6, 2000.

[13] W. Shiue, C. Chakrabarti. Memory Exploration for Low Power Embedded Systems. Design Automation Conference, 1999.

[14] Z. Wu, W. Wolf. Iterative Cache Simulation of Embedded CPUs with Trace Stripping. International Workshop on Hardware/Software Codesign, 1999.

[15] M. Lajolo, A. Raghunathan, S. Dey, L. Lavagno, A. Sangiovanni-Vincentelli. Efficient Power Estimation Techniques for HW/SW Systems. IEEE Alessandro Volta Memorial Workshop on Low-Power Design, 1999.

[16] D. Kirovski, C. Lee, M. Potkonjak, W. Mangione-Smith. Synthesis of Power Efficient Systems-on-Silicon. Asian South Pacific Design Automation Conference, 1998.

[17] R.L. Mattson, J. Gecsei, D.R. Slutz, I.L. Traiger. Evaluation Techniques for Storage Hierarchies. IBM Systems Journal, vol. 9, no. 2, pp. 78-117, 1970.