Data Prefetching using a modified Markov Predictor

Ben Weber
Department of Computer Science
California Polytechnic State University
San Luis Obispo, CA 93407
bgweber@calpoly.edu

Abstract

As processors are clocked faster and faster, the gap between processor and memory speeds grows exponentially. This difference in performance causes significant memory latencies for programs with a large number of pointer references. C++ and Java programs suffer performance hits, because of this memory delay. Several methods have been developed to overcome this gap, including hardware and software techniques. These methods attempt to prefetch data into the processor before the processor needs the data, avoiding cache misses. A hardware based approach that improves caching performance is the Markov predictor.

This paper presents two modified versions of the Markov predictor. The first design includes the addition of a pointer detection table offering an insignificant speedup for the SPEC2000 benchmarks. The second modified version of the Markov predictor uses a linked data structure, improving the L2 cache performance of 10 SPEC2000 benchmarks by an average of 6%.

1 Introduction

High level languages such as Java and C++ allow programmers to develop for a target platform without specific knowledge of the hardware, but suffer large performance penalties due to memory latencies. High level languages are affected by memory accesses more than lower level languages, because of the usage of pointers. Pointers create a problem for current computer architectures, because the temporal and spatial algorithms implemented by caches can not predict the memory location contained in a pointer. Due to a high cache miss rate, programs that use a large number of pointers suffer performance hits. This problem is especially evident in high level programs using linked data structures. One of the methods used to speed up program execution is intelligent pointer prefetching.

This paper investigates the results of L2 cache hit rates using a modified Markov predictor [1]. The predictor is an on-chip component that attempts to prefetch data into the L2 cache before misses occur. The goal of the predictor is to avoid the latencies caused by accessing the memory subsystem. Figure 1 displays the hit ratios of several SPEC2000 benchmarks. Using the 64k entry predictor described in this paper, the overall performance of the L2 cache was improved by six percent.

This paper first surveys various techniques for prefetching. The first modification of the Markov predictor is described in Section 3. The second design is discussed in Section 4. Next, the ideal and a real world implementation of the predictor are tested. The last sections discuss future improvements and conclusions.
2 Related Work

Currently there are several methods used to hide the latencies caused by memory. Hardware, software, and hybrid techniques have been developed to prefetch data. All of these methods have tradeoffs between complexity, accuracy, and efficiency. The complexity of a method involves the amount of additional hardware required or additional compiler complexity. The accuracy of a prefetching technique is its ability to prevent cache misses. Finally, the efficiency of these methods is the ability to only prefetch data that is actually used by the processor.

2.1 Software Based Techniques

In order to increase the performance of Java and C++ programs, two software prefetching techniques have been developed. The first method is the addition of a prefetch instruction, which tells the processor to prefetch data into cache [4]. This is a software based technique, but additional hardware is required to implement the prefetch instruction. This technique allows linked data structures to be prefetched into memory before the data structure is referenced. The addition of a prefetch instruction prevents a portion of the cache misses from occurring. However, it is difficult to effectively add prefetch instructions to a program. Prefetch instructions must bring data into the cache before the processor references the pointer; otherwise the prefetch offers no performance benefit. The prefetch also can not be requested too early, because the cache would get polluted by prefetched data. Therefore, this method can not be applied as a general approach to improving processor performance.
A second approach to improving cache performance is compiler driven prefetching [5]. It is possible to group dynamically allocated objects together to take advantage of the spatial characteristics of cache. These techniques are usually applied to the compiler, but have also been adapted by a Java virtual machine [8]. Applying prefetching techniques adds considerable complexity to the compiler or the virtual machine, but improves the accuracy and efficiency of prefetching. However, neither of these software techniques can be directly applied to unmodified SPEC2000 benchmarks. In order to get a more general increase in processor performance, it is necessary to use a hardware based prefetching mechanism.

2.2 Hardware Based Techniques

This paper surveys two hardware methods of prefetching: stride prefetchers [7] and Markov predictors [1]. Stride prefetchers run ahead of the processor and prefetch data into cache. These prefetchers are very accurate and efficient, because they exploit the spatial nature of programs. However, these prefetchers fail to prefetch pointer references, because of the indirection caused by pointers. A different type of prefetcher must be used for detecting pointers.

Another hardware based approach to prefetching is the Markov predictor. The Markov predictor is an attempt to improve caching by detecting reference patterns. The predictor watches the cache miss stream and forms predictions based on the misses. By watching the miss stream, the Markov predictor can prefetch data that other techniques would miss. The predictor improves cache performance, but has several drawbacks. In order for the predictor to issue a prefetch, several cache misses must occur first. Also, it is difficult to limit the memory bandwidth used by the predictor. Finally, the predictor is less accurate than other prefetching techniques.

3 Hardware Predictor Design

The goal of the Markov predictor is to improve caching by prefetching data that is not cached by traditional methods. In order to be useful, the predictor must be able to outperform simply adding more L2 cache to the processor. An example design for an effective Markov predictor is described by Joseph and Grunwald [1].

3.1 The Basic Markov Predictor

The predictor evaluated by Joseph and Grunwald is an on-chip prefetcher which consists of a prediction table and a prefetch queue. The design for the prefetch unit is displayed in Figure 2. The predictor operates by listening to the cache miss stream. When an L2 cache miss occurs, the predictor adds an entry into the prediction table. The next couple of cache misses that occur are added as the prediction for the previous address. The exact number of addresses that are added to the table can be varied. These addresses form the prediction and are prefetched into memory when the corresponding miss address is referenced by the processor. The prefetcher assumes that the prediction addresses will be referenced shortly after the miss address is referenced. This process allows the predictor to improve caching by discovering reference patterns.
The Markov predictor is able to improve cache performance, but there are several problems with the design. The most inherit problem is that the predictor has a large learning phase; it must wait for two cache misses before an entry is added to the table. Data is not prefetched until one of the missed addresses is referenced again. The predictor would be more effective if pointer references were detected before they occurred as cache misses. Another problem with the predictor is deciding when to add entries to the table and selecting the appropriate number of prediction addresses. If there are too many predictions, then the predictor will be less accurate and cause more cache pollution. In order to limit the bandwidth used by the predictor, it is necessary to devise an efficient policy for adding predictions.

### 3.2 The Modified Design

The first modified predictor consists of the original design plus an additional table used to detect pointer references. Also, there are more predictions associated with each miss address. The goal of the modified design is to limit bandwidth usage by only adding entries to the table when pointer references are detected. This adds more complexity to the predictor, but should improve prefetch accuracy and limit cache pollution. The design of the modified predictor is shown in Figure 3. An entry is added to the pointer table when a cache miss occurs. Once the value for the address is known, it is added to the value field in the table. When a cache miss occurs, the predictor checks if the address is the same as any of the value fields. This situation occurs when a pointer is referenced, and the original address is added to the prediction table. The next cache misses that occur become part of this prediction entry. The rest of the predictor works the same way as the original design.
### 3.3 Performance of the Modified Design

The first design was modeling using output from SMTSIM simulating the SPEC2000 benchmarks shown in Figure 1. The first experiment was to determine the amount of pointer references detected using this technique. All pointer references can be detected by assuming an infinitely large pointer detection table. The results for the experiment are displayed in Figure 4. It is evident that almost none of the benchmarks will benefit from this technique, because an insignificant amount of the instructions are detected as pointers. The only benchmark that has a possible benefit is gcc. These benchmarks most likely contain more pointer references, but this hardware technique only detects a fraction of them. In order to get any performance benefit from this design, it is necessary to have a large number of predictions associated with each miss address.

![Figure 3 – First modification of the Markov predictor](image)

#### Figure 3 – First modification of the Markov predictor

![Figure 4 - Pointer references for SPEC2000 benchmarks](image)

#### Figure 4 - Pointer references for SPEC2000 benchmarks
A second experiment was conducted using a pointer table with 64 entries and three variations of the prediction table size. The results are shown in Figure 5. On average, the predictor was only able to successfully prevent under 4% of the cache misses. Overall, the prefetcher would take about 256 KB of space. Considering the size of the prefetcher, it would be more effective to add more cache to the processor than to use this design.

![Predictor Performance](image)

**Figure 5 - Predictor performance of the first modification**

There are several problems with this design. In order to offer any perceivable benefit to the processor, it is necessary to make the predictor table significantly large. The table is able to detect pointer references, but it is not able to successfully prefetch linked data structures. Due to the large prediction size, the predictor acts more like a trace cache than a pointer prefetcher. Also, the table still has a large learning phase. In order to improve the performance of the predictor, a more dynamic design is explored.

### 4 A Revised Design

One of the main problems with the previous design was its static nature to predictions. One prediction did not relate to any other predictions, causing the number of instructions that could be prefetched to be limited. This section explores a revised design of the prefetcher that allows an almost limitless number of predictions, because the table entries are linked together.
4.1 Predictor Design
The second design does not have the pointer table described in the previous section and the prediction sizes are limited to one address. However, there is an additional field in the table which points to the next table index to prefetch. The revised design is shown in Figure 6. When a cache miss occurs, the predictor checks if there is an entry for the missed address. If there is an entry, then the corresponding address is prefetched into L2 cache and the predictor stores the next table index. The next cycle, the predictor checks if the miss address of the stored index is equal to the address previously prefetched. If the addresses are the same, the predictor repeats this process. Using this technique, it is possible to prefetch several addresses when only one cache miss occurs. However, it is now more difficult to add entries into the table.

<table>
<thead>
<tr>
<th>Miss Address 1</th>
<th>Prediction</th>
<th>Next Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>Prediction</td>
<td>Next Index</td>
</tr>
<tr>
<td>...</td>
<td>Prediction</td>
<td>Next Index</td>
</tr>
<tr>
<td>Miss Address N</td>
<td>Prediction</td>
<td>Next Index</td>
</tr>
</tbody>
</table>

Figure 6 - Second modification of the Markov predictor

When a cache miss occurs and there is no corresponding entry for the address, a new entry is added to the table. Adding entries to the table requires additional hardware, because the predictor must remember the previously added entry. When the new entry is added, the predictor sets the next index for the previous entry. Also, when a new entry is added, the next index is initially set to a value of 0. The revised design is more complex than the original Markov predictor, but should lessen the memory latencies caused by linked data structures.

4.2 Hardware Implementation
Implementing the revised design requires adding complexity to the predictor. The main additional hardware is used to iterate through entries in the predictor. This requires a place to store the next index to retrieve and the previous address that was prefetched. It is not actually necessary to add the next index field to the table, because the next index can be found by indexing the prefetched address. However, if a complex indexing scheme is used, it is easier to store the index than to recalculate it. The performance of this design is evaluated in the following section.
5 Predictor Evaluation

The performance of the predictor was tested using SMTSIM and the SPEC2000 benchmarks shown in Figure 1. These benchmarks were chosen, because they represent a wide range of L2 cache performances. The L2 cache status from SMTSIM was output to a text file which was parsed by a Java program. The Java program simulated the number of cache misses that would have been prevented if the predictor had been used by SMTSIM. The Java program assumed that the predictions would not cause any L2 cache pollution. Therefore the results of these experiments do not exactly model actual predictor performance.

5.1 An Ideal Predictor

In order to determine the maximum benefit for the predictor, an ideal predictor was evaluated. The ideal predictor was modeled by assuming an infinitely large prediction table. The performance of an ideal predictor is shown in Figure 7. Overall, the predictor is able to prefetch over 36% of the cache misses. The results for the art benchmark are the most impressive. The normal L2 performance for art was a miss rate of about 95%. The ideal predictor would be able to prevent more than 90% of these misses. The predictor also offers significant benefits for the ammp benchmark. However, the rest of the benchmarks that gain significant improvement already have high L2 hit rates. Unfortunately, the benchmarks equake and swim have no improvement. An actual implementation of this predictor is explored next.

![Ideal Predictor Performance](image)

Figure 7 - Performance of the ideal predictor
5.2 A Real World Implementation

The performance of the predictor was modeled using four different prediction table sizes: 1k, 4k, 16k, and 64k entries. The addresses were indexed using the least significant bits, excluding the two least significant. The results for this experiment are shown in Figure 8. Overall the predictor prefetched about 22% of the cache misses with 64k entries in the predictor table, a difference of 14% from the ideal case. The main differences between the ideal predictor are the ammp and art benchmarks. The overall performance of the L2 cache using the predictor is shown in Figure 9. The predictor improved the performance of the cache by over 6%. Using this design, only the ammp and art benchmarks showed significant improvements.

![Revised Predictor Performance](image)

5.3 Predictor Coverage and Efficiency

The revised predictor has good efficiency, but has poor coverage. The predictor has high efficiency, because a prefetch only occurs when a pattern detected. However, the predictor had poor coverage, because it must to wait for misses before it can issue a prefetch. The main problem with the predictor is one of the flaws of the original Markov predictor; it must wait for cache misses to occur. This paper did not evaluate the affects of cache pollution, but it appears that the predictor would have a minimal impact. First of all, data is prefetched into the L2 cache, as opposed to the L1 cache. Also, the processor used most of the data prefetched due to the high efficiency rate of the predictor.
Overall L2 Cache Performance

The predictor offers only a slight improvement in performance even though it requires a large amount of space to implement. It would be more efficient to add more L2 cache rather than implement the predictor. However, simply adding more cache to a processor can only improve processor performance by a limited amount. As processors are clocked faster and faster, it is necessary to develop alternative caching techniques. The best approach to prefetching might actually be a hybrid hardware and software approach, such as Dynamic Speculative Precomputation [6].

6 Conclusion

This paper has introduced two modified versions of the Markov predictor, a hardware based data prefetching mechanism. The first design includes the addition of a pointer detection table offering an insignificant speedup for the SPEC2000 benchmarks. The second modified version of the Markov predictor uses an additional field which links prefetch predictions together. Using this prefetching technique, it is possible to improve the L2 cache performance of 10 SPEC2000 benchmarks by an average of 6%. However, the current predictor design is unsuitable for an actual processor, because of the amount of space required for the predictor.

Future improvements to the predictor include adding association to the prediction table. Indexing caused several prediction conflicts, and alternative indexing methods or adding more association would possibly improve performance. Other improvements including moving the predictor off-chip and letting the predictor run asynchronous to the processor. The goal of the predictor was to improve the caching performance of linked data structures; therefore a linked prediction unit was evaluated. The predictor improves L2 cache performance, but it is more effective to just add more cache to a processor.
7 References


