# Cache Optimization For Embedded Processor Cores: An Analytical Approach

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

## ABSTRACT

*Embedded microprocessor cores are increasingly being used in embedded and mobile devices. The software running on these embedded microprocessor cores is often a priori known, thus, there is an opportunity for customizing the cache subsystem for improved performance. In this work, we propose an efficient algorithm to directly compute cache parameters satisfying desired performance criteria. Our approach avoids simulation and exhaustive exploration, and, instead, relies on an exact algorithmic approach. 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 [6][2][5]. Specifically, microprocessor cores (a.k.a., embedded processors) are playing an increasing role in such systems' design [1][9][7]. 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 [9][3][10]. 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 [15][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 [16][8]. In some of the other efforts, one-pass techniques are used in which numerous cache configurations are evaluated simultaneously during a single simulation run [4][11]. While these techniques reduce the

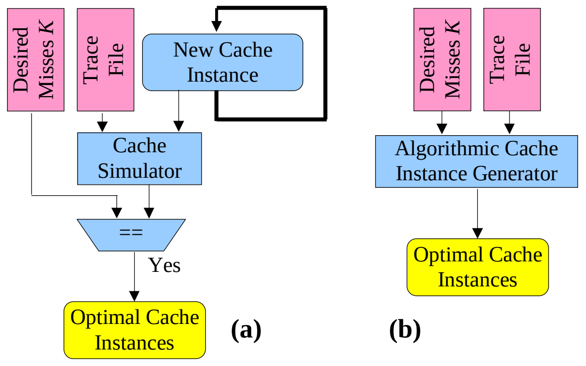

Figure 1: Design space exploration of caches: (a) traditional approach, (b) proposed approach.

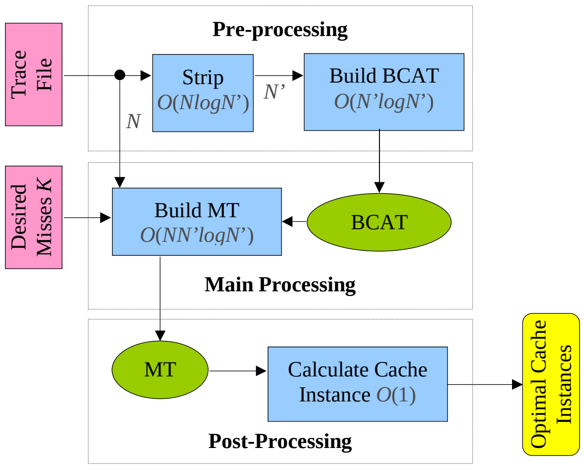

Figure 2: Block diagram of proposed algorithm.

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 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 amount of storage available to accommodate data/instruction words mapping to the same cache row (a.k.a., cache 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 row size as a varying parameter. In part, our decision is due to the fact that a change in the cache row size would require redesign of processor memory interface, bus architecture, main memory controller, as well as main memory organization. Thus, changing of cache row size requires a more encompassing design space exploration. Likewise, we have assumed fixed least recently used replacement and write-back policies, as these are common and often optimal choices.

Our approach can be divided into three phases, the pre-processing phase, the main processing phase and the post-processing phase. During the pre-processing phase, we read the trace file and construct a binary-tree data structure, called the *Binary Cache Allocation Tree* BCAT. In the main processing phase, we compute the *Miss Table* MT during a depth first traversal of the BCAT. In the post-processing phase, we generate 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 three phases of our algorithm and the associated data structures.

### 2.2 Pre-Processing 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 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 and 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.

In this example, the trace contains *N*=10 4-bit references. Of those, there are *N'*=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.) Next we describe the BCAT data structure.

A BCAT data structure fully captures how references are mapped onto a cache of any possible organization. Prior to

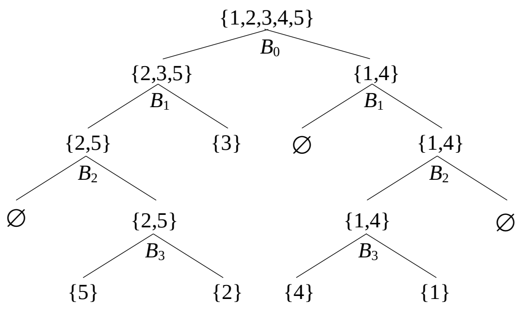

Figure 3: BCAT data structure.

computing the BCAT data structure, we 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 the running example, shown in Table 1, the zero/one sets are given in Table 3.

Table 3: Zero/one sets.

| | Z | O |
|---|---|---|
| **$B_0$** | {2,3,5} | {1,4} |
| **$B_1$** | {2,5} | {1,3,4} |
| **$B_2$** | {1,4} | {2,3,5} |
| **$B_3$** | {3,4,5} | {1,2} |

Next, the zero/one sets are used to construct the BCAT tree. 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 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 and Algorithm 2 recursively build a BCAT data structure as described here.

**Algorithm 1: Build-BCAT**

```
Input: Stripped Trace $T'=R_0, R_1 \ldots R_{N'-1}$
Output: $BCAT$ Data Structure
for each $i \in [M-1 \ldots 0]$ do // assume $M$-bit references
  $Z_i := O_i := \varnothing$
  for each $R_j \in T'$ do
    if $j^{th}$ bit of $R_j$ is 0 then
      $Z_j := Z_j \cup \{i\}$
    else
      $O_j := O_j \cup \{i\}$
$BCAT.root \Leftarrow Z_0 \cup O_0$
$BCAT$ := Recursive-Build-BCAT($BCAT.root, Z, O$, 1)
```

**Algorithm 2: Recursive-Build-BCAT**

```
Input: $BCAT$ Data Structure, Node $N$, Sets $Z/O$, and level $L$
Output: $BCAT$ Data Structure
if $|N| >= 2$ then
  $N.left \Leftarrow N \cap Z_L$
  $BCAT$ := Recursive-Build-BCAT($N.left, Z, O, L + 1$)
  $N.right \Leftarrow N \cap O_L$
  $BCAT$ := Recursive-Build-BCAT($N.right, Z, O, L + 1$)
```

The complete BCAT data structure of the running example is shown in Figure 3.

Associated with each node, we maintain a trace, called the *Relevant Trace Set* RTS. The RTS of a node is a subset of the RTS of its parent node containing only the references mapped onto the current node. For the root, RTS is the original trace. For other nodes, RTS is created dynamically during the main processing phase. (See Algorithm 7.)

## 2.3 Main Processing Phase

In the main processing phase, we build up the *Miss Table* MT data structure by processing each node as it is encountered in a depth first traversal of the BCAT tree.

The MT data structure maintains, for each level $L$ of the BCAT, the number of misses for every associativity being considered, i.e., $A$=1 to $A=A_{max}$. Note that each level of the tree corresponds to a particular cache depth $D=2^L$. For example, level one of the tree (root being level zero) corresponds to a cache of depth two. Also, the maximum associativity at a given level, which results in no misses, can be calculated by setting $A$ to the maximum cardinality of all nodes in the BCAT at that level. An entry $MT_{L,A}$ gives the number of misses at level $L$ (i.e., depth $D=2^L$) for associativity $A$. For example, $MT_{3,2}$=15 means a cache of dept $D=2^3=8$ with associativity $A$=2 will result in 15 misses. The complete MT data structure for our running example is shown in Table 4.

Table 4: MT data structure.

| **Assoc. ⇒** | **1** | **2** | **3** | **4** | **5** |
|---|---|---|---|---|---|
| **Level** | | | | | |
| **0** | 5 | 4 | 4 | 2 | 0 |
| **1** | 5 | 2 | 0 | 0 | 0 |
| **2** | 4 | 0 | 0 | 0 | 0 |
| **3** | 4 | 0 | 0 | 0 | 0 |
| **4** | 0 | 0 | 0 | 0 | 0 |

The MT data structure is built using Algorithm 3.

**Algorithm 3: Build-MT**

```
Input: Original Trace $T$, Desired Misses $K$
Input: $BCAT$ Data Structure
Output: $MT$ Data Structure
$MT := \varnothing$; $BCAT.root.RTS = T$
for each node $N \in BCAT$ (depth first) do
  ($MT,N$) := Process-Node($MT,N,K$)
```

Processing of each node involves updating the MT data structure and creating the RTS (explained earlier) as well as the *Memory Reference Conflict Table* MRCT (explained next) for the children nodes, as shown in Algorithm 4.

**Algorithm 4: Process-Node**

```
Input: $MT$ Data Structure, Node $N$, and Desired Misses $K$
Output: $MT$ Data Structure, Node $N$
$MRCT$ := Build-MRCT($N$)
$MT$ := Update-MT($MRCT,K,MT,N.level$)
$N$ := Create-Children-RTS($N$)
```

The MRCT data structure of a node $N$ captures, for each occurrence of a reference $R$, the number of unique references that may cause a conflict with the next occurrence of $R$ in the RTS of $N$. In other words, the MRCT associated with node $N$ is an array of vectors, one for each unique reference $R$ mapped to $N$, containing a count of references that may cause a conflict with $R$. The MRCT data structure associated with the root node of the running example is shown in Table 5.

Table 5: MRCT data structure for root node.

| ID | Conflict Vectors |
|---|---|
| 1 | (3,3) |
| 2 | (4) |
| 3 | (4) |
| 4 | (3) |
| 5 | (0) |

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 element $MRCT_{1,1}$=3). The last occurrence of “1011” can potentially be a miss due to a conflict with references “0100”, “1100”, or “0011” (i.e., the element $MRCT_{1,2}$=3). So, the conflict vector for reference “1011” contains two elements, namely (3,3). Algorithm 5 builds the MRCT data structure as described above.

**Algorithm 5: Build-MRCT**

```
Input: Node $N$
Output: $MRCT$ Data Structure
$MRCT$ := $temp$ := $last$ := $\emptyset$
for $R_i \in N.RTS$ do
  for $R_j \in N$ do
    if ($j \neq i$) && ($i \notin temp_j$) then
      $MRCT[j][last[j]]$ := $MRCT[j][last[j]] + 1$
      $temp[j]$ := $temp[j] \cup \{i\}$
    else
      $last[j]$ := $last[j] + 1$
      $temp[j]$ := $\emptyset$
```

To update the MT, we observe that the value $MRCT_{i,j}$ provides the upper bound on the degree of associativity, for which the $i^{th}$ occurrence of the $j^{th}$ reference will result in a miss. To illustrate, let us look at the root of the BCAT example with $N$={1,2,3,4,5}. From the MRCT data structure we obtain the conflict vectors of the first element, namely $V_{11}$=3 and $V_{12}$=3. Since the value of $V_{11}$ is 3, we increment our miss count at that level by 1 for all associativities from 1 to 3. Likewise since the value of $V_{12}$ is 3, we increment our miss count at that level for a second time. We repeat the same for the remaining elements in $N$. Note that a miss count is associated with each degree of associativity $A$ under consideration (i.e., 1, 2…$A_{max}$). We stop to consider a particular degree of associativity $A$ when its miss count goes beyond the desired number of desired misses $K$, as shown in Algorithm 6.

**Algorithm 6: Update-MT**

```
Input: $MRCT$ Data Structure, Desired Misses $K$
Input: $MT$ Data Structure, Level $L$
Output: $MT$ Data Structure
for each row $i \in MRCT$ do
  for each element $j \in MRCT[i]$ do
    for $A \in [1…MRCT[i][j]]$ do
      if ($MT[L][A]$ != -1) && ($MT[L][A] > K$) then
        $MT[L][A]$ := -1 and break
      $MT[L][A]$ := $MT[L][A] + 1$
```

Finally, to build the RTS of the children, we follow the steps outlines in Algorithm 7.

**Algorithm 7: Generate-Children-RTS**

```
Input: Node $N$
Output: Node $N$
$N.left\text{-}child.RTS$ := $N.right\text{-}child.RTS$ := $\emptyset$
for $R_i \in N.RTS$ do
  if $R_i \in N.left\text{-}child$ then
    $N.left\text{-}child.RTS$ := $N.left\text{-}child.RTS \cup \{R_i\}$
  else
    $N.right\text{-}child.RTS$ := $N.right\text{-}child.RTS \cup \{R_i\}$
```

## 2.4 Postlude Phase

During the last phase of the algorithm, we read the MT data structure and output a set of cache depth and associativity pairs that satisfy the desired performance in terms of the number of cache misses, as shown in Algorithm 8.

**Algorithm 8: Calculate-Cache-Instances**

```
Input: $MT$ Data Structure
Print: A Set of ($D$,$A$) Cache Instances
for each level $L \in MT$
  $A$ := 0
  while $MT[L][A]$ = -1 do
    $A$++
  print cache instance ($2^L$,$A$)
```

In Algorithm 8, for depths (number of rows) equal to 1, 2, 4, etc. we print the optimal caches having the smallest degree of associativity to guarantee no more misses than the desired value $K$.

## 2.5 Time Complexity

For time complexity analysis, we use the size of the trace $N$ and the number of unique references $N'$ as the input parameters. We note that in most cases, $N'$ is much smaller than $N$. Moreover, log($N'$) is bounded by the width of the memory references (i.e., processor data-path), which is typically 32 or 64. We have shown the time complexity of each part of the algorithm in Figure 2, as explained next.

The average time taken to strip the trace amounts to sorting the references and thus is $O(N \times \log(N'))$.

The average time taken to build the BCAT data structure is $O(N'\times\log(N'))$. At the root, we processes one node by looking at the $N'$ unique references at a cost of $O(1\times N')$, at level one, we process two nodes by looking at $N'/2$ unique references at a cost of $O(2\times N'/2)$, at level two, we process four nodes by looking at $N'/4$ unique references at a cost of $O(4\times N'/4)$, etc. In general, at each level of the tree, the computation is bounded by $O(N')$. Since the number of nodes in the tree is $O(N')$ it follows that the depth of the is $O(\log(N'))$. Combining these, we obtain $O(N'\times\log(N'))$.

The average time taken to build the MT data structure is $O(N\times N'\times\log(N'))$ which is dominated by the computation involved in building the MRCTs of each node in BCAT. At the root, we process one node for which we compute the RTS data structure (taking $O(N)$) followed by the MRCT, which involves one pass over the RTS for each unique reference occurring at that node, (taking $O(N\times N')$). At the next level, we process two nodes for which we compute the RTS data structure (taking $O(2\times N/2)$) followed by the MRCT, which involves one pass over the RTS for each unique reference occurring at that node, (taking $O(2\times N/2\times N'/2)$), and so on for the remaining levels. In general, at each level of the tree, the computation is bounded by $O(N\times N')$. Since the number of nodes in the tree is $O(N')$ it follows that the depth of the tree is $O(\log(N'))$. Combining these, we obtain $O(N\times N'\times\log(N'))$.

Finally, the post-processing phase of the algorithm takes constant time to output the cache instances. Overall, the presented technique takes $O(N\times N'\times\log(N'))$ step to execute.

## 2.6 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 can be improved substantially by using a hash-table structure to keep track of unique reference. Moreover, the building of the MRCT data structure can be performed during the stripping of the trace with no additional added time complexity if a hash-table is used.

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 7 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 to reduce memory usage. Further, the data structures associated with each node can be deleted, after the node has been processed.

## 3. EXPERIMENTS

For our experiments, we have used 14 typical embedded system applications that are part of the PowerStone benchmark applications [1]. The applications include a JPEG decoder called *jpeg*, a modem decoder called *v42*, 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 group three fax decoder called *g3fax*, a sorting algorithm called *ucbqsort*, an image rendering algorithm called *blit*, a POCSAG communication protocol for paging applications called *pocsag*, and a few other embedded applications.

We first compiled and executed the benchmark applications on a MIPS R3000 simulator. Our processor simulator is instrumented to output instruction/data memory reference traces. The size of the traces $N$, the number of unique references $N'$, and the execution time of our algorithm are reported for data/instruction traces in Table 6/Table 7.

Table 6: Data trace statistics.

| Benchmark | Total Refs. *N* | Unique Refs. *N'* | Time (sec) |
|---|---|---|---|
| *adpcm* | 18431 | 381 | 2.7 |
| *bcnt* | 456 | 162 | 0.11 |
| *blit* | 4088 | 2027 | 6.879 |
| *compress* | 58250 | 8906 | 466.87 |
| *crc* | 2826 | 603 | 0.43 |
| *des* | 20162 | 2241 | 19.268 |
| *engine* | 211106 | 225 | 10.786 |
| *fir* | 5608 | 146 | 0.39 |
| *g3fax* | 229512 | 3781 | 221.098 |
| *jpeg* | 1311693 | 39302 | 100576 |
| *pocsag* | 13467 | 515 | 1.582 |
| *qurt* | 503 | 84 | 0.07 |
| *ucbqsort* | 61939 | 1144 | 17.516 |
| *v42* | 649168 | 23942 | 15628 |

Table 7: Instruction trace statistics.

| Benchmark | Total Refs. *N* | Unique Refs. *N'* | Time (sec) |
|---|---|---|---|
| *adpcm* | 63255 | 611 | 12.689 |
| *bcnt* | 1337 | 115 | 0.12 |
| *blit* | 22244 | 149 | 0.781 |
| *compress* | 137832 | 731 | 23.044 |
| *crc* | 37084 | 176 | 1.653 |
| *des* | 121648 | 570 | 22.954 |
| *engine* | 409936 | 244 | 34.47 |
| *fir* | 15645 | 327 | 1.60 |
| *g3fax* | 1127387 | 220 | 67.73 |
| *jpeg* | 4594120 | 623 | 693.876 |
| *pocsag* | 47840 | 560 | 5.988 |
| *qurt* | 1044 | 179 | 0.151 |
| *ucbqsort* | 219710 | 321 | 17.165 |
| *v42* | 2441985 | 656 | 389.856 |

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 1%, 2%, 3%, and 4%

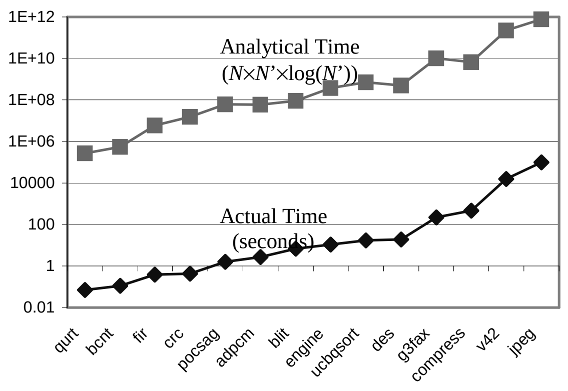

Figure 4: Analytical time complexity vs. actual run times.

cache misses. For brevity, we have presented the optimal cache instances for only one of the benchmarks, namely the data trace of *adpcm* in Table 8. The correctness of the proposed approach has been verified by subsequent cache simulation.

In this table, the inner entries are the degree of associativity *A* necessary to ensure the desired number of cache misses. For example, if 2% cache misses are allowed, a two-way set associative cache of depth 1024 would suffice.

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 the last column of tables Table 6 and Table 7.

In Figure 4 we have plotted the average measured time taken to produce results along with the analytical time complexity computed as $N \times N' \times \log(N')$ on a logarithmic scale. We note that the pattern of the plots match.

Table 8: Optimal cache instances of *adpcm*

| **Cache Depth *D*** | **Degree of Associativity *A*** | | | |
|---|---|---|---|---|
| | **Desired Cache Misses *K* as a Percentage** | | | |
| | **1%** | **2%** | **3%** | **4%** |
| 2 | 133 | 133 | 133 | 133 |
| 4 | 115 | 115 | 115 | 115 |
| 8 | 115 | 115 | 115 | 115 |
| 16 | 62 | 61 | 61 | 61 |
| 32 | 34 | 34 | 34 | 33 |
| 64 | 20 | 19 | 19 | 18 |
| 128 | 10 | 10 | 9 | 9 |
| 256 | 6 | 5 | 5 | 5 |
| 512 | 5 | 3 | 3 | 3 |
| 1024 | 3 | 2 | 2 | 2 |
| 2048 | 1 | 1 | 1 | 1 |

## 4. CONCLUSION

We have proposed an efficient algorithm to directly compute cache parameters satisfying desired performance criteria. The proposed approach avoids simulation and exhaustive exploration. Here, 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. The feasibility of the proposed approach has been verified experimentally using the PowerStone benchmarks. Future direction of research will focus on incorporating artifacts such as write-back policy, replacement policies, line size, multilevel caches, and bus architecture effects.

## 5. REFERENCES

[1] 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.

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

[3] 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.

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

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

[6] International Technology Roadmap for Semiconductors.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

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

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

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

[16] Z. Wu, W. Wolf. Iterative Cache Simulation of Embedded CPUs with Trace Stripping. International Workshop on HW/SW Codesign, 1999.