Improved Indexing for Cache Miss Reduction in Embedded Systems

Tony Givargis
Department of Computer Science, Center for Embedded Computer Systems
University of California, Irvine, Irvine, CA 92697

givargis@ics.uci.edu

Abstract
The increasing use of microprocessor cores in embedded systems as well as mobile and portable devices creates an opportunity for customizing the cache subsystem for improved performance. In traditional cache design, the index portion of the memory address bus consists of the K least significant bits, where $K = \log_2(D)$ and $D$ is the depth of the cache. However, in devices where the application set is known and characterized (e.g., systems that execute a fixed application set) there is an opportunity to improve cache performance by choosing an optimal set of bits used as index into the cache. This technique does not add any overhead in terms of area or delay. We give an efficient heuristic algorithm for selecting K index bits for improved cache performance. We show the feasibility of our algorithm by applying it to a large number of embedded system applications as well as the integer SPEC CPU 2000 benchmarks.

Categories and Subject Descriptors
B.3.2 [Design Styles]

General Terms: Algorithms, Performance, Experimentation

Keywords
Cache Optimization, Design Space Exploration, Index Hashing

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

The use of data and instruction caches has been a major factor in improving processing speed of today’s microprocessors. Generally, a well-tuned cache hierarchy and organization can reduce the time overhead of fetching instruction and data from main memory, which in most cases resides off-chip, requiring power costly communication over the off-chip system bus.

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

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

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

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

In related work, researchers have studied data layout and memory/cache aware compiler techniques for improved cache
we can potentially use any combination of size and has line size equal to a heuristic that is efficient in running time and produces good results when applied in practice. In the case of the code segment, data movement is typically performed at the basic block granularity. In other related work, researchers have studied indexing and hashing in the context of IP routing [15][16].

Specifically, let us assume that a processor has an all address bits of a processor for indexing into the cache.

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

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

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

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

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

2.2 NP Completeness

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

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 $X_0, X_1, \ldots X_{m-1}$ 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.

<table>
<thead>
<tr>
<th>Index Bits</th>
<th>Set Intersections</th>
<th>Cardinality/#Misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_0$</td>
<td>$X_0 = {(4,2),(5,1)}$</td>
<td>$3 + 2 = 5$</td>
</tr>
<tr>
<td>$X_1$</td>
<td>$X_1 = {(4,3)}$</td>
<td>$3 + 1 = 4$</td>
</tr>
<tr>
<td>$X_2$</td>
<td>$X_2 = {(4,3),(4,2),(5,1),(5,3)}$</td>
<td>$3 + 2 = 5$</td>
</tr>
</tbody>
</table>

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

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

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

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

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

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

2.3 Heuristic Algorithm

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

\[
Q_i = \frac{\min(Z_i, O_i)}{\max(Z_i, O_i)} \text{ where } Z_i : \text{the number of references having 0 at bit } A_i, O_i : \text{the number of references having } 1 \text{ at bit } A_i
\]

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

<table>
<thead>
<tr>
<th>$A_0$</th>
<th>$A_1$</th>
<th>$A_2$</th>
<th>$A_3$</th>
<th>$A_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3: A sample striped application trace.

Here, $Q_0, Q_1, \ldots Q_8$ are computed as shown in Table 4.

<table>
<thead>
<tr>
<th>$Q_0$</th>
<th>$Q_1$</th>
<th>$Q_2$</th>
<th>$Q_3$</th>
<th>$Q_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3/7</td>
<td>1</td>
<td>1</td>
<td>2/3</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4: Quality measures.
As an example, looking at Table 3 and for $Q_0$, $A_0^{th}$ column, there are seven zeros and three one bits, thus we compute as follows.

$$Z_4 = 7, O_4 = 3$$

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

For each pair of bits in our address space, we compute a corresponding correlation measure. This correlation measure is a real number in the range of zero to one. A correlation measure of one indicates that a pair of address bits split the unique references in completely different ways. 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. 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 zero indicates that a pair of address bits split the unique references in exactly the same way.

Figure 2: Correlation measure: (a) $A_1$ used as index, (b) $A_2$ used as index, (c) $A_0$ and $A_1$ used as indices.

As an example, looking at Table 3 and for $Q_0$, $A_0^{th}$ column, there are seven zeros and three one bits, thus we compute as follows.

$$Z_4 = 7, O_4 = 3$$

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

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

In terms of running time complexity, our algorithm takes $O(N \cdot \log(N))$ to execute. Note that reading the trace takes $O(N)$, as the length of the trace is $N$. Reducing the trace down to only the unique references involves what amounts to sorting the trace and thus takes $O(N \cdot \log(N))$. 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 $A_0, A_1, A_2, A_3, A_4, A_5$. In most cases $M$ is a small number, like 32, and thus is assumed to be a constant.

### 3. Experiments

For experiments, we have used the Powerstone embedded benchmarks [4] as well as the integer SPEC CPU 2000 general benchmarks [18]. The PowerStone benchmarks include a JPEG decoder called jpeg, a modem protocol processor called v42, a Unix compression utility called compress, a CRC checksum algorithm called crc, an encryption algorithm called des, an engine controlled called engine, an FIR filter called fir, a group three fax decoder called g3fax, a sorting algorithm called ucbsort, a rendering algorithm called blit, a POCSSAG communication protocol for paging called pocssag, etc.

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

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

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

On the average, for the data/instruction traces, the improved cache indexing achieved 23%/14%, 19%/10%, and 14%/7.7% reduction in cache misses, for cache configurations A, B, and C respectively, as shown in Figure 3. In some cases the reduction in misses was up to 45% for data traces and 31% for instruction traces. For smaller caches, or larger application benchmarks, a larger reduction was observed. The technique benefited data caches more than address caches.
4. Conclusion
We have proposed a zero cost technique for improving cache performance in embedded systems as well as mobile and portable general-purpose devices that execute a known application set. Our technique involves selecting an optimal set of bits used for indexing into the cache. We have provided an efficient algorithm for computing an optimal indexing scheme. Our heuristic algorithm produces good results, as demonstrated by experiments on a large number of benchmarks.

5. Acknowledgement
This work was supported by the National Science Foundation.

6. References