Improved Indexing for Cache Miss Reduction in Embedded Systems

Tony Givargis
Department of Computer Science, Center for Embedded Computer Systems
University of California, Irvine, CA 92697
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. In traditional cache design, the index portion of the memory address bus consists of the K least significant bits, where K=$\log_2$(D) and D is the depth of the cache. However, in devices where the application set is known and characterized (e.g., systems that execute a fixed application set) there is an opportunity to improve cache performance by choosing an optimal set of bits used as index into the cache. This technique does not add any overhead in terms of area or delay. We give an efficient heuristic algorithm for selecting K index bits for improved cache performance. We show the feasibility of our algorithm by applying it to a large number of embedded system applications as well as the integer SPEC CPU 2000 benchmarks.

Categories and Subject Descriptors
B.3.2 [Design Styles]

General Terms: Algorithms, Performance, Experimentation

Keywords
Cache Optimization, Design Space Exploration, Index Hashing

1. Introduction
The growing demand for embedded computing platforms, mobile systems, handheld devices, and dedicated servers coupled with shrinking time-to-market windows are leading to new core based system-on-a-chip (SOC) architectures [5][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 functional 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 can reduce the time overhead of fetching instruction and data from main memory, which in most cases resides off-chip, requiring power costly communication over the off-chip system bus.

Particularly, in embedded, mobile, and handheld devices, 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 area, clock latency, or energy.

In this work, we propose a zero cost technique for improving cache performance (i.e., reduce misses). Our technique involves selecting an optimal set of bits used as index into the cache. In traditional cache design, the index portion of the memory address bus consists of the K least significant bits, where $K=\log_2(D)$ and D is the depth of the cache [10]. In general, any of the address bits can be used for indexing. In our technique, we assume that the processor and cache cores are black-box entities to be integrated on a single SOC. However, we do assume that the integration of cores, more specifically, routing of the address bus wires is flexible, as is commonly the case in core-based SOC design.

We pictorially depict the idea of cache indexing by showing the traditional approach, **Figure 1**(a), versus our approach, **Figure 1**(b). Here we have a 16-bit processor core connected to a 1K-cache core, which in turn is connected to 64K of memory. In **Figure 1**(a), the least significant address bit is used for the byte-offset calculation (assuming the cache is organized with each line being two bytes wide). The next nine least significant bits are used for cache indexing and the remaining bits are used for tag comparison. In **Figure 1**(b), we have swapped bits seven and ten in order to achieve better cache indexing. Note that the reverse of the indexing scheme is performed on the cache-to-memory side in order to preserve functional correctness.

The problem of cache indexing is one of hashing. In traditional cache design, reference A maps to cache location L, using the following hash function: $L = A\%D$. Here, D is the depth of the cache. In general, we can use any hash function as follows: $L = h(A)$. Here, h is the arbitrary hash function. While it may be possible to compute a perfect hash function, given the cache organization and a trace file, in this work, we focus on a special class of hash functions, namely those that have zero cost overhead (e.g., zero delay, area, power, etc.). In other words, we focus on the class of hash function that only swap the address bits.

In related work, researchers have studied data layout and memory/cache aware compiler techniques for improved cache

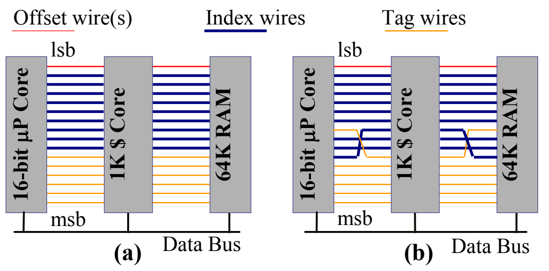


Figure 1: Cache indexing: (a) traditional approach, (b) our approach.

performance [11][12][13][14]. In these approaches, the code and data segment of a compiled application is moved such as to eliminate conflict misses. In the case of the code segment, data movement is typically performed at the basic block granularity. In the case of data segment, data movement is typically performed at array boundaries. A drawback of such approaches is that the degree of freedom in moving data is limited (e.g., a large continues array or a basic block of code cannot be split). In other related work, researchers have studied indexing and hashing in the context of IP routing [15][16]. In some of these approaches lookup tables are used to define the hash function. In other approaches analytical functions that optimize the hashing criteria are utilized. These approaches, if applied to processor cache indexing, would introduce a large unacceptable overhead, since memory access is already a bottleneck in improving processor performance. We are unaware of any direct research related to processor cache indexing as stated in this work.

The remainder of this paper is organized as follows. In Section 2, we formulate the problem and give our heuristic solution. In Section 3, we state our experiments. In Section 4, we conclude.

2. Optimal Cache Indexing

In this section, we first formulate the problem of optimal cache indexing. Then, we show that the problem of optimal cache indexing belongs to the class NP-complete. Last, we provide a heuristic that is efficient in running time and produces good results when applied in practice.

2.1 Problem Formulation

Optimal cache indexing is the problem of selecting K bits among all address bits of a processor for indexing into the cache. Specifically, let us assume that a processor has an M-bit bus and is connected to a cache of size S bytes that is A-way set associative and has line size equal to L bytes. K can be computed as follows:

$$K = \log_2\left(\frac{S}{L \times A}\right)$$

Here, the term $S / (L \times A)$ gives the depth D of the cache (i.e., the number of rows). Note that K is the number of bits used by the row decoder of the cache. Since there are a total of M address bits, we can potentially use any combination of size K for cache indexing as follows.

$$\binom{M}{K} = \frac{M!}{K! \times (M-K)!}$$

The problem is to find the one combination that reduces cache misses for a fixed application set. Specifically, we assume that a trace of memory references, corresponding to the application set, is available and is the input to our problem. In an exhaustive approach, one can find an optimal cache index set by enumerating all possible combinations, integrating the processor and cache accordingly, and simulating the application trace while keeping track of the one combination resulting in minimum misses. Such an approach is clearly not tractable as the number of combinations is normally very large. For example, assume a 32-bit processor connected to an 8192 bytes two-way set associative cache with line size equal to four bytes. K=10 is computed as follows.

$$K = \log_2\left(\frac{8192}{4 \times 2}\right) = 10$$

The number of possible cache index sets is over 64 million, and is computed as follows.

$$\binom{32}{10} = \frac{32!}{10! \times (32-10)!} = 64{,}512{,}240$$

We next show that the problem of optimal cache indexing belongs to the class NP-complete (i.e., unsolvable in polynomial time).

2.2 NP Completeness

The stated problem of optimal cache indexing belongs to the class NP-complete. For brevity, we only outline the proof idea. First we show that our problem belongs to the class NP. Then, we show that the NP-complete set intersection problem [17] is polynomial time reducible to our problem. Thus, it follows that the problem of optimal cache indexing is also NP-complete.

The problem of optimal cache indexing belongs to the class NP. To show this, we non-deterministically select N bits as the cache index set, integrate the processor and cache accordingly, and simulate the application trace. If the number of cache misses is zero we halt, otherwise, we repeat the process, for 1, 2 … N, where N is the length of the trace, misses. The above non-deterministic algorithm will find an optimal cache index set that results in the least number of cache misses.

We show that the set intersection problem is reducible to the problem of optimal cache indexing. In the set intersection problem, we are given a collection of sets S_1, S_2, … S_K and an integer m. The goal is to find a subset C of S_1, S_2, … S_K whose intersection (i.e., the intersection of all sets in C) has cardinality equal to m. Toward this goal, we first show how the problem of optimal cache indexing can be stated in a set theoretic form.

We first define a set U containing all the cold references. Next, we extract from the trace M sets X_0, X_1 … X_{M-1} where M is the address bus width of the processor. A set X_i captures memory conflicts that would occur in a cache of depth two and the i^{th} address bit used as the cache index. We illustrate this with an example. Consider the trace shown in **Table 1**.

ID	A_2	A_1	A_0
1	0	0	1
2	0	1	1
3	0	0	0
4	0	0	1
5	0	1	1

Table 1: A sample application trace.

Here, the address bus width M is three and the trace has five entries identified as one, two, three, four, and five. Note that the trace has three unique references, namely those identified as one, two, and three. References four and five are repetitions of previously seen values. The unique set U and the conflict sets X_0, X_1, and X_2 are given as follows.

$$U = \{1,2,3\}$$
$$X_0 = \{(4,2),(5,1)\}, X_1 = \{(4,3)\}, X_2 = \{(4,3),(4,2),(5,1),(5,3)\}$$

The set U contains the unique references in the trace. Each set X_i contains members that are pairs. The first element of each pair corresponds to the reference that results in a miss, given a cache of depth two with A_i used as the index bit. The second element of each pair, which is a member of U, corresponds to a reference that can cause a miss. The second element is a reference that may be replaced on a miss caused by the first element of the pair. In our example, in a cache of depth two, with A_0 used as the index bit, reference four would be a miss because of reference two, thus (4,2) is an entry into the set X_0. Likewise, reference five would be a miss because of reference one, thus (5,1) is an entry into the set X_0. For A_2, reference four would be a miss because of reference three as well as reference two, thus we have (4,3) and (4,2) as a member of the set X_2, and so on.

The unique set U and the conflict sets X_0, X_1 … X_{M-1} fully capture the information content of the trace necessary to compute cache performance for any arbitrary configuration of the cache. The number of cache misses for a cache of depth two and associativity of one, using A_i as the index bit, is given by the cardinality of the corresponding set X_i plus the cardinality of the unique set U as shown in the first three rows of **Table 2**.

Index Bits	Set Intersections	Cardinality/#Misses
X_0	$X_0 = \{(4,2), (5,1)\}$	3 + 2 = 5
X_1	$X_1 = \{(4,3)\}$	3 + 1 = 4
X_2	$X_2 = \{(4,3), (4,2), (5,1), (5,3)\}$	3 + 2 = 5
X_0 and X_1	$X_0 \cap X_1 = \varnothing$	3 + 0 = 3
X_0 and X_2	$X_0 \cap X_2 = \{(4,2), (5,1)\}$	3 + 2 = 5
X_1 and X_2	$X_1 \cap X_2 = \{(4,3)\}$	3 + 1 = 4
X_0, X_1 and X_2	$X_0 \cap X_1 \cap X_2 = \varnothing$	3 + 0 = 3

Table 2: Using set intersections to compute number of misses.

In computing the cardinality, we avoid double counting pairs that have identical first element. For example, in X_2, (4,3) and (4,1) are counted once, as they both refer to the same missed reference, namely the reference identified as four. In general, the cardinality calculation can be generalized for caches of higher associativity as shown in the following function.

$$Cardinality(X, A) := |U| + m \quad where$$
$$m = m_0 + m_1 + \ldots + m_k \quad where$$
$$m_i := \begin{cases} 0 & e_i > A \\ 1 & otherwise \end{cases} \quad where$$
$$e_i := 0$$
$$for\ (i, j) \in X$$
$$e_i := e_i + 1$$

Here, bottom up, we compute for each unique entry in a set X, its number of appearance as e_i. For example, in X_2, the reference identified as four appears twice, thus e_4 is two, and the reference identified as five appears twice, thus e_5 is also two. Then, we count reference i as a miss, denoted by m_i, if its count is greater than the degree of the associativity of the cache. The actual number of misses, denoted by m, is the sum of m_0, m_1 … m_k.

To continue, let us consider a cache of depth four. Here, the misses for each possible index mapping is given by taking the cardinality of the pair wise intersection of the conflict sets as shown in the middle three rows of **Table 2**. Likewise, in our example, for a cache of depth eight, we take the triple intersection of the conflict sets, as shown in the last row of **Table 2**.

Generally, once a trace has been captured as a collection of conflict sets, the problem of finding an optimal cache indexing solution can be found by attempting to find a subset of these conflict sets, such that when intersected, has the lowest minimal cardinality, as defined by the cardinality function. This is an identical problem to the set intersection problem stated earlier.

2.3 Heuristic Algorithm

Since the problem of optimal cache indexing is NP-complete, we give a heuristic algorithm that is efficient and performs well for a large number of applications in our experiments. The first step of the algorithm is simply reading a trace into memory. We denote the size of the trace as N. The next step is to reduce the trace to the unique references, denoted as N', where $N' \le N$. We next describe the remaining parts of the algorithm.

For each bit in our address space, we compute a corresponding quality measure. This quality measure is a real number in the range of zero to one. Having a quality of zero would indicate that the bit, if used as an index into a cache of depth two, would be a poor choice, as it would place all the references into a single location in the cache. On the other hand, having a quality of one would indicate that the bit, if used as an index into a cache of depth two, would be a good choice, as it would equally split all the references among the two cache locations. We compute the quality Q_i for address bit A_i by taking the ratio of zeros and ones along the A_i^{th} column. This is shown in the following equation.

$$Q_i = \frac{\min(Z_i, O_i)}{\max(Z_i, O_i)} \quad where$$
$$Z_i : the\ number\ of\ refereces\ having\ 0\ at\ bit\ A_i$$
$$O_i : the\ number\ of\ refereces\ having\ 1\ at\ bit\ A_i$$

As an example, consider the trace shown in **Table 3**.

A_5	A_4	A_3	A_2	A_1	A_0
0	1	1	0	1	1
0	0	1	1	0	0
0	0	0	1	1	0
0	1	0	0	1	1
1	0	1	0	1	1
0	0	0	1	0	0
0	1	1	1	0	0
0	0	0	0	1	1
0	0	1	0	1	1
1	0	0	1	0	0

Table 3: A sample striped application trace.

Here, Q_0, Q_1 … Q_5 are computed as shown in **Table 4**.

Q_5	Q_4	Q_3	Q_2	Q_1	Q_0
¼	3/7	1	1	2/3	1

Table 4: Quality measures.

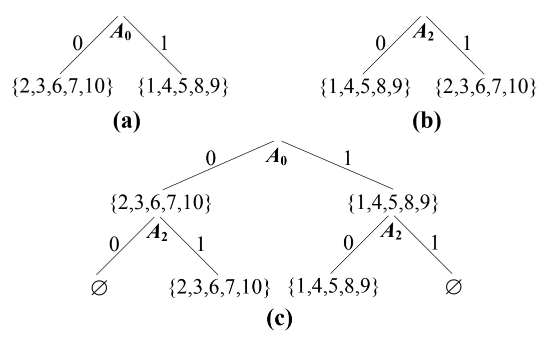


Figure 2: Correlation measure: (a) A_0 used as index, (b) A_2 used as index, (c) A_0 and A_2 used as indices.

As an example, looking at **Table 3** and for Q_4, A_4th column, there are seven zeros and three one bits, thus we compute as follows.

$$Z_4 = 7, O_4 = 3$$
$$Q_0 = \frac{\min(7,3)}{\max(7,3)} = \frac{3}{7}$$

For each pair of bits in our address space, we compute a corresponding correlation measure. This correlation measure is a real number in the range of zero to one. A correlation measure of zero indicates that a pair of address bits split the unique references in exactly the same way. A correlation measure of one indicates that a pair of address bits split the unique references in completely different ways. To illustrate further, **Figure 2**(a) and **Figure 2**(b) pictorially depict how A_0 and A_2 split the trace shown in **Table 3**. (Note that according to our quality measure, both A_0 and A_2 are ideal indices to use in a cache of depth two.) Now consider the case where we have a cache of size four, thus needing a pair of indices. If we use A_0 and A_2, the trace would be split into the four cache locations as shown in **Figure 2**(c). Note that even though the cache has four slots, two slots receive the references, and two slots remain empty. The reason for this is that A_0 and A_2 are correlated. From looking at the trace, we can see that the A_2 is simply the complement of A_0. In such a case, we would have a correlation measure C_{ij} equal to zero. In general, we can compute the correlation C_{ij}, for bits A_i *and* A_j as follows.

$$C_{ij} = \frac{\min(E_{ij}, D_{ij})}{\max(E_{ij}, D_{ij})} \quad where$$
$$E_{ij} : the\ number\ of\ refereces\ having\ identical\ bits\ at\ A_i\ and\ A_j$$
$$D_{ij} : the\ number\ of\ refereces\ having\ different\ bits\ at\ A_i\ and\ A_j$$

The correlation measures for our example are given in **Table 5**.

	A_5	A_4	A_3	A_2	A_1	A_0
A_5	0	1	1	1	2/3	1
A_4	1	0	2/3	2/3	1	2/3
A_3	1	2/3	0	2/3	1	2/3
A_2	1	2/3	2/3	0	1/9	0
A_1	2/3	1	1	1/9	0	1/9
A_0	1	2/3	2/3	0	1/9	0

Table 5: Correlation measures.

As an example, looking at **Table 3**, for C_{23}, along columns A_2 and A_3, there are six rows where the bits are different and four rows where the bits are identical, thus we compute as follows.

$$E_{23} = 4, D_{23} = 6$$
$$C_{23} = \frac{\min(4,6)}{\max(4,6)} = \frac{4}{6} = \frac{2}{3}$$

During the last step of the algorithm, we use the quality measure along with the correlation measure to compute the final index mapping as shown in Algorithm 1.

```
Algorithm 1
Input:   $Q_0, Q_1 \ldots Q_{M-1}$
Input:   $C_{00}, Q_{01} \ldots C_{(M-1)\times(M-1)}$
Output:  an ordering of $A_0, A_1 \ldots A_{M-1}$
loop
  select $A_b = \max\{ Q_0, Q_1 \ldots Q_{M-1} \}$
  for each $Q_i \in \{ Q_0, Q_1 \ldots Q_{M-1} \}$
    $Q_i := Q_i \times C_{bi}$
  halt when all $A_i$'s are selected
```

This algorithm repeatedly selects an address bit with the highest corresponding quality measure and then updates the quality measures using the correlations. For example, for the trace given in **Table 3** and quality/correlation measures computed in **Table 4** and **Table 5**, the algorithm first select A_0 as the best index bit and updates the quality measures Q_i by multiplying with C_{0i} to obtain a new set of quality measures. Next, having the largest quality measure, the algorithm selects A_3, and update the quality measures again, and so on. On termination, we obtain A_0, A_3, A_5, A_4, A_1, A_2 as the final cache index mapping. This ordering defines a near-optimal solution to the problem of cache indexing. To build a cache of depth two we choose A_0. To build a cache of depth four we choose A_0 and A_3, and so on.

In terms of running time complexity, our algorithm takes $O(N \times \log(N))$ to execute. Note that reading the trace takes $O(N)$, as the length of the trace is N. Reducing the trace down to only the unique references involves what amounts to sorting the trace and thus takes $O(N \times \log(N))$. Computing the quality and correlation measures takes $O(N')$, where $N' \leq N$ is the number of unique references, as a single pass over the unique references is needed to compute these values. The final phase of the algorithm takes $O(M)$ where M is the width of the address bus, as the loop iterates exactly M times to order $A_0, A_1 \ldots A_{M-1}$. In most cases M is a small number, like 32, and thus is assumed to be a constant.

3. Experiments

For experiments, we have used the Powerstone embedded benchmarks [4] as well as the integer SPEC CPU 2000 general benchmarks [18]. The PowerStone benchmarks include a JPEG decoder called *jpeg*, a modem protocol processor 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*, a rendering algorithm called *blit*, a POCSAG communication protocol for paging called *pocsag*, etc.

We have compiled and executed each benchmark application on a MIPS R3000 simulator, instrumented to output memory reference traces for both instruction and data accesses. We have run the traces through our heuristic algorithm to obtain improved cache index mappings. Our results are summarized in **Table 6**. The last column of the table gives the improved cache index sets (the most significant 10 bits, as used in our cache configurations are shown).

Benchmark Name	#Refs	#Unique Refs	Improved Cache Index Mappings
PowerStone/Data			
adpcm	18431	381	4,6,8,9,5,7,12,10,11,13
Bcnt	456	162	6,4,5,9,7,8,14,15,10,3
Blit	4088	2027	4,5,14,6,7,8,9,10,11,12
compress	58250	8906	6,9,8,5,4,7,12,10,14,11
Crc	2826	603	4,7,6,3,5,9,11,8,10,2
Des	20162	2241	5,4,7,8,6,9,10,14,11,15
engine	211106	225	4,10,17,7,9,5,8,6,3,2
Fir	5608	146	7,4,5,8,6,2,9,10,11,22
g3fax	229512	3781	7,2,4,3,6,22,12,8,5,9
Jpeg	1311693	39302	8,4,6,5,7,10,11,9,12,14
pocsag	13467	515	4,7,5,6,2,10,8,3,9,11
Qurt	503	84	4,10,5,6,7,8,9,11,15,2
ucbqsort	61939	1144	6,5,8,9,10,4,7,11,16,19
V42	649168	23942	6,9,4,7,5,8,10,11,12,13
SPEC'00/Data			
Bzip2	40.1G	91.4M	17,8,14,19,18,28,16,22,23,7
crafty	70.2G	1.94M	8,9,10,15,16,20,11,19,21,14
Eon	38.8G	0.559M	22,10,14,11,2,6,7,9,5,19
Gap	80.9G	67.3M	20,19,21,12,11,17,6,8,25,15
Gcc	25G	161M	18,24,6,13,25,8,20,3,7,15
Gzip	24.8G	89.8M	16,20,8,18,23,26,14,13,2,3
Mcf	23.1G	198M	16,27,7,28,12,17,14,4,21,5
parser	191G	38.2M	18,10,17,16,6,7,4,8,25,13
perlbmk	18.6G	77.2M	19,18,6,28,7,11,3,22,20,13
Twolf	112G	5.73M	17,24,25,6,16,23,5,9,11,3
vortex	48.2G	76.2M	7,25,23,11,16,3,26,12,28,22
Vpr	37.1G	51.7M	18,14,7,12,11,26,25,10,22,4
PowerStone/Instruction			
adpcm	63255	611	2,3,8,5,7,4,6,9,12,10
Bcnt	1337	115	2,3,4,5,6,7,8,11,9,0
Blit	22244	149	2,3,4,5,10,7,8,9,11,12
compress	137832	731	3,2,7,4,11,5,8,6,10,9
Crc	37084	176	2,3,4,6,11,7,9,10,12,8
Des	121648	570	2,3,7,4,5,8,12,9,10,11
engine	409936	244	2,3,4,5,7,10,8,6,11,12
Fir	15645	327	7,2,3,8,4,5,6,9,11,12
g3fax	1127387	220	2,4,3,6,5,8,7,9,12,13
jpeg	4594120	623	2,3,5,4,8,6,7, 13,14,10
pocsag	47840	560	2,6,3,5,4,10,9,8,7,11
qurt	1044	179	2,3,5,4,8,6,10,9,7,11
ucbqsort	219710	321	2,3,5,4,6,12,13,8,7,10
v42	2441985	656	2,3,8,12,13,5,6,4,7,9
SPEC'00/Instruction			
bzip2	109G	0.00487M	7,8,9,10,13,14,15,16,12,6
crafty	192G	0.16M	12,13,14,15,18,19,20,21,5,6
eon	80.6G	0.206M	18,19,20,21,2,3,4,5,6,12
gap	214G	0.123M	3,4,5,6,13,14,15,16,11,12
gcc	46.1G	0.986M	18,19,20,21,14,15,16,17,12,13
gzip	844G	0.00486M	5,6,7,8,2,3,4,11,12,13
mcf	61.9G	0.0475M	9,10,11,12,8,13,14,15,16,7
parser	547G	0.105M	9,10,11,12,16,17,18,19,5,6
perlbmk	41.1G	0.328M	2,3,4,5,17,18,19,20,6,7
twolf	346G	0.177M	16,17,18,19,2,3,4,5,6,9
vortex	119G	0.358M	17,18,19,20,4,5,6,7,8,9
vpr	84.3G	0.156M	18,19,20,21,2,3,4,5,15,16

Table 6: Optimal cache indexing.

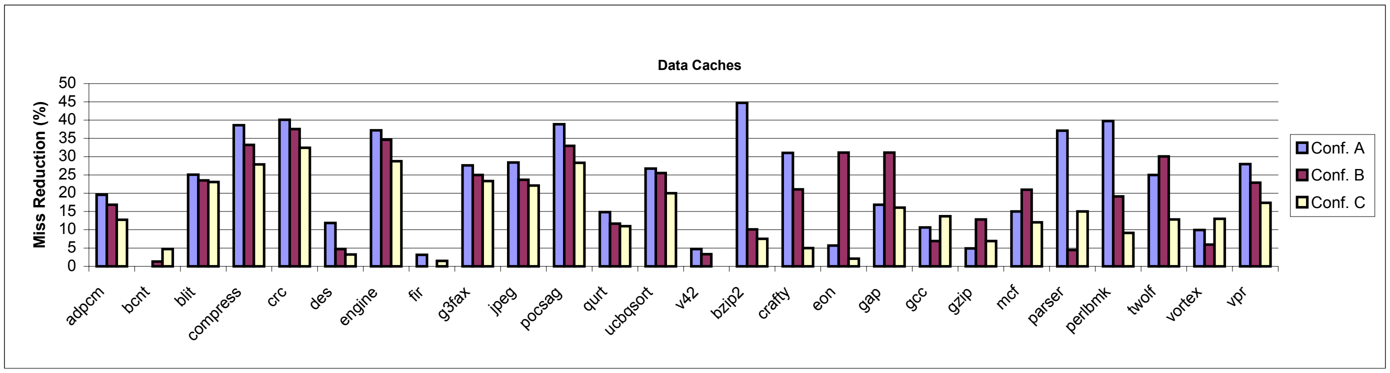
We have simulated the traces under three typical cache organization schemes. Configuration *A* with 4Kb, direct mapped, and 4-byte line, configuration *B* with 8Kb, 2-way, and 8-byte line; and configuration *C* with 16Kb, 4-way, and 16-byte line.

For each of the three cache configurations, we have measured the number of misses when traditional **(T)** cache indexing as well as when the proposed (i.e., improved) **(P)** cache indexing is used. The results are summarized in **Table 7**.

Benchmark	Config. A		Config. B		Config. C	
	T	P	T	P	T	P
PowerStone/Data						
adpcm	5193	4175	2181	1813	621	542
bcnt	164	164	156	154	147	140
blit	4034	3022	4025	3078	4038	3106
compress	12659	7772	9603	6414	7861	5671
crc	694	416	485	303	228	154
des	15155	13360	12849	12239	10523	10179
engine	7131	4479	3482	2277	132	94
fir	658	637	139	139	136	134
g3fax	127828	92503	65143	48855	35158	26940
jpeg	267567	191542	169490	129399	79258	61757
pocsag	1238	757	530	355	268	192
qurt	115	98	77	68	73	65
ucbqsort	10862	7955	3309	2463	804	643
v42	157469	150021	111108	107441	87592	87592
SPEC'00/Data						
bzip2	3.15M	1.74M	1.39M	1.25M	1.07M	0.989M
crafty	15.8M	10.9M	8.46M	6.68M	3.20M	3.04M
eon	2.97M	2.8M	1.27M	0.874M	0.288M	0.282M
gap	5.45M	4.53M	1.43M	0.985M	0.886M	0.744M
gcc	1.69M	1.51M	1.15M	1.07M	1.04M	0.898M
gzip	3.64M	3.46M	2.81M	2.45M	2.30M	2.14M
mcf	7.81M	6.64M	7.31M	5.78M	7.18M	6.32M
parser	22.9M	14.4M	11.2M	10.7M	6.65M	5.65M
perlbmk	1.71M	1.03M	0.571M	0.462M	0.340M	0.309M
twolf	8.95M	6.71M	2.96M	2.07M	1.48M	1.29M
vortex	7.25M	6.53M	4.52M	4.25M	3.47M	3.02M
vpr	6.62M	4.77M	3.41M	2.63M	2.02M	1.67M
PowerStone/Instruction						
adpcm	23392	22204	2824	2691	159	148
bcnt	115	115	58	58	31	30
blit	149	122	75	66	40	37
compress	4435	4054	383	357	199	153
crc	176	147	90	75	49	34
des	23113	21938	5993	5889	146	144
engine	244	226	125	114	65	61
fir	1566	1548	167	167	87	87
g3fax	220	197	112	105	58	52
jpeg	26097	23072	314	286	159	140
pocsag	3730	3221	311	232	148	131
qurt	179	170	91	86	50	47
ucbqsort	30629	28352	166	148	87	78
v42	555022	536798	51230	50613	171	166
SPEC'00/Instruction						
bzip2	8.58M	6.09M	3.78M	3.17M	2.92M	2.83M
Crafty	43.2M	37.5M	23.1M	18.5M	8.77M	8.59M
Eon	6.18M	5.75M	2.63M	2.11M	0.597M	0.550M
Gap	14.4M	10.7M	3.78M	3.59M	2.34M	2.13M
Gcc	3.12M	2.53M	2.13M	2.09M	1.92M	1.77M
Gzip	124M	102M	95.8M	88.1M	78.2M	72.0M
mcf	20.9M	15.5M	19.6M	18.8M	19.2M	19.0M
Parser	65.6M	46.6M	32.2M	28.3M	19.0M	18.1M
Perlbmk	3.78M	2.99M	1.26M	1.05M	0.751M	0.676M
twolf	27.7M	23.0M	9.16M	7.51M	4.56M	4.29M
Vortex	17.9M	14.5M	11.2M	10.8M	8.56M	7.96M
Vpr	15.1M	10.8M	7.75M	6.35M	4.58M	4.49M

Table 7: Cache miss comparison.

On the average, for the data/instruction traces, the improved cache indexing achieved 23%/14%, 19%/10%, and 14%/7.7% reduction in cache misses, for cache configurations *A*, *B*, and *C* respectively, as shown in **Figure 3**. In some cases the reduction in misses was up to 45% for data traces and 31% for instruction traces. For smaller caches, or larger application benchmarks, a larger reduction was observed. The technique benefited data caches more than address caches.



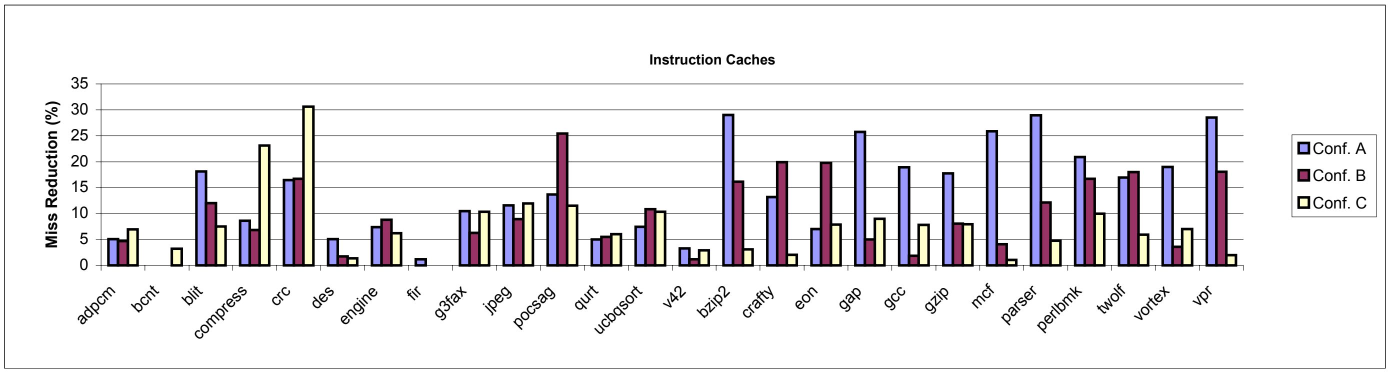


Figure 3: Cache miss reduction using improved indexing.

4. Conclusion

We have proposed a zero cost technique for improving cache performance in embedded systems as well as mobile and portable general-purpose devices that execute a known application set. Our technique involves selecting an optimal set of bits used for indexing into the cache. We have provided an efficient algorithm for computing an optimal indexing scheme. Our heuristic algorithm produces good results, as demonstrated by experiments on a large number of benchmarks.

5. Acknowledgement

This work was supported by the National Science Foundation.

6. References

[1] 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. Conference on HW/SW Codesign, 1999.

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

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

[6] K. Suzuki, T. Arai, N. Kouhei, I. Kuroda. V830R/AV: Embedded Multimedia Superscalar RISC Processor. IEEE Micro, 1998.

[7] P. Petrov, A. Orailoglu. Towards Effective Embedded Processors in Codesigns: Customizable Partitioned Caches. 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 GP Processor Architectures. International Symposium on Microarchitecture, 2000.

[10] D.A. Patterson, J.L. Hennessy. Computer Organization and Design: The Hardware/Software Interface, Second Edition. Morgan Kaufmann, 1997.

[11] P.R. Panda, H. Nakamura, N.D. Dutt, A. Nicolau. Improving Cache Performance through Tiling and Data Alignment. Workshop on Parallel Algorithms for Irregularly Structured Problems, 1997.

[12] A. Vandecappelle, M. Miranda, E. Brockmeyer, F. Catthoor, D. Verkest. Global Multimedia System Design Exploration using Accurate Memory Organization Feedback. Design Automation Conference, 1999.

[13] G. Rivera, C. Tseng. Compiler Optimizations for Eliminating Cache Conflict Misses. Department of Computer Science, Technical Report, UM, 1997.

[14] T.L. Johnson, M.C. Merten, W.W. Hwu. Run-Time Spatial Locality Detection and Optimization. International Symposium on Microarchitecture, 1997.

[15] M. Waldvogel, et al. Scalable High Speed IP Routing Lookups. ACM Special Interest Group on Data Communication, 1997.

[16] S. Nilsson, G. Karlsson. Fast address lookup for Internet routers. IEEE Broadband Communications, 1998.

[17] E. Gurari. An Introduction to the Theory of Computation. Ohio State University, Computer Science Press, 1989.

[18] SPEC CPU 2000. http://www.spec.org.

[19] D. Burger, T.M. Austin, The SimpleScalar Tool Set, Version 2.0. University of Wisconsin-Madison Computer Sciences Department Technical Report #1342, June 1997.