Optimal Indexing for Cache Miss Reduction in Embedded Systems

Tony Givargis

Technical Report CECS-02-10
July 4, 2002

Department of Information and Computer Science
Center for Embedded Computer Systems
University of California, Irvine 92697
givargis@ics.uci.edu

Optimal Indexing for Cache Miss Reduction in Embedded Systems

Tony Givargis

Technical Report CECS-02-10
July 4, 2002

Department of Information and Computer Science
Center for Embedded Computer Systems
University of California, Irvine 92697
givargis@ics.uci.edu

Abstract

The increasing use of microprocessor cores in embedded systems 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, for embedded systems that execute a fixed application, 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 show that this problem belongs to the NP-complete class of problems. Further, we give a heuristic algorithm for selecting the K index bits that is efficient and produces good results. We show the feasibility of our algorithm by applying it to a large number of embedded system applications.

Table of Contents

Abstract		5
Keywords		5
1.	Introduction	5
2.	Cache Indexing	7
2.1	Problem Formulation	7
2.2	NP Completeness	7
2.3	Heuristic Algorithm	9
3.	Experiments	12
4.	Conclusion	15
5.	References	15

List of Figures & Tables

Figure 1: Cache index mapping: (a) traditional approach, (b) our approach. 6
Figure 2: Block diagram of a heuristic algorithm for computing a good cache index set. 9
Figure 3: Correlation measure: (a) A0 used as index, (b) A2 used as index, (c) A0 and A2 used as indices. 10
Figure 4: Percent reduction in cache misses, for data traces, using near-optimal cache indexing. 15
Figure 5: Percent reduction in cache misses, for instruction traces, using near-optimal cache indexing. 16
Table 1: A sample application trace. 8
Table 2: Using set intersections to compute the number of cache misses. 8
Table 3: A sample striped application trace. 10
Table 4: Quality measures. 10
Table 5: Correlation measures. 11
Table 6: Updated quality measures. 12
Table 7: Data trace results for PowerStone benchmark applications. 12
Table 8: Instruction trace results for PowerStone benchmark applications. 13
Table 9: Cache organizations used in experiments. 13
Table 10: Data trace cache misses using traditional index mapping. 13
Table 11: Instruction trace cache misses using traditional index mapping. 14
Table 12: Data trace cache misses using near-optimal index mapping. 14
Table 13: Instruction trace cache misses using near-optimal index mapping. 14

Optimal Indexing for Cache Miss Reduction in Embedded Systems

Tony Givargis
Department of Information and Computer Science
Center for Embedded Computer Systems
University of California, Irvine 92697
givargis@ics.uci.edu

Abstract

The increasing use of microprocessor cores in embedded systems 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, for embedded systems that execute a fixed application, 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 show that this problem belongs to the NP-complete class of problems. Further, we give a heuristic algorithm for selecting the K index bits that is efficient and produces good results. We show the feasibility of our algorithm by applying it to a large number of embedded system applications.

Keywords

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

1. Introduction

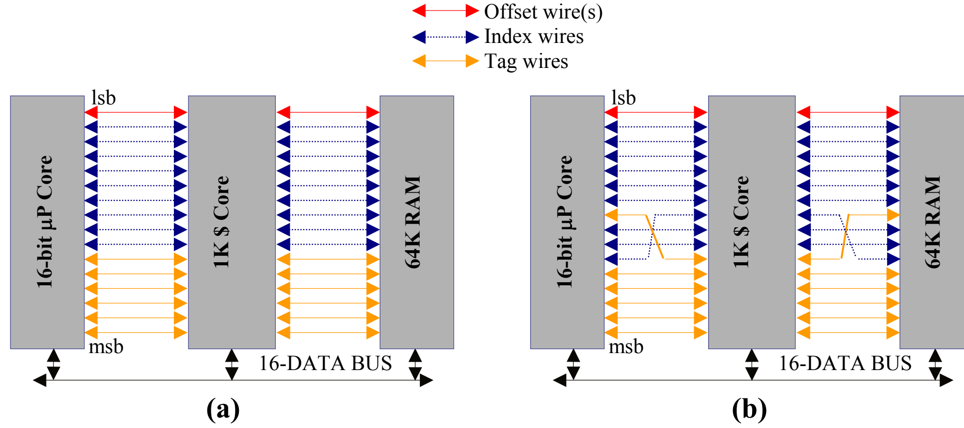
The growing demand for embedded computing devices and the shrinking time-to-market windows are leading to new core based system-on-a-chip (SOC) architectures intended for embedded systems [1][2]. Specifically, microprocessor cores (a.k.a., embedded processors) are playing an increasing role in embedded system design [3][4]. 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 embedded systems, 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 systems, optimizing of the processor cache hierarchy has received a lot of attention from the research community [5][6][7]. This is in part due to the large performance gained by tuning caches to the application set of the embedded system. 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.

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 [8]. 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.

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



We pictorially depict the idea of optimal index mapping 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 index mapping 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.

$$L = h(A)$$

Where h is the arbitrary hash function. While it may be possible to compute a perfect hash function, given the cache organization and a trace capturing the application memory behavior, 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 performance [9][10][11][12]. 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 limitation 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 [13][14]. 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 our problem, show that it belongs to the NP-complete class of problems, and give a heuristic algorithm that is efficient and produces good results. In Section 3, we give our experimental results. In Section 4, we state our concluding remarks.

2. 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 an M=32-bit processor connected to a cache of size S=8192 bytes that is A=2-way set associative and has line size equal to L=4 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 and thus most probably cannot be solved efficiently (i.e., in polynomial time).

2.2 NP Completeness

The stated problem of optimal cache indexing belongs to the class NP-complete. Here, we first show that the problem belongs to the class NP. Then, we show that the set intersection problem [15], a known NP-complete problem, is polynomial time reducible to the problem of optimal cache index mapping. Since the set intersection problem is NP-complete, 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 processes, 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 can show that the set intersection problem is reducible to the problem of optimal cache indexing. The set intersection problem is as follows. We are given a collection of sets S_1, S_2, … S_K and an integer m. The problem is to find a subset C of the sets 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, let us 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; in other words, U contains the memory references that are unique in the input trace. 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 ith address bit used as the cache index. We illustrate this with an example. Consider the memory reference 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.

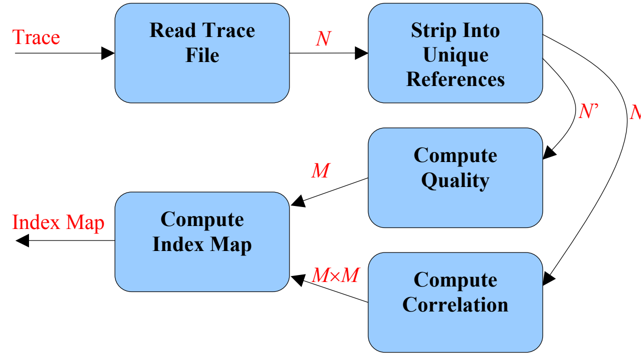
The unique set U and the conflict sets X_0, X_1 … X_{M-1} fully capture the information content of the trace necessary to compute cache performance for any arbitrary configuration of the cache. The number of cache misses for a cache of depth two and associativity of one, using A_i as the index bit, is 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 Number of 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 and X_1 and X_2	$X_0 \cap X_1 \cap X_2 = \varnothing$	3 + 0 = 3

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

Note that 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.

Figure 2: Block diagram of a heuristic algorithm for computing a good cache index set.



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

Here, bottom up, we fist 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 our set theoretic cache miss calculation, 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 our cardinality function given earlier. This is an identical problem to the NP-complete set intersection problem stated earlier. Indeed, we have to construct, in reverse, a set of binary numbers, namely a trace of references, of width equal to the number of sets in our set intersection problem instance. This process requires a large number of straightforward transformations that we have left out for brevity.

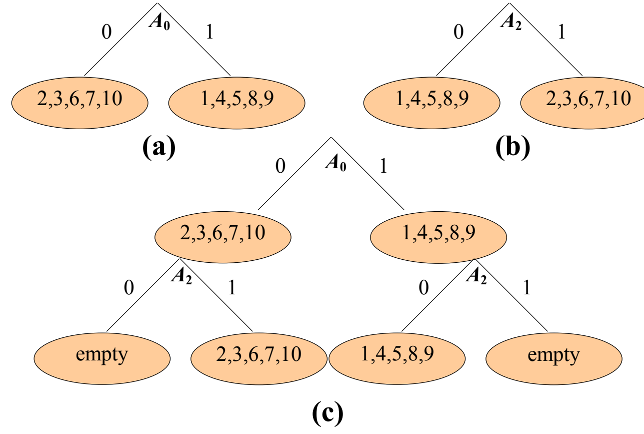
2.3 Heuristic Algorithm

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. A block diagram of our algorithm is shown in Figure 2. The first step of the algorithm is simply reading a trace into memory. In Figure 2, we denote the size of the trace as N. The next step is to strip from the trace just the unique references[1], denoted as N' in Figure 2, where $N' \leq N$. 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

[1] In an actual implementation, to avoid memory overflow, we do not read the entire trace into memory, instead we read the elements one by one keeping only the unique references.

Figure 3: Correlation measure: (a) A_0 used as index, (b) A_2 used as index, (c) A_0 and A_2 used as indices.



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 to have the values shown in Table 4.

Q_5	Q_4	Q_3	Q_2	Q_1	Q_0
1/4	3/7	1	1	2/3	1

Table 4: Quality measures.

As an example, looking at Table 3 and for Q_4, along the A_4^{th} column, there are seven bits that are zero and three bits that are one, 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 with an example, in Figure 3(a) we pictorially depict how A_0 would split the trace shown in Table 3. Likewise, in Figure 3(b) we show how A_2 would split the same trace. Note that according to our quality measure, both A_0 and A_2 would be ideal bits to use in a cache of depth two, as they split the trace equally into the two locations of the cache. 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 3(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.

$$C_{ij} = \frac{\min(E_{ij}, D_{ij})}{\max(E_{ij}, D_{ij})} \quad where$$

$$E_{ij} : the\ number\ of\ refereces\ having\ identical\ bits\ at\ A_i\ and\ A_j$$

$$D_{ij} : the\ number\ of\ refereces\ having\ different\ bits\ at\ A_i\ and\ A_j$$

We give the complete correlation measures for our sample trace 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 and 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.

$$E_{23} = 4$$

$$D_{23} = 6$$

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

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: $Q_0, Q_1 \ldots Q_{M\text{-}1}$

Input: $C_{00}, Q_{01} \ldots C_{(M\text{-}1)\times(M\text{-}1)}$

Output: an ordering of $A_0, A_1 \ldots A_{M\text{-}1}$

```
loop
   select Ab corresponding to Qb = min{ Q0, Q1 … QM-1 }
   for each Qi ∈ { Q0, Q1 … QM-1 }
      Qi := Qi × Cbi
   halt when each Ai has been selected
endloop
```

The algorithm works as follows. We repeatedly select an address bit with the highest corresponding quality measure and then update the quality measures using the correlations. As an 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[2]. Then, the algorithm updates the quality measures Q_i by multiplying with C_{0i}, to obtain a new set of quality measures, as shown in Table 6.

Q_5	Q_4	Q_3	Q_2	Q_1	Q_0
1/4	6/21	2/3	0	2/27	0

Table 6: Updated quality measures.

Next, having the largest quality measure, the algorithm selects A_3, and update the quality measures again, and so on. When the algorithm terminates, we obtain the following ordering of the address bits.

$$A_0, A_3, A_5, A_4, A_1, A_2$$

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. To build a cache of depth eight we choose A_0, A_3, and A_5, etc. From this point on, we refer to the problem of optimal cache indexing as *the problem of near-optimal cache indexing*.

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. Stripping 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 or 64, and thus is assumed to be a constant.

3. Experiments

For our experiments, we have used 14 typical embedded system applications that are part of the *PowerStone* benchmark applications [3]. The applications include a JPEG image 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*, 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, instrumented to output memory reference traces for both instruction and data accesses. Then, we ran the traces through our heuristic algorithm to obtain the near-optimal cache index sets. Our results are summarized in Table 7 and Table 8.

Benchmark	#Refs	#Unique Refs	Near-Optimal Index Map
adpcm	18431	381	4,6,8,9,5,7,12,10,11,13,2,3,14,22,0,1,15,16,17,18,19,20,21,23,24,25,26,27,28,29,30,31
bcnt	456	162	6,4,5,9,7,8,14,15,10,3,2,11,22,12,0,1,13,16,17,18,19,20,21,23,24,25,26,27,28,29,30,31
blit	4088	2027	4,5,14,6,7,8,9,10,11,12,13,15,2,3,16,22,0,1,17,18,19,20,21,23,24,25,26,27,28,29,30,31
compress	58250	8906	6,9,8,5,4,7,12,10,14,11,13,3,16,15,2,17,18,22,0,1,19,20,21,23,24,25,26,27,28,29,30,31
crc	2826	603	4,7,6,3,5,9,11,8,10,2,22,12,0,1,13,14,15,16,17,18,19,20,21,23,24,25,26,27,28,29,30,31
des	20162	2241	5,4,7,8,6,9,10,14,11,15,12,2,3,13,22,0,1,16,17,18,19,20,21,23,24,25,26,27,28,29,30,31
engine	211106	225	4,10,17,7,9,5,8,6,3,2,11,22,0,1,12,13,14,15,16,18,19,20,21,23,24,25,26,27,28,29,30,31
fir	5608	146	7,4,5,8,6,2,9,10,11,22,12,3,0,1,13,14,15,16,17,18,19,20,21,23,24,25,26,27,28,29,30,31
g3fax	229512	3781	7,2,4,3,6,22,12,8,5,9,10,11,15,13,14,16,0,1,17,18,19,20,21,23,24,25,26,27,28,29,30,31
jpeg	1311693	39302	8,4,6,5,7,10,11,9,12,14,3,13,15,16,17,18,2,19,22,0,1,20,21,23,24,25,26,27,28,29,30,31
pocsag	13467	515	4,7,5,6,2,10,8,3,9,11,12,13,22,0,1,14,15,16,17,18,19,20,21,23,24,25,26,27,28,29,30,31
qurt	503	84	4,10,5,6,7,8,9,11,15,2,3,22,0,1,12,13,14,16,17,18,19,20,21,23,24,25,26,27,28,29,30,31
ucbqsort	61939	1144	6,5,8,9,10,4,7,11,16,19,2,3,12,13,22,0,1,14,15,17,18,20,21,23,24,25,26,27,28,29,30,31
v42	649168	23942	6,9,4,7,5,8,10,11,12,13,3,15,2,14,16,17,18,22,0,1,19,20,21,23,24,25,26,27,28,29,30,31

Table 7: Data trace results for *PowerStone* benchmark applications.

[2] When there is a tie in the quality measure, we select the less significant bit first.

Table 7 shows the results for the data traces. The first column of the table gives the application name, the second column of the table gives the total number of references, and the third column of the table gives the number of unique references. The last column of the table gives the address bit ordering output by our heuristic algorithm.

Benchmark	#Refs	#Unique Refs	Near-Optimal Index Map
adpcm	63255	611	2,3,8,5,7,4,6,9,12,10,11,0,1,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31
bcnt	1337	115	2,3,4,5,6,7,8,11,9,0,1,10,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31
blit	22244	149	2,3,4,5,10,7,8,9,11,12,0,1,6,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31
compress	137832	731	3,2,7,4,11,5,8,6,10,9,12,0,1,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31
crc	37084	176	2,3,4,6,11,7,9,10,12,8,5,0,1,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31
des	121648	570	2,3,7,4,5,8,12,9,10,11,13,6,0,1,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31
engine	409936	244	2,3,4,5,7,10,8,6,11,12,13,9,0,1,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31
fir	15645	327	7,2,3,8,4,5,6,9,11,12,13,10,0,1,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31
g3fax	1127387	220	2,4,3,6,5,8,7,9,12,13,11,10,0,1,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31
jpeg	4594120	623	2,3,5,4,8,6,7, 13,14,10,9,11,12,0,1,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31
pocsag	47840	560	2,6,3,5,4,10,9,8,7,11,13,12,0,1,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31
qurt	1044	179	2,3,5,4,8,6,10,9,7,11,13,12,0,1,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31
ucbqsort	219710	321	2,3,5,4,6,12,13,8,7,10,9,11,0,1,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31
v42	2441985	656	2,3,8,12,13,5,6,4,7,9,11,10,0,1,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31

Table 8: Instruction trace results for *PowerStone* benchmark applications.

Table 8 shows the results for the instruction traces and is similar to Table 7 in content.

To measure the performance benefit of near-optimal index mapping, we have simulated the traces under three different cache organization (i.e., configuration) schemes, as shown in Table 9.

Configuration	I/D$ Size	I/D$ Line	I/D$ Assoc	Cache Depth / #Index Bits
1	4096	4	1	1024 / 10
2	8192	8	2	512 / 9
3	16384	16	4	256 / 8

Table 9: Cache organizations used in experiments.

For each of the cache configurations shown in Table 9, we have measured the number of misses when traditional cache indexing is used as well as when near-optimal cache indexing, as computed by our heuristic and shown in the Table 7 and Table 8, is used. The results are reported in Table 10, Table 11, Table 12, and Table 13.

Benchmark	Configuration 1	Configuration 2	Configuration 3
adpcm	5193	2181	621
bcnt	164	156	147
blit	4034	4025	4038
compress	12659	9603	7861
crc	694	485	228
des	15155	12849	10523
engine	7131	3482	132
fir	658	139	136
g3fax	127828	65143	35158
jpeg	267567	169490	79258
pocsag	1238	530	268
qurt	115	77	73
ucbqsort	10862	3309	804
v42	157469	111108	87592

Table 10: Data trace cache misses using traditional index mapping.

Table 10 gives, for data traces, the number of cache misses for each of the three configurations given in Table 9. Here, traditional cache indexing is used.

Benchmark	Configuration 1	Configuration 2	Configuration 3
adpcm	23392	2824	159
bcnt	115	58	31
blit	149	75	40

compress	4435	383	199
crc	176	90	49
des	23113	5993	146
engine	244	125	65
fir	1566	167	87
g3fax	220	112	58
jpeg	26097	314	159
pocsag	3730	311	148
qurt	179	91	50
ucbqsort	30629	166	87
v42	555022	51230	171

Table 11: Instruction trace cache misses using traditional index mapping.

Table 11 gives, for instruction traces, the number of cache misses for each of the three configurations given in Table 9. Here, traditional cache indexing is used.

Benchmark	Configuration 1	Configuration 2	Configuration 3
adpcm	4175	1813	542
bcnt	167	154	140
blit	3022	3078	3106
compress	7772	6414	5671
crc	416	303	154
des	13360	12239	10179
engine	4479	2277	94
fir	637	140	134
g3fax	92503	48855	26940
jpeg	191542	129399	61757
pocsag	757	355	192
qurt	98	68	65
ucbqsort	7955	2463	643
v42	150021	107441	89942

Table 12: Data trace cache misses using near-optimal index mapping.

Table 12 gives, for data traces, the number of cache misses for each of the three configurations given in Table 9. Here, near-optimal cache indexing is used.

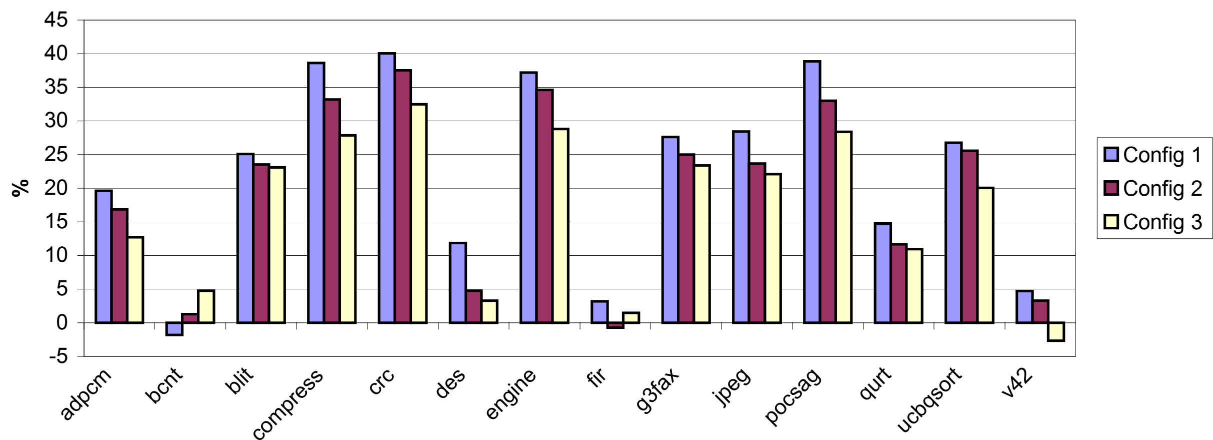
Benchmark	Configuration 1	Configuration 2	Configuration 3
adpcm	22204	2691	148
bcnt	116	58	30
blit	122	66	37
compress	4054	357	153
crc	147	75	34
des	21938	5889	144
engine	226	114	61
fir	1548	167	87
g3fax	197	105	52
jpeg	23072	286	140
pocsag	3221	232	131
qurt	170	86	47
ucbqsort	28352	148	78
v42	536798	50613	166

Table 13: Instruction trace cache misses using near-optimal index mapping.

Table 13 gives, for instruction traces, the number of cache misses for each of the three configurations given in Table 9. Here, near-optimal cache indexing is used.

The percent reduction in cache misses is shown graphically in Figure 4 and Figure 5. On the average, for the data traces, near-optimal index mapping achieved 22%, 20%, and 17% reduction in cache misses, for cache configurations one, two, and three respectively. On the average, for the instruction traces, near-optimal index mapping achieved 8%, 8%, 7% reduction in cache misses, for cache configurations one, two, and three respectively. In some cases the reduction in misses is up to 40% for data traces and 30% for instruction traces. We note that, for smaller caches, or larger application benchmarks, a larger reduction is observed. Likewise, the

Figure 4: Percent reduction in cache misses, for data traces, using near-optimal cache indexing.



technique benefits data caches more than address caches, as data access patterns tend to have a larger degree of non-sequential nature as compared to the highly sequential instruction accesses. We note that in a few cases, our heuristic computed an index map that resulted in a slight increase in cache misses. Specifically, in the case of *bcnt*, *fir*, and *v42*, a less than 2% increase in cache misses was observed.

4. Conclusion

We have proposed a zero cost technique for improving cache performance in embedded systems. Our technique involves selecting an optimal set of bits used for indexing 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. In our approach, we propose an optimal set of K index bits. We have shown that the problem of optimal cache indexing belongs to the class NP-complete. We have provided an $O(N\times\log(N))$ algorithm, where N is the number of references in the trace of the application set that the cache is being optimized for. Our heuristic algorithm produces good results, as demonstrated by experiments on a large number of embedded system applications. We have experimentally shown that for the *PowerStone* benchmark applications, and three typical ache configurations, our heuristic reduces the number of cache misses an average of 20% for data and 8% for instruction caches.

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] 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.

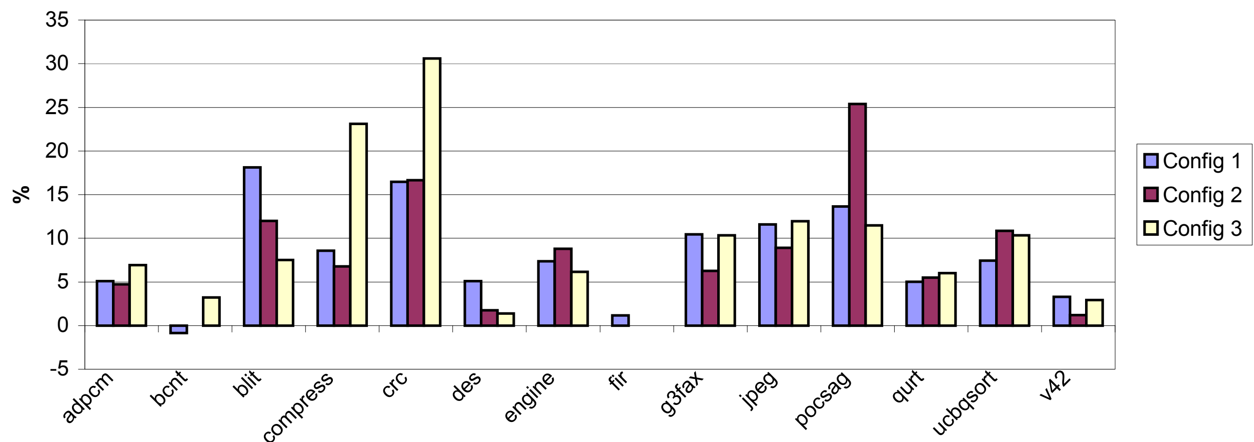
[4] K. Suzuki, T. Arai, N. Kouhei, I. Kuroda. V830R/AV: Embedded Multimedia Superscalar RISC Processor. IEEE Micro, vol. 18, No. 2, pp.36-47, 1998.

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

[6] 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.

[7] 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.

Figure 5: Percent reduction in cache misses, for instruction traces, using near-optimal cache indexing.



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

[9] 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.

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

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

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

[13] M. Waldvogel, G. Varghese, J. Turner, B. Plattner. Scalable High Speed IP Routing Lookups. ACM Special Interest Group on Data Communication, 1997.

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

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