Zero Cost Indexing for Improved Processor Cache Performance

TONY GIVARGIS
University of California, Irvine

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 a near-optimal set of bits used as index into the cache. This technique does not add any overhead in terms of area or delay. In this article, we present 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. Specifically, for data traces, we show up to 45% reduction in cache misses. Likewise, for instruction traces, we show up to 31% reduction in cache misses. When a unified data/instruction cache architecture is considered, our results show an average improvement of 14.5% for the Powerstone benchmarks and an average improvement of 15.2% for the SPEC'00 benchmarks.

Categories and Subject Descriptors: B.3.0 [**Memory Structures**]: General

General Terms: Design, Performance

Additional Key Words and Phrases: Cache optimization, design 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 [ITRS 2005; Kozyrakis and Patterson 1998; Vahid and Givargis 1999]. Specifically, microprocessor cores (a.k.a., embedded processors) are playing an increasing role in such systems' design [Wong et al. 2004]. 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,

This work was supported by the National Science Foundation (NSF) award number CCR-0205712.
Author's address: Department of Computer Science, Center for Embedded Computer Systems, University of California, Irvine, Irvine, CA 92697; email: givargis@uci.edu.

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. Specifically, if a cache is designed to reduce the total number of cache misses during the execution of an application, the bit-traffic to and from the off-chip memory will be reduced. Since the capacitive load of a communication bus crossing the chip boundary is relatively large, a reduction in bit-traffic over such a bus will yield a reduction in power consumption and overall energy use of the application.

Consequently, in embedded, mobile, and handheld devices, optimizing of the processor cache hierarchy has received a lot of attention from the research community [Balasubramonian et al. 2000; C.L and Despain 1995; Malik et al. 2000; Petrov and Orailoglu 2001; Suzuki et al. 1998]. This is due in part 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 a near-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[1] of the cache [Patterson and Hennessy 1997]. 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, as well as board-based design.

We pictorially depict the idea of near-optimal 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, say, a near-optimal

[1]The *depth of a cache*, in terms of the number of index bits K, is defined to be 2^K (i.e., the number of lines in a cache). Note that a line may contain one or more data sets, depending on the degree of associativity of the cache.

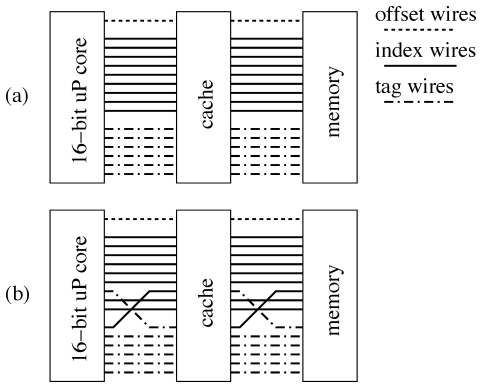

Fig. 1. Index mapping example.

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 intuition behind our work is simple. We exploit the fact that, for embedded applications (where the execution pattern is mostly constant), there is an opportunity to use alternate index bits to achieve better cache performance without increasing cache size or degree of associativity.

The problem of cache indexing is one of hashing. In traditional cache design, reference A maps to cache location L, using the hash function shown in Eq. (1):

$$L = A \pmod{D}. \tag{1}$$

In Eq. (1), D is the depth of the cache. In general, data can be mapped onto a cache using the generic hash function shown in Eq. (2)

$$L = \mathcal{H}(A). \tag{2}$$

In Eq. (2), $\mathcal{H}$ 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 a 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.

The remainder of this article is organized as follows: In Section 2, we summarize related work. In Section 3, we formulate the problem and give our heuristic solution. In Section 4, we state our experiments. In Section 5, we state our concluding remarks and define some possible future directions along the lines of the proposed work.

2. PREVIOUS WORK

While the problem of hashing is a well-studied topic in the general area of computer science [Corman et al. 2001], we are unaware of any direct research related to a hardware approach to processor cache indexing, as stated in this work. However, there are a number of compiler (software) techniques that aim at achieving similar goals as ours. In this section we briefly elaborate on some of these techniques.

Rivera and Tseng [1997] present two data only compiler transformations to eliminate conflict misses. One of these transformations is to modify variable base addresses. The other transformation is to pad inner array dimensions. Unlike compiler transformations that restructure the application code, these two techniques modify the application data layout in order to improve cache performance.

Panda et al. [1997] present a data alignment technique that pads arrays, to improve program performance through minimizing cache conflict misses. They also describe algorithms for selecting tile sizes for maximizing data cache utilization, and computing pad sizes for eliminating self-interference conflicts in the chosen tiles.

Abella et al. [2002] exploit the effectiveness of the memory hierarchy by means of program transformations (code segment), such as padding, to reduce conflict misses. They present an approach to perform near-optimal code segment padding for a system with multilevel caches by analyzing programs, detecting conflict misses (by means of cache miss equations), and using a genetic algorithm to compute the transformations.

Huang et al. [2003] propose loop (code) and data tiling for improving data locality in partial-differential equation (PDE) solvers running on single processor systems with a memory hierarchy. They combine loop (code) tiling with array layout transformation in such a way that a significant amount of cache misses that would otherwise be present are eliminated. They compare their results to nine existing loop tiling algorithms and show that their techniques delivers impressive performance speedups (faster by factors of 1.55–2.62) and smooth performance curves across a range of problem sizes on representative machine architectures.

Ding and Kennedy [1999] explore ways to achieve better data reuse of dynamic applications by improving both code and data locality in irregular programs. They demonstrate that runtime program transformations can substantially improve code and data locality despite the added complexity and cost. They utilize a compiler to automate such transformations, eliminating much of the associated runtime overhead.

Grun et al. [2000, 2001] describe a memory-aware compiler approach that exploits efficient memory access modes by extracting accurate timing information, allowing the compiler's scheduler to perform global code reordering to better hide the latency of memory operations. In this context, memory access modes include page mode, burst mode, and pipelined access. In their work, they also assume a more application specific, tuned, memory subsystem that may include one or more caches and memories organized to meet low power or high performance demands in addition to the traditional memory/cache hierarchy. Further, they present a compiler technique that in the presence of caches actively manages cache misses, and performs global miss traffic optimizations, to better hide the latency of the memory operations.

Software-based cache-sensitive compiler techniques, such as those outlined above, use code motion, data realignment, and data padding as a mechanism to improve cache performance. With respect to our approach, these techniques: (1) have a limited degree of freedom in moving code or data (e.g., code segment can

only be moved at the basic block boundary or data objects may only be moved in a non-overlapping way), (2) may incur added cost (e.g., consecutive basic block segments, if split would require an added jump instruction or padded arrays may require added index computation overhead), (3) have a limited degree of freedom in padding arrays and structures (e.g., a structure object can only be padded at the element boundary to comply with the source programing language semantics or multidimensional arrays usually are padded at the row or column boundary but not both), (4) are targeted for compute-intensive loops or nested loops operating on large arrays, and (5) have a limited view of the overall software (e.g., do not consider multiple concurrent tasks or interleaved operating system activity.) We note that our approach can be applied over and beyond these software-based compiler techniques for added benefit.

3. 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.

3.1 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 shown in Eq. (3)

$$K = \log_2\left(\frac{S}{L \times A}\right). \tag{3}$$

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. The number of combinations, thus, is computed as shown in Eq. (4).

$$\binom{M}{K} = \frac{M!}{K! \times (M-K)!}. \tag{4}$$

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. We note that a trace of memory references will be different during different runs of a program, if the input to the program changes. For embedded systems, we assume a *representative trace* to be one that is obtained by concatenating traces obtained from different runs of the program using typical input data. Our technique can be applied to the representative trace in order to arrive at an indexing scheme that is ideal for the kinds of input the device may receive.

In an exhaustive approach, one can find an optimal set of index bits 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 very large for all interesting cases. For example, assume a 32-bit processor connected to an 8192 bytes two-way set associative cache with line size equal to four bytes. Using Eq. (3), we compute $K = 10$. Further, using Eq. (4), we compute the number of possible cache index sets to be over 64 million.

3.2 NP-Completeness

We show that the problem of *optimal cache indexing* belongs to the NP-complete class of problems. Our proof is by reduction from the *set cover* NP-complete problem [Gurari 1989]. In the *set cover* problem, we are given a set of sets S over some universe U and a number k. We compute *yes* if some k-sized subfamily C of S has the same union as U and *no* otherwise. Recall that in the *optimal cache indexing* problem, we have a sequence of memory addresses T, a number k, and a number m. We compute *yes* if it is possible to choose k of the memory address bits so that a direct mapped cache with size 2^k causes m cache misses on the sequence T and *no* otherwise. These two problems are defined as follows:

Set Cover Instance:
Input: $S = \{S_1, S_2 ... S_n\}$ over the universe $U = \{1, 2 ... l\}$, that is, $S_i \subseteq U$
Input: An integer k
Question: Does $C = \{C_1, C_2 ... C_k\}$ $(C_i \in S)$ exist such that $\bigcup_{i=1..k} C_i = U$?

Optimal Cache Indexing Instance:
Input: A sequence of memory addresses T
Input: Integers k and m
Question: Does there exist k index bits in a direct mapped 2^k cache causing m misses on T?

THEOREM 1. *Optimal cache indexing* $\in$ *class NP-complete.*

PROOF. In our proof strategy, we first show that the problem of *optimal cache indexing* belongs to the NP class of problems, that is, *optimal cache indexing* $\in$ NP. Then, we show that the problem of *set cover* is reducible to the problem of *optimal cache indexing* in linear time, that is, *set cover* $\leq_P$ *optimal cache indexing*.

To show that *optimal cache indexing* $\in$ NP, we nondeterministically select k bits as the cache index set, configure a 2^k direct mapped cache simulator accordingly, and simulate the sequence of memory addresses T. If the number of cache misses is m we halt and output *yes*; otherwise, we halt and output *no*.

To show that *optimal cache indexing* $\in$ NP-hard, we show that *set cover* $\leq_P$ *optimal cache indexing*. The reduction is as follows. Let $S = \{S_1, S_2, \ldots, S_n\}$ be the set of sets (over a universe U) from the *set cover* instance. From this *set cover* instance, we construct an instance of the *optimal cache indexing* in which the memory addresses are n-bit wide. Furthermore, each bit of a memory address corresponds to one of the members of S (i.e, the least significant bit of a memory address corresponds to S_1, the next least significant bit of a memory

Table I. Constructing a Sequence of Memory Addresses (i.e., T)

$x \in U$	Address
a_1	"0000"
b_1	"0001"
a_1	"0000"
a_2	"0001"
b_2	"1011"
a_2	"0001"
a_3	"0010"
b_3	"0111"
a_3	"0010"
a_4	"0011"
b_4	"1111"
a_4	"0011"

address corresponds to S_2 and so on). For each set member x in the *set cover* instance (i.e., $x \in U$), we create a sequence of three addresses $a_x \rightarrow b_x \rightarrow a_x$, where a_x and b_x differ in exactly those bit positions corresponding to sets containing x. We choose these addresses so that all addresses a_x, a_y, b_x, and b_y are distinct for distinct $x \in U$ and $y \in U$. The overall sequence of addresses (i.e., T) is formed by concatenating together all these triples. Finally, we choose k to be the same as that from the *set cover* instance (i.e., the number of subsets) and $m = 2/3 \times |T|$ (i.e., $m = 2 \times$ *the-number-of-triples*). This completes the reduction.

By simulating the *optimal cache indexing* instance, and within each triple, we will either get two cache misses (the first a_x and b_x), or three (also the second a_x). We get two misses if and only if the k chosen address bits include one corresponding to a set that covers x, so that a_x and b_x land in different cache entries. Therefore, there exists a set of k index bits causing m misses if and only if there exists a size-k set cover. □

Let us give a simple example to highlight the reduction outlined in our proof. Consider the following instance of the set cover problem.

Set Cover Instance:

Input: $U = \{1, 2, 3, 4\}$

Input: $S = \{S_1, S_2, S_3, S_4\}$ where $S_1 = \{1, 3\}$, $S_2 = \{2\}$, $S_3 = \{3, 4\}$, and $S_4 = \{2, 4\}$

Input: $k = 2$

Question: Does $C = \{C_1, C_2\}$ $(C_1, C_2 \in S)$ exist such that $C_1 \cup C_2 = U$?

In accordance with the reduction outlined in the proof, we create an instance of the *optimal cache indexing* problem as follows. For the reduction, we choose the address width to be 4 bits (i.e., same as $|S|$). For each member $x \in U$, we construct a sequence of three addresses $a_x \rightarrow b_x \rightarrow a_x$, as shown in Table I. Here, we begin by assigning unique binary values to each of a_1, a_2, a_3, and a_4. Then, we obtain the memory addresses b_x from the associated a_x values

by flipping the bit positions corresponding to sets where x is a member. For example, since $x = 3$ is a member of S_1 and S_3, we need to make b_3 different from a_3 in exactly bit positions zero and two. We repeat this for all members of U and obtain a trace T of size 12. We set $m = 2/3 \times |T| = 8$. When done, we arrive at the following instance of the *optimal cache indexing* problem.

Optimal Cache Indexing Instance:
Input: T as shown in Table I
Input: $k = 2$ and $m = 8$
Question: Does there exist $k = 2$ index bits in a direct mapped 2^2 cache causing $m = 8$ misses on T?

Let us now select bits zero and one as the index into our cache. Here, we note that for the triples corresponding to $x = 1, 2, 3$ we get exactly two misses each. In other words, $a_1/a_2/a_3$ will be a miss, $b_1/b_2/b_3$ will be a miss, but the second occurrence of $a_1/a_2/a_3$ will be a hit (this is because at least one of the selected index bits will differentiate between the a and b values). However, for the triples corresponding to $x = 4$ we get exactly three misses (this is because the selected index bits will not differentiate between a_4 and b_4). Thus, in this case, the total misses will be $m = 9$. In fact, the only way to arrive at $m = 8$ would be to select a pair of index bits that differentiate between all pairs of a and b. In other words, cover the universe U in the *set cover* problem.

3.3 Heuristic Algorithm

Since the problem of optimal cache indexing is NP-complete, we give a heuristic algorithm that is efficient and performs well when executed on a large number of typical embedded applications. 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 $N' \leq N$. 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 measure of zero would indicate that the bit, if used as an index into a cache of depth two (i.e., a cache with two addressable data locations), would be a poor choice, as it would place all the references into a single location in the cache, thus causing the most number of conflicts. On the other hand, having a quality measure 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 the references among the two cache locations, causing the least number of conflicts. We compute the quality measure Q_i for address bit A_i by taking the ratio of zeros and ones along the A_i^{th} column, as shown in Eq. (5):

$$Q_i = \frac{\min(Z_i, O_i)}{\max(Z_i, O_i)}. \tag{5}$$

In Eq. (5), Z_i denotes the number of references having the value zero at address bit A_i and O_i denotes the number of references having the value one

Table II. Running Example: A Striped Trace File

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 III. Running Example: Quality Measures

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

at address bit A_i. We further illustrated the concept of quality measure with a running example. Consider the striped trace shown in Table II.

Applying Eq. (5) to our running example, we compute the quality measures shown in Table III.

In a particular example, Eq. (6) illustrates the computation of the quality measure Q_4, corresponding to the address bit A_4. Note that $Z_4 = 7$ is obtained by counting the zeros along the A_4th column of Table II and $O_4 = 3$ is obtained by counting the ones along the A_4th column of Table II.

$$\begin{aligned} Z_4 &= 7 \\ O_4 &= 3 \\ Q_4 &= \frac{\min(7,3)}{\max(7,3)} = \frac{3}{7}. \end{aligned} \tag{6}$$

Next, 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 II. Note that according to our quality measure, both A_0 and A_2 are ideal indices to use in a cache of depth two (i.e., a cache with two addressable data locations). Now consider the case where we have a cache of depth four (i.e., a cache with four addressable data locations), 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 addressable data locations, only two slots receive the references, and the other two slots remain unused. The reason for this is that A_0 and A_2 are correlated. From looking at the trace, we can see that A_2 is simply the complement of A_0. In such a case, we would have a correlation

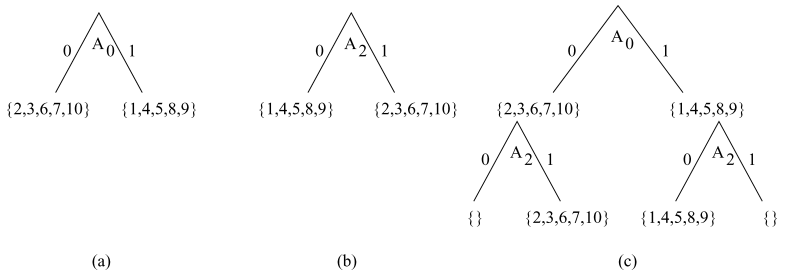

Fig. 2. Correlation Measures: (a) A_0 Used as Index, (b) A_2 Used as Index, and (c) A_0 and A_2 Used as Indices.

Table IV. Running Example: Correlation Measures

	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

measure $C_{i,j}$ equal to zero. In general, we can compute the correlation $C_{i,j}$, for address bits A_i and A_j as shown in Eq. (7).

$$C_{i,j} = \frac{\min(E_{i,j}, D_{i,j})}{\max(E_{i,j}, D_{i,j})}. \tag{7}$$

In Eq. (7), $E_{i,j}$ denotes the number of references having identical values at address bits A_i and A_j. Likewise, $D_{i,j}$ denotes the number of references having different values at address bits A_i and A_j. Applying Eq. (7) to our running example, we compute the correlation measures shown in Table IV.

In a particular example, Eq. (8) illustrates the computation of the correlation measure $C_{2,3}$, corresponding to the address bits A_2 and A_3. Note that $E_{2,3} = 4$ is obtained by counting the number of bits that have the same values along the A_2nd and A_3rd columns of Table II. Likewise, $D_{2,3} = 6$ is obtained by counting the number of bits that have different values along the A_2nd and A_3rd columns of Table II

$$\begin{aligned} E_{2,3} &= 4 \\ D_{2,3} &= 6 \\ C_{2,3} &= \frac{\min(4,6)}{\max(4,6)} = \frac{2}{3}. \end{aligned} \tag{8}$$

In the last step of the algorithm, we use the quality measures along with the correlation measures to compute the near-optimal index ordering as shown in Algorithm 1.

Algorithm 1 repeatedly selects an address bit with the highest corresponding quality measure and then updates the quality measures using the correlation measures. For the running example shown in Table II and quality/correlation

Algorithm 1. Computes Near-optimal Index Ordering

```
1: Input: M {the size of the address space}
2: Input: Q_{M-1}...Q_0 and C_{M-1,M-1}...C_{0,0} {quality and correlation measures}
3: S ← ∅
4: for i ← 0 to M − 1 do
5:    A_best ← max(Q_{M-1}...Q_0)|A_best ∉ S
6:    S ← S ∪ A_best
7:    for j ← 0 to M − 1 do
8:       Q_j ← Q_j * C_{best,j}
9:    end for
10:   print A_best
11: end for
```

Table V. Running Example: Quality Measures Recomputed After the First Iteration of Algorithm 1

Q_5	Q_4	Q_3	Q_2	Q_1	Q_0
1/4*1 = 1/4	3/7*2/3 = 6/21	1*2/3 = 2/3	1*0 = 0	2/3*1/9 = 2/27	1*0 = 0

measures computed in Table III and Table IV, Algorithm 1 first selects A_0 as the best index bit (ties are broken in an arbitrary manner) and updates the quality measures $Q_{M-1} \cdots Q_0$ by multiplying with $C_{0,M-1}, C_{0,M-2} \cdots C_{0,0}$ to obtain a new set of quality measures, as illustrated in Table V. Next, having the largest quality measure, the algorithm selects A_3, and updates the quality measures again, etc. On termination, Algorithm 1 prints A_0, A_3, A_5, A_4, A_1, and A_2, in that order. This ordering defines a near-optimal solution to the problem of cache indexing. Subsequently, to build a cache of depth two (i.e., a cache with two addressable data locations) we choose A_0. To build a cache of depth four (i.e., one with four addressable data locations) we choose A_0 and A_3, and so on.

3.4 Time Complexity and Further Remarks

In terms of running time complexity, our heuristic takes $O(N \times \log_2 N)$ to execute, where N is the size of the original memory trace. This time complexity is computed as follows. Reading the trace takes $O(N)$, as the length of the original trace is N. Reducing the trace down to only the unique references involves what amounts to sorting the trace and thus takes $O(N \times \log_2 N)$. 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 (i.e., Algorithm 1) takes $O(M^2)$ where M is the size of the processor address space (i.e, the width of the address bus). Since, in most instances M is a small integer like 16, 32, or 64, we assume it to be constant. Therefore, Algorithm 1 executes in $O(1)$ time. Thus, the overall heuristic algorithm executes in time $O(N \times \log_2 N)$.

In an implementation of the above-mentioned heuristic, the entire memory trace does not need to be loaded into memory prior to striping it down to the unique references. Instead, a set data structure can be used to track unique references during a single pass over the original trace. Thus, the space complexity of our heuristic is bounded by $O(N')$, where $N' \leq N$ is the number of unique references. It is reasonable to assume that in most cases $N' \ll N$. As a result,

Table VI. Results for Powerstone Data Caches

Benchmark	Refs	Unique Refs	Improved Cache Index Mappings
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

the effective execution time of our heuristic is linear with respect to the size of the trace.

We note that our heuristic only looks at the address patterns and not the sequence or frequency of memory reference occurrences. Furthermore, our heuristic is by no means optimal, but it performs well and runs very efficiently as supported by our experimental results. In fact, our experiments with alternate heuristics that take into account the frequency of occurrence of memory references or sequence of memory references have resulted in marginal (and sometimes negative) performance improvements when compared to the presented indexing scheme.

It is important to note that our heuristic will converge to a random selection of index bits in the unlikely scenario where each memory location is accessed at least once. In such cases, one can limit the trace to the segment corresponding to the time critical regions of the code.

4. EXPERIMENTS

For experiments, we have used the Powerstone-embedded benchmarks [Malik et al. 2000; PowerStone 1999] as well as the integer SPEC CPU 2000 general benchmarks [SPEC'00 2000]. 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*, and a few other typical embedded applications.

4.1 Benchmark Results

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. Results for data caches are show in Table VI and Table VII for Powerstone and SPEC'00 benchmarks, respectively.

Table VII. Results for SPEC'00 Data Caches

Benchmark	Refs	Unique Refs	Improved Cache Index Mappings
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

Table VIII. Results for Powerstone Instruction Caches

Benchmark	Refs	Unique Refs	Improved Cache Index Mappings
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

Table IX. Results for SPEC'00 Instruction Caches

Benchmark	Refs	Unique Refs	Improved Cache Index Mappings
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

Results for instruction caches are shown in Table VIII and Table IX for Powerstone and SPEC'00 benchmarks, respectively. The last column of these tables shows the computed, near-optimal, index mapping. The values are ordered from left to right (i.e., the leftmost number denotes the best address bit, the second leftmost number denotes the second best address bit, etc.).

Table X. Cache Miss Results for Powerstone Data Caches

Benchmark	Conf. A (T)	Conf. A (P)	Conf. B (T)	Conf. B (P)	Conf. C (T)	Conf. C (P)
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

Table XI. Cache Miss Results for SPEC'00 Data Caches

Benchmark	Conf. A (T)	Conf. A (P)	Conf. B (T)	Conf. B (P)	Conf. C (T)	Conf. C (P)
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

We have simulated the traces under three typical cache organization schemes. Configuration *A* with 4-Kb, direct mapped, and 4-byte line; configuration *B* with 8-Kb, 2-way, and 8-byte line; and configuration *C* with 16-Kb, 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. Results for data caches are shown in Table X and Table XI for Powerstone and SPEC'00 benchmarks, respectively. Results for instruction caches are shown in Table XII and Table XIII for Powerstone and SPEC'00 benchmarks, respectively.

On the average, for the data traces, the improved cache indexing achieved 23%, 19%, and 14% reduction in cache misses, for cache configurations *A*, *B*, and *C*, respectively, as shown in Figure 3 and Figure 4. In some cases, the reduction in misses was up to 45% for data traces. On the average, for the instruction traces, the improved cache indexing achieved 14%, 10%, and 7.7% reduction in cache misses, for cache configurations *A*, *B*, and *C*, respectively, as shown in Figure 5 and Figure 6. In some cases the reduction in misses was up

Table XII. Cache Miss Results for Powerstone Instruction Caches

Benchmark	Conf. A (T)	Conf. A (P)	Conf. B (T)	Conf. B (P)	Conf. C (T)	Conf. C (P)
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

Table XIII. Cache Miss Results for SPEC'00 Instruction Caches

Benchmark	Conf. A (T)	Conf. A (P)	Conf. B (T)	Conf. B (P)	Conf. C (T)	Conf. C (P)
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

to 31% for instruction traces. For smaller caches, or larger application benchmarks, a larger reduction was observed. Moreover, the technique benefited data caches more than instruction caches.

In Table XIV and Table XV, we have summarized all the data reported thus far. Specifically, we have averaged the number of instruction and data cache among all three configurations (i.e., configuration A, B, and C). Moreover, we have used the *Wattch* simulator [Tiwari and Martonosi 2000] to obtain performance and energy figures. As predicted, our results show that both performance (i.e., execution time) and power are improved when a near-optimal cache indexing scheme is used.

4.2 Effects of Input Data on Near-Optimal Cache Indexing

In the next set of experiments, we have measured the effects of input data variation (i.e., variation on data fed to the benchmark applications) on data access pattern, control flow, and the resulting impact on the near-optimal cache indexing scheme. For these experiments we have simulated the traces under a

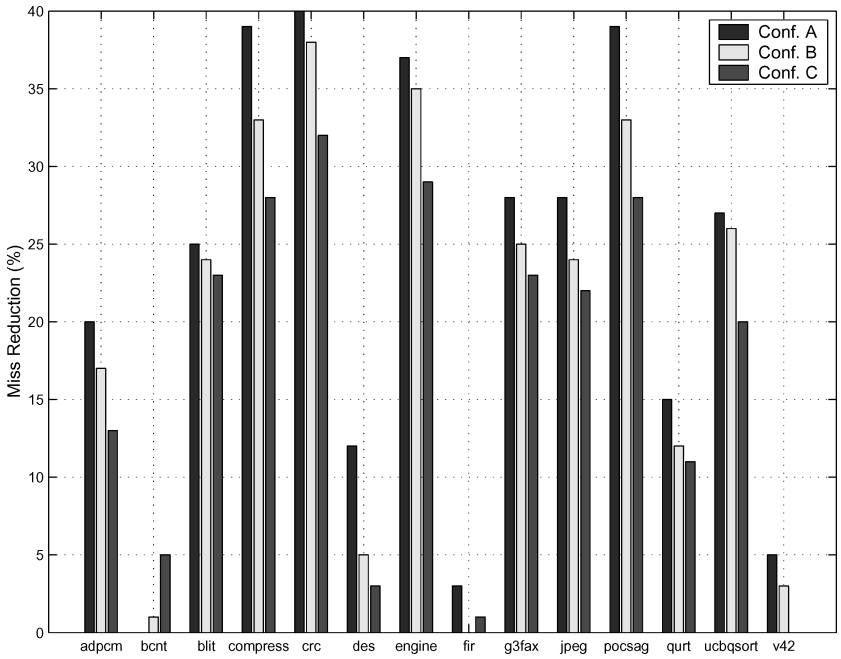

Fig. 3. Powerstone data cache miss reduction.

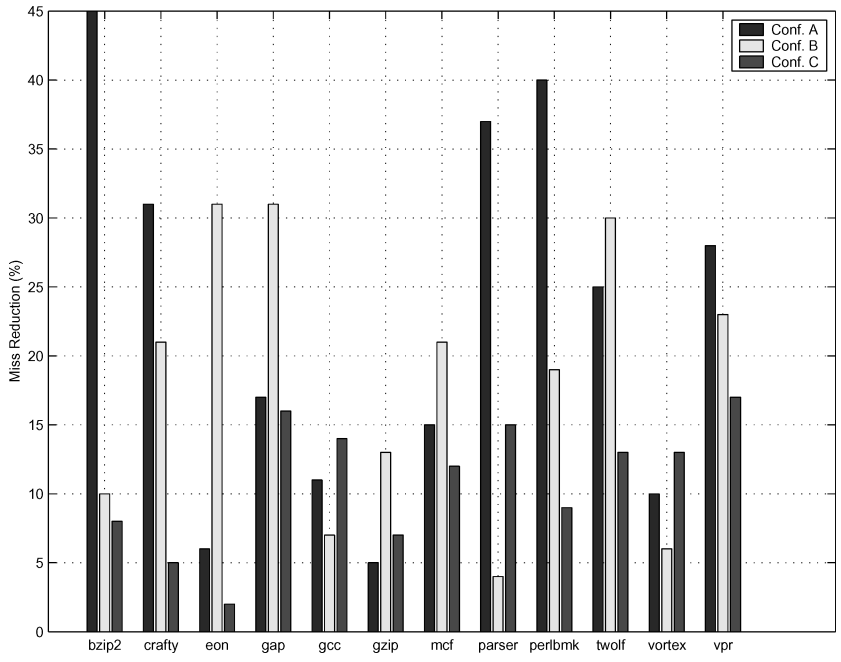

Fig. 4. SPEC'00 data cache miss reduction.

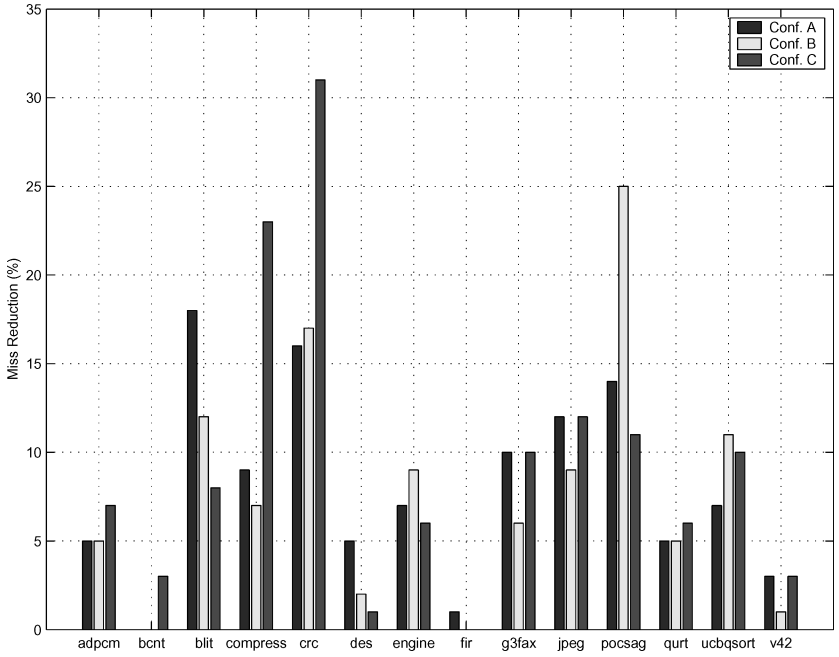

Fig. 5. Powerstone instruction cache miss reduction.

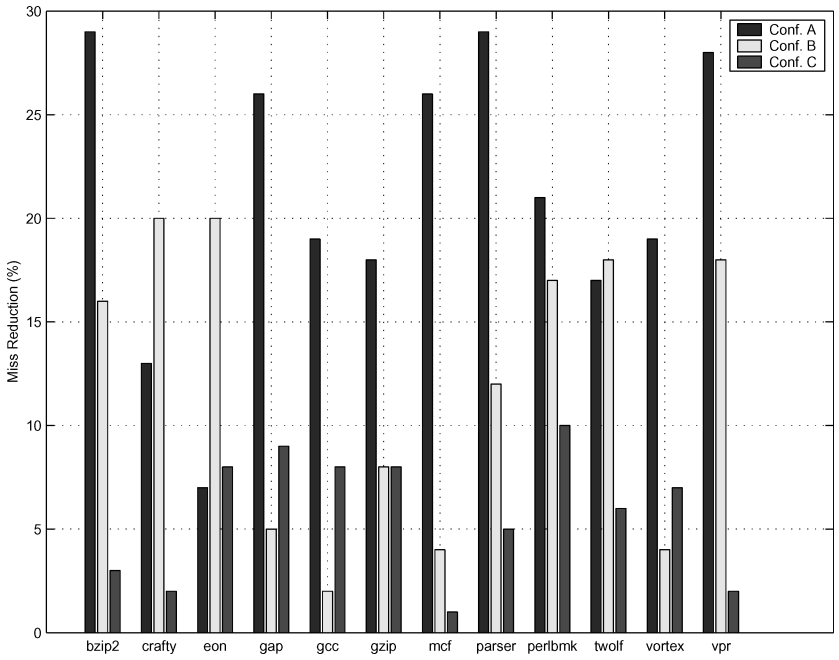

Fig. 6. SPEC'00 instruction cache miss reduction.

Table XIV. Averaged Results for the Powerstone Benchmarks

Benchmark	Misses (T)	Misses (P)	Perf. (T)	Perf. (P)	Energy (T)	Energy (P)
adpcm	11456	10524	1.23ms	1.15ms	0.655nJ	0.608nJ
bcnt	223	220	25.0us	24.0us	13.7pJ	13.1pJ
blit	4120	3143	431us	350us	0.232nJ	0.184nJ
compress	11713	8140	1.63ms	1.33ms	0.782nJ	0.603nJ
crc	574	376	181us	164us	69.1pJ	59.2pJ
des	22593	21249	2.36ms	2.24ms	1.27nJ	1.20nJ
engine	3726	2417	2.38ms	2.27ms	0.807nJ	0.742nJ
fir	917	904	147us	146us	67.0pJ	66.0pJ
g3fax	76173	56217	10.9ms	9.21ms	5.17nJ	4.17nJ
jpeg	180961	135398	34.8ms	31.ms	15.0nJ	12.7nJ
pocsag	2075	1629	377us	340us	0.165nJ	0.143nJ
qurt	195	178	21.0us	20.0us	11.0pJ	10.0pJ
ucbqsort	15285	13213	2.21ms	2.04ms	1.05nJ	0.942nJ
v42	320864	310877	37.ms	36.2ms	19.1nJ	18.6nJ

Table XV. Averaged Results for the SPEC'00 Benchmarks

Benchmark	Misses (T)	Misses (P)	Perf. (T)	Perf. (P)	Energy (T)	Energy (P)
bzip2	6963333	5356333	5.55s	5.42s	1.73uJ	1.68uJ
crafty	34176666	28403333	11.6s	11.1s	3.82uJ	3.62uJ
eon	4645000	4122000	4.37s	4.32s	1.36uJ	1.34uJ
gap	9428666	7559666	10.6s	10.5s	3.28uJ	3.21uJ
gcc	3683333	3289333	2.68s	2.64s	0.84uJ	0.83uJ
gzip	102250000	90050000	37.5s	36.5s	12.3uJ	11.8uJ
mcf	27333333	24013333	5.11s	4.83s	1.81uJ	1.69uJ
parser	52516666	41250000	29.0s	28.0s	9.22uJ	8.82uJ
perlbmk	2804000	2172333	2.22s	2.17s	0.70uJ	0.67uJ
twolf	18270000	14956666	16.8s	16.5s	5.22uJ	5.10uJ
vortex	17633333	15686666	7.04s	6.88s	2.29uJ	2.22uJ
vpr	13160000	10236666	5.14s	4.90s	1.67uJ	1.57uJ

cache configuration with 4 Kb, direct mapped, and 4-byte line. For each benchmark, we have selected four input data sets I1, I2, I3, and I4. (Input data set I1 corresponds to the default input data set used in our earlier experiments.) Results for data cache misses are shown in Table XVI and Table XVII for Powerstone and SPEC'00 benchmarks, respectively. Results for instruction cache misses are shown in Table XVIII and Table XIX for Powerstone and SPEC'00 benchmarks, respectively.

The results are summarized as follows. For each benchmark, we selected the near-optimal cache indexing scheme computed using the default input data set I1. Then, without changing the near-optimal cache indexing scheme, we ran the benchmarks using the additional input data sets I2, I3, and I4. The presented results show the percent increase or decrease in cache misses relative to the default input data set I1.

The results show that for the embedded applications (i.e., most of the Powerstone benchmarks) the effects of input data variation on the near-optimal cache indexing is minimal. For the desktop applications (i.e., most of the SPEC'00

Table XVI. Cache Miss Results, using Different Input Data, for Powerstone Data Caches

Benchmark	I1	I2	I3	I4
adpcm	5193	1.67%	−0.72%	3.75%
bcnt	164	0.44%	3.42%	4.50%
blit	4034	0.00%	0.00%	0.00%
compress	12659	4.92%	1.89%	0.77%
crc	694	0.00%	0.00%	0.00%
des	15155	−0.10%	3.85%	−0.06%
engine	7131	1.00%	0.13%	0.31%
fir	658	2.01%	0.69%	1.34%
g3fax	127828	4.11%	1.27%	4.55%
jpeg	267567	0.55%	0.01%	0.35%
pocsag	1238	4.07%	2.27%	0.70%
qurt	115	1.29%	2.23%	5.38%
ucbqsort	10862	3.48%	7.06%	4.62%
v42	157469	−0.10%	1.06%	2.91%

Table XVII. Cache Miss Results, using Different Input Data, for SPEC'00 Data Caches

Benchmark	I1	I2	I3	I4
bzip2	3.15M	6.37%	−2.83%	13.56%
crafty	15.8M	1.61%	15.23%	16.35%
eon	2.97M	10.07%	1.76%	3.34%
gap	5.45M	18.20%	8.53%	3.81%
gcc	1.69M	1.24%	20.26%	16.24%
gzip	3.64M	−0.24%	17.15%	0.23%
mcf	7.81M	−1.32%	11.17%	2.38%
parser	22.9M	7.44%	3.09%	5.85%
perlbmk	1.71M	15.30%	5.93%	16.70%
twolf	8.95M	2.24%	8.96%	19.63%
vortex	7.25M	15.15%	10.40%	3.35%
vpr	6.62M	4.83%	10.11%	19.97%

Table XVIII. Cache Miss Results, using Different Input Data, for Powerstone Instruction Caches

Benchmark	I1	I2	I3	I4
adpcm	23392	2.70%	−0.51%	5.01%
bcnt	115	0.61%	6.42%	6.04%
blit	149	0.00%	0.00%	0.00%
compress	4435	6.88%	3.87%	1.39%
crc	176	0.00%	0.00%	0.00%
des	23113	0.11%	7.26%	0.08%
engine	244	−0.20%	0.03%	0.01%
fir	1566	2.82%	1.35%	2.14%
g3fax	220	5.95%	3.04%	6.16%
jpeg	26097	1.20%	3.84%	7.24%
pocsag	3730	5.86%	4.95%	1.22%
qurt	179	1.91%	4.65%	7.36%
ucbqsort	30629	1.00%	0.41%	6.27%
v42	555022	−0.06%	1.99%	3.98%

Table XIX. Cache Miss Results, using Different Input Data, for SPEC'00 Instruction Caches

Benchmark	I1	I2	I3	I4
bzip2	8.58M	5.15%	−0.71%	11.90%
crafty	43.2M	1.28%	14.24%	14.47%
eon	6.18M	7.98%	1.67%	3.24%
gap	14.4M	14.43%	8.75%	4.81%
gcc	3.12M	1.03%	17.78%	14.09%
gzip	124M	−0.13%	16.11%	0.81%
mcf	20.9M	−0.95%	11.25%	2.94%
parser	65.6M	5.90%	3.02%	6.58%
perlbmk	3.78M	12.19%	7.10%	14.99%
twolf	27.7M	1.89%	8.55%	17.64%
vortex	17.9M	12.06%	11.34%	4.07%
vpr	15.1M	3.86%	10.56%	18.25%

Table XX. Cache Miss Results for Powerstone Unified Cache

Benchmark	Traditional (T)	Improved (P)	Difference (%)
adpcm	24678	24678	0.0%
bcnt	278	258	7.2%
blit	4095	2967	27.5%
compress	16066	10949	31.8%
crc	762	540	29.1%
des	36168	34325	5.1%
engine	6035	4608	23.6%
fir	2125	2020	4.9%
g3fax	115029	91613	20.4%
jpeg	263525	202692	23.1%
pocsag	4276	3780	11.6%
qurt	291	267	8.2%
ucbqsort	40361	36291	10.1%
v42	612380	612380	0.0%

benchmarks), the effects of input data variation on the near-optimal cache indexing is more pronounced. In a particular example, `gcc` behaved very differently in both control and data cache performance with different input data set, which consisted of compiling different C/C++ programs. On the other hand, `jpeg` behaved with little variation in both control and data cache performance regardless of the input data set, which consisted of decoding different jpeg images.

4.3 Near-Optimal Cache Indexing Applied to Unified Caches

In the next set of experiments, we have applied our near-optimal cache indexing scheme to unified caches. For these experiments, we have simulated the traces under a cache configuration with 8 Kb, direct mapped, and 4-byte line. This particular cache configuration corresponds to a cache that is as large as the combined instruction and data caches of our earlier experiments (configuration *A*) on split cache architectures. Results are summarized in Table XX and Table XXI for Powerstone and SPEC'00 benchmarks respectively. The tables provide the number of cache misses under a traditional (T) indexing scheme, the number of cache misses under an improved (P) indexing scheme, and the

Table XXI. Cache Miss Results for SPEC'00 Unified Cache

Benchmark	Traditional (T)	Improved (P)	Difference (%)
bzip2	9.41M	7.38M	21.6%
crafty	58.4M	44.3M	24.1%
eon	7.61M	7.61M	0.0%
gap	17.9M	13.9M	22.7%
gcc	4.55M	3.97M	12.6%
gzip	111M	98.0M	11.6%
mcf	26.6M	21.0M	21.2%
parser	84.8M	58.5M	31.1%
perlbmk	4.73M	3.96M	16.1%
twolf	33.6M	28.7M	14.6%
vortex	22.4M	20.0M	10.5%
vpr	21.1M	15.4M	27.0%

percent difference (using the traditional (T) indexing scheme as the reference point).

Our results show an average improvement of 14.5% for the Powerstone benchmarks and an average improvement of 15.2% for the SPEC'00 benchmarks.

4.4 Final Remarks

We conclude this section by making some final remarks about our experiments and the obtained results.

—Not in all cases did our heuristic algorithm obtain an indexing scheme that was better (or significantly better) than the traditional indexing scheme. For example, Figure 3 to Figure 6, as well as Table XX and Table XXI show a few instances where the miss reduction was reported to be zero. In such cases, we were able to apply an alternate search heuristic (based on a simulated annealing greedy algorithm) that was able to obtain some indexing scheme better than the traditional. However, the run time of the search heuristic was impractical in practice (e.g., taking many days to run on smaller benchmarks and much long on larger benchmarks). Therefore, we conclude that the near-optimal cache indexing technique is always feasible, however our heuristic could be improved further.

—Based on our results, the average improvement was better for split data/instruction cache architecture than for unified cache architecture. However, in both architectures the improvement was significantly better than the traditional indexing scheme.

—The particular input data used when running a benchmark application to obtain a trace file can have an impact on the final indexing scheme that is selected. However, for most embedded benchmarks (i.e., those from Powerstone), a cache indexing scheme obtained from a particular input data set performed well when the application was simulated using alternate input data sets, as shown in Table XVI and Tale XVIII. However, for desktop applications (i.e., those from SPEC'00), the results showed a larger deviation from the near-optimal solution, as shown in Table XVII and Table XIX.

—In terms of absolute performance and energy, as reported in Table XIV and Table XV, we note that significant improvements are achievable by applying our technique.

5. 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 near-optimal set of bits used for indexing into the cache. While an optimal selection of index bits is shown to be NP-complete, we have provided an efficient heuristic algorithm for computing a near-optimal indexing scheme. This heuristic algorithm computes in polynomial time and produces good results, as demonstrated by experiments on a large number of Powerstone and SPEC'00 benchmarks. Specifically, for data traces, our technique achieves up to 45% reduction in cache misses. Likewise, for instruction traces, our technique achieves up to 31% reduction in cache misses. When a unified data/instruction cache architecture is considered, our results show an average improvement of 14.5% for the Powerstone benchmarks and an average improvement of 15.2% for the SPEC'00 benchmarks.

In the future, we plan to investigate cache indexing schemes that may incur some constrained cost and analyze the tradeoffs in terms of improved hit rate versus increased cache access time. For example, we may consider cache indexing schemes that use one, two, or n-level logic. We also plan to investigate a dynamic approach to cache indexing by using a reprogrammable crossbar along the processor/cache and cache/memory buses, enabling on-the-fly swapping of cache address wires by a task or an operating system.

ACKNOWLEDGMENTS

The author would like to thank Gopi Meenakshisundaram and David Eppstein for their contributions to this work.

REFERENCES

ABELLA, J., GONZALEZ, A., LIOSA, J., AND VERA, X. 2002. Near-optimal loop tiling by means of cache miss equations and genetic algorithms. In *Proceedings of the International Conference on Parallel Processing Workshops* (Washington, DC). IEEE Computer Society Press, Los Alamitos, CA, 568–580.

BALASUBRAMONIAN, R., ALBONESI, D., BUYUKTOSUNOGLU, A., AND DWARKADAS, S. 2000. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *Proceedings of the International Symposium on Microarchitecture*. ACM, New York, 245–257.

CORMAN, T., LEISERSON, C., RIVEST, R., AND STEIN, C. 2001. *Introduction to Algorithms*, 2 ed. The MIT Press and McGraw-Hill, Cambridge, MA.

DING, C. AND KENNEDY, K. 1999. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the Conference on Programming Language Design and Implementation*. ACM, New York, 229–241.

GRUN, P., DUTT, N., AND NICOLAU, A. 2000. Memory aware compilation through accurate timing extraction. In *Proceedings of the Design Automation Conference*. ACM, New York, 316–321.

GRUN, P., DUTT, N., AND NICOLAU, A. 2001. Aggressive memory-aware compilation. In *Intelligent Memory Systems*. Springer-Verlag, London, UK, 147–151.

GURARI, E. 1989. *An introduction to the theory of computation*. McGraw-Hill, New York.

HUANG, Q., XUE, J., AND VERA, X. 2003. Code tiling for improving the cache performance of pde solvers. In *Proceedings of the International Conference on Parallel Processing*. ACM, New York, 615–626.

ITRS. 2005. Technology roadmap for semiconductors. http://www.itrs.com.

KOZYRAKIS, C. AND PATTERSON, D. 1998. A new direction for computer architecture research. *IEEE Comput. 31*, 11, 24–32.

MALIK, A., MOYER, B., AND CERMAK, D. 2000. A low power unified cache architecture providing power and performance flexibility. In *Proceedings of the Symposium on Low Power Electronics and Design*. ACM, New York, 241–243.

PANDA, P., NAKAMURA, H., DUTT, N., AND NICOLAU, A. 1997. Improving cache performance through riling and data alignment. In *Proceedings of the Workshop on Parallel Algorithms for Irregularly Structured Problems*. ACM, New York, 167–185.

PATTERSON, D. AND HENNESSY, J. 1997. *Computer Organization and Design: The Hardware/Software Interface*, 2 ed. Morgan-Kaufmann, San Francisco, CA.

PETROV, P. AND ORAILOGLU, A. 2001. Towards effective embedded processors in codesigns: Customizable partitioned caches. In *Proceedings of the International Conference on Hardware/Software Codesign*. ACM, New York, 79–84.

POWERSTONE. 1999. The powerstone benchmarks. www.motorola.com.

RIVERA, G. AND TSENG, C.-W. 1997. Compiler optimizations for eliminating cache conflict misses. Tech. Rep. CS-TR-3819, University of Maryland.

SPEC'00. SPEC CPU 2000. http://www.spec.org.

SU, C. L. AND DESPAIN, A. 1995. Cache design trade-offs for power and performance optimization: A case study. In *Proceedings of the International Symposium on Low Power Electronics and Design*. ACM, New York, 63–68.

SUZUKI, K., ARAI, T., KOUHEI, N., AND KURODA, I. 1998. V830R/Av: Embedded multimedia superscalar RISC processor. *IEEE Micro 18*, 2, 36–47.

TIWARI, B. AND MARTONOSI, M. 2000. Wattch: A framework for architecture-level power analysis and optimization. In *Proceedings of the Symposium on Computer Architecture*. ACM, New York, 83–94.

VAHID, F. AND GIVARGIS, T. 1999. The case for a configure-and-execute paradigm. In *Proceedings of the International Conference on Hardware/Software Codesign*. ACM, New York, 59–63.

WONG, S., VASSILIADIS, S., AND COTOFANA, S. 2004. Future directions of (programmable and reconfigurable) embedded processors. In *Domain-Specific Processors: Systems, Architecture, Modeling, and Simulation*. Marcel Dekker, Inc., London, UK, 235–257.

Received February 2004; revised February 2005; accepted May 2005