Cache Prefetching

Stefan G. Berg

Department of Computer Science & Engineering
University of Washington
Seattle, WA 98195-2350

Technical Report UW-CSE 02-02-04

ABSTRACT

Cache prefetching is a memory latency hiding technique that attempts to bring data to the caches before the occurrence of a miss. A central aspect of all cache prefetching techniques is their ability to detect and predict particular memory reference patterns. In this paper we will introduce and compare how this is done for each of the specific memory reference patterns that have been identified. Because most applications contain many different memory reference patterns, we will also discuss how prefetching techniques can be combined into a mechanism to deal with a larger number of memory reference patterns. Finally, we will discuss how applicable the currently used prefetching techniques are for a multimedia processing system.

Keywords: prefetching, cache, memory, multimedia

1. INTRODUCTION

The gap between memory and processor speed is a widely known phenomena to computer architects. It exists not because we are unable to build memories fast enough to keep up with the processor, but because we cannot make such memories big enough to hold the working-set of typical applications. This is due to the inverse relationship between the size and access time of computer memories [1].

The memory hierarchy of register file, caches, main memory, and disks exploits the facts that smaller memories are faster and most applications have spatial and temporal locality [1]. Temporal locality is a critical component, because if the memory access pattern were randomly distributed, then most references would go to the larger, slower memories and little would be gained from the fast memories. Temporal locality allows us to place data in such a way that in many applications the vast majority of the references go to the faster, yet smaller, memories. Data placement is a key component in achieving this goal.

The intelligent movement of data to faster memories (closer to functional units) is more important than data movement to slower memories. When a functional unit operates on some data that is not available in a fast memory, then this operation must be delayed until the data has been transfered from a slow memory. If this happens frequently, functional units may become underutilized due to the unavailability of data. In comparison. data that must be moved to slower memories can usually be stored in a write buffer and scheduled for write-back at a later time without the need to complete to operation immediately.

Without cache prefetching, data is transfered from the lower level of the memory hierarchy to a register (RISC) or functional unit (CISC) by the use of an explicit memory instruction. The computation of the memory address and the scheduling of the memory instruction is limited by control-flow and data-flow dependencies. Because of this, data may not be available to the functional units when it is needed.

Cache prefetching is a mechanism to speculatively move data to higher levels in the cache hierarchy in anticipation of instructions that require this data. Prefetching can be performed on some architectures by issuing load instructions to a non-existent register, for example register zero in architectures where this register is hardwired to zero. In this case, prefetching is controlled by the compiler and is called software prefetching. Hardware prefetching is the alternative case, where a hardware controller generates prefetching requests from information it can obtain at run-time (e.g., memory reference and cache miss addresses). Generally, software prefetchers use compile-time and profiling information while hardware prefetchers use run-time information. Both have their advantages and both can be very effective.

Cache prefetching reduces the cache miss rate because it eliminates the demand fetching of cache lines in the cache hierarchy [1]. It is also called a latency hiding technique because it attempts to hide the long-latency transfers from lower levels to higher levels of the memory hierarchy behind periods of time during which the processor executes instructions.

The remainder of the paper is organized as follows. In Section 2 we will discuss the metrics used for evaluating prefetching techniques. Section 3 compares the different prefetching techniques, forming the main part of this paper. Before concluding, we will discuss in Section 4 how prefetching applies to the domain of multimedia processing.

2. METRICS

Prefetching aims to reduce the average memory latency and thus decrease the execution time. Apart from execution time, a number of other metrics have been used in discussing the effectiveness of the various prefetching techniques. In this section, we will introduce and discuss these metrics to define a consistent vocabulary for the remainder of the paper. We will assume a uniprocessor environment for this paper, although cache prefetching can certainly be applied to multiprocessors, too.

2.1. Coverage and Accuracy

Joseph and Grunwald [2] define prefetching coverage as “the fraction of miss references that are removed by prefetching” and accuracy as “the fraction of prefetches that are useful.” For our discussion, we will assume a useful prefetch to contain data that the processor references at least once before it is replaced. The definition for coverage is intuitive, but as we will show, it is not the only definition used. First, we will need to define some terminology though.

One place to store prefetched data is the L1 cache [3–8]. However, when the prefetch accuracy (and timeliness) is not perfect, then storing prefetched data in the L1 cache risks replacement of useful data with useless data. An alternative approach stores prefetched data in a prefetch buffer [9–12]. On a hit in the prefetch buffer, data is transfered from prefetch buffer to L1 cache. If this never happens, a replacement scheme will remove the useless data from the prefetch buffer eventually. With good prefetch timing (see Section 2.2), prefetched data is transfered soon into the L1 cache and thus a small prefetch buffer is typically sufficient.

Let $M_{np} = m_1, m_2, ..., m_k$ be the sequence of miss addresses exhibited by a particular application running on an architecture without prefetch mechanism. In an architecture with prefetch buffers, such as the Markov prefetcher by Joseph and Grunwald [9], some of the misses in M_{np} will be serviced by the prefetch buffer and therefore appear to be fully hidden to the processor.[1] When removing the hidden L1 cache misses from M_{np}, we arrive at M_{pb}, the sequence of miss addresses of our application running on the architecture with prefetching to a buffer.

Let us consider M_p, the sequence of miss addresses with an architecture that prefetches directly into the L1 cache. It is now possible to prefetch useless data into the L1 cache which may, depending on associativity and replacement scheme of the L1 cache, introduce several new misses that were not present in M_{np}. In fact, with a pseudo-random replacement scheme, there may be little resemblance between M_p and M_{np}. To line them up and be able to determine the misses that were removed and those that were added, we need to associate them with the actual sequence of memory reference addresses (which includes cache hits and cache misses). This is a computationally expensive task because the number of memory references can be quite large. Instead of storing the actual memory references, the non-prefetching and prefetching simulation could also be run in parallel. Usually for these prefetching techniques, however, coverage is instead defined only in terms of the prefetching simulation, thereby avoiding the need to correlate misses from non-prefetching and prefetching simulations.

Before introducing the alternate definition of coverage, we will need to define some terms. Let $m = m_{\text{late}} + m_{\text{early1}} + m_{\text{early2}} + m_{\text{nopf}}$ be the total number of misses with its constituents defined as follows:

m_{late}: The number of L1 cache misses for which a prefetch request has already been issued, but the data has not yet arrived at the cache or buffer. A reduced miss time is associated with these misses.

m_{early1}: The number of L1 cache misses on data that was prefetched into the L1 cache or buffer, but replaced before being used.

m_{early2}: The number of L1 cache misses on data that was replaced by a prefetch, but only counted when the prefetched data itself has not been accessed at the time of the cache miss. These misses are caused by early prefetching and can only occur when prefetching directly into the L1 cache.

[1] In practice, the prefetch buffer and L1 cache are checked in parallel to ensure that an L1 miss and prefetch buffer hit will be serviced just as fast as an L1 hit.

$\boldsymbol{m}_{\textbf{nopf}}$**:** The number of all remaining L1 cache misses. These are cold misses and those replacement misses that are not counted by m_{early1} and m_{early2}.

Let $p = p_{\text{overhead}} + p_{\text{useless}} + p_{\text{early}} + p_{\text{late}} + p_{\text{hit}}$ be the total number of prefetch addresses generated by a prefetch algorithm. The number p consists of these quantities:

$\boldsymbol{p}_{\textbf{overhead}}$**:** The number of prefetch addresses that are found to be already in the L1 cache or buffer and can therefore be discarded. Mowry et al. label these prefetches as *unnecessary* [6]. These are of particular concern to software prefetchers because issue slots are wasted.

$\boldsymbol{p}_{\textbf{useless}}$**:** The number of prefetch addresses that are loaded into the L1 cache or buffer and then fall into one of these two classes. 1) Throughout the program execution the address is not referenced. The prefetched data may eventually be replaced from the cache or buffer. 2) The prefetched data is never referenced, eventually replaced, and then prefetched again. The subsequence prefetch itself may also be useless or fall into some other category.

$\boldsymbol{p}_{\textbf{early}} = \boldsymbol{m}_{\textbf{early1}}$**:** The number of prefetch addresses that are loaded into the L1 cache or buffer, then replaced, and finally missed on.

$\boldsymbol{p}_{\textbf{late}} = \boldsymbol{m}_{\textbf{late}}$**:** The number of prefetch addresses that are referenced by the processor before the data has been stored in the cache or a buffer.

$\boldsymbol{p}_{\textbf{hit}}$**:** All other prefetch addresses. These are exactly those that are prefetched into the L1 cache and subsequently referenced with an L1 cache hit. Note that a prefetch counted as p_{hit} may still be an early prefetch in the sense that it may have replaced some data that was referenced before the prefetched data was needed. This can only happen when the cache is at least two-way associative. The resultant miss would be counted by m_{early2}.

We can now define the prefetch coverage like so:

$$\text{Coverage}_{\text{Joseph}} = \frac{p_{\text{hit}}}{p_{\text{hit}} + m_{\text{late}} + m_{\text{early1}} + m_{\text{early2}} + m_{\text{nopf}}}$$

For prefetching methods that utilize a prefetch buffer, this definition is exactly the same as the one by Joseph and Grunwald since p_{hit} equals the miss references removed by prefetching and the denominator of the equation equals the number of misses of the base architecture without prefetching.[2] For simulations of architectures that prefetch into the L1 cache, this definition of coverage is relatively easy to compute because it does not attempt to correlate particular misses of a base architecture to misses of the prefetching architecture.

Mowry et al. use a different definition of coverage [5, 6]:

[2] Remember that m_{early2} equals zero when prefetching into a buffer.

$$\text{Coverage}_{\text{Luk}} = \frac{p_{\text{hit}} + m_{\text{late}} + m_{\text{early1}}}{p_{\text{hit}} + m_{\text{late}} + m_{\text{early1}} + m_{\text{early2}} + m_{\text{nopf}}}$$

In this definition, m_{early1} and m_{late} misses are counted as if they are a successful prefetch. The idea is that the addresses were computed accurately, but that the timing was wrong. There seems to be disagreement, whether coverage should be sensitive to the prefetch timing or not. We also observed that a late miss is sometimes counted as a fractional miss, based on the fraction of its completion time relative to the full miss latency [7].

When prefetching into the L1 cache (i.e. no prefetch buffer), the equation for coverage has one shortcoming. The misses in m_{early2} are those misses that were introduced as a side-effect of prefetching. If this happens frequently, then the denominator of the coverage equation is increased and hence coverage decreases. Poor prefetch timing and poor prefetch accuracy can therefore reduce the numerical value of coverage, even though coverage could be quite good in the intuitive sense (i.e. the removal of misses only). We are not aware of any prefetching results that use a coverage definition which excludes the m_{early2} misses.

Counting the quantities for m_{early1}, m_{early2} and p_{useless} can add a significant amount of complexity to the simulation code. For prefetching into the L1 cache (a similar technique works for prefetching into a buffer, although here only m_{early1} and p_{useless} are ever non-zero), we must use a data structure to remember, for each cache line, all the prefetched cache lines that were replaced there. The other quantities can be computed with less overhead.

Finally, we can also give an equation for accuracy:

$$\text{Accuracy} = \frac{p_{\text{late}} + p_{\text{hit}}}{p_{\text{overhead}} + p_{\text{useless}} + p_{\text{early}} + p_{\text{late}} + p_{\text{hit}}}$$

Here, a late prefetch is still counted as useful. Coverage and accuracy are sometimes inversely related to each other. Typically, coverage can be improved by more aggressive speculation on the prefetching addresses which in turn can reduce accuracy. This is illustrated graphically very nicely in Figure 5 of Joseph and Grunwald's paper [9].

2.2. Timeliness

The timeliness of prefetching requests is not a precisely defined numeric value. A perfectly timed prefetching request will complete shortly before the data is needed by an instruction. The perfect time for a prefetch to issue is not an instance in time, but a range in which no negative effects are incurred. This implies that we can have two other cases, early and late prefetching.

An early prefetch occurs when prefetched data arrives long before it is needed. This can replace data that is going to be referenced before the prefetched data. The processor will incur a miss that will be counted as part of m_{early2}. Furthermore, if the associativity of the cache is small, the prefetched data itself may be replaced before it is referenced. A subsequent miss on the data would be counted as m_{early1}. Early prefetching can therefore increase the number of cache misses and also bus traffic.

A late prefetch occurs when data has been prefetched, but it arrives after it is needed by the processor. Late prefetches do not fully hide a cache miss, but they do provide some benefit by reducing the cache miss time. They are counted by m_{late}.

To avoid the negative impact of early and late prefetching, some prefetching algorithms incorporate methods for controlling the time when a prefetch request is issued. Good timing, however, is difficult to achieve.

2.3. Load Latency

The average load latency is used by Roth et al. [12] to show the benefit of prefetching compared to a base architecture without prefetching. It is similar to comparing miss rates, however, it properly measures the reduced miss latency of late misses. It can also expose excessive prefetching within a particular memory system. With excessive prefetching, the bus bandwidth can become a bottleneck and miss latencies can increase. This requires accurate modeling of the memory bus. Excessive prefetching can slow down both cache hits and cache misses if access to cache tags is blocked frequently by the prefetch engine. Some prefetch techniques will therefore only use the tag memories when they are idle and will only issue prefetches when the memory bus is not used.

2.4. Bus Bandwidth

Prefetching increases the utilized bus bandwidth when accuracy is not perfect. Since bus bandwidth is generally a valuable resource that can limit performance, bus bandwidth increases due to prefetching are frequently measured [5, 7, 12].

Another aspect of bus bandwidth is that it is difficult to improve performance with prefetching for applications that already fully utilize a processor's memory bandwidth. Prefetching only hides cache misses, but does not eliminate memory transfers. While reordering of memory accesses can improve the available bandwidth (e.g., by exploiting page mode accesses or caches in lower levels of the hierarchy), these effects have not been studied, let alone be exploited, by cache prefetching mechanisms.

Bus bandwidth can therefore be used for computing an upper-bound of performance improvement of a prefetching technique. Performance can only increase until memory bandwidth is fully utilized. An alternate upper-bound for achievable performance improvement of prefetching is to assume a 0% cache miss rate [7]. Due to the fact that prefetching does not reduce bus bandwidth, the latter bound can be less accurate (i.e. lower) when bus bandwidth is close to fully utilized. However, assuming a 0% cache miss rate for computing an upper-bound can provide a more accurate result when the functional units are close to fully utilized.

3. PREFETCHING TECHNIQUES

There are several different prefetching techniques. As was already discussed, some prefetch into a L1 cache while others prefetch into separate buffers. Some prefetching techniques use the compiler to insert prefetch instructions into the code while others rely on hardware to issue prefetch requests. Prefetching done by the compiler, called software prefetching, can use profiling or code analysis to identify data to be prefetched. Hardware prefetching typically uses data cache miss addresses to compute the prefetching addresses, but some techniques also look at the instruction addresses or other information to determine which data to prefetch.

The most important distinguishing factor between prefetching techniques, however, is the memory reference pattern that they are designed to recognize and prefetch. We call the set of reference patterns that a prefetching technique was optimized for, the prefetch domain. With the exception of Markov prefetching, all the prefetching techniques are designed to handle a very particular memory reference pattern and are unable to prefetch other memory reference patterns.

The prefetch domain is an important characteristic of a prefetching technique because data accessed by a reference pattern outside the prefetch domain is not usually prefetched. This limits the achievable prefetch coverage. Furthermore, unsupported memory reference patterns can confuse the prefetch algorithm and reduce prefetch accuracy.

3.1. Memory reference patterns

Five elementary memory reference patterns have been identified in the literature. Most cache prefetchers are designed to prefetch data from only some of these recognized memory reference patterns.

Stride-0: These are memory references to isolated addresses that are not related to other addresses. For example, local and global variable accesses, array references using a constant index, and certain references to members of a structure. Caches are generally very effective for these memory references. Few prefetch techniques are able to preload these memory references to the caches because their addresses are difficult to predict.

Stride-1: We call a memory reference pattern that accesses strictly increasing or decreasing, adjacent memory address locations stride-1. Many array references are stride-1. Early prefetching techniques, such as Jouppi's prefetch buffers [10], were designed to prefetch data from this category of memory references.

Stride-n: These are memory reference addresses that are strictly increasing or decreasing and separated by a constant stride. It is a natural extension of the stride-1 reference pattern and has been exploited by numerous prefetching techniques [6–8, 11, 13–15]. The typical data structure that is accessed with a stride-n reference pattern is the array.

Linked: In a linked memory reference pattern, the data pointed to by one address is used to compute the subsequent address. In C notation, this occurs when the statement `ptr = ptr->next;` is executed in a loop. The offset will be zero only if `next` is the first element of the structure pointed to by `ptr`. Linked lists and tree data structures are accessed with a linked memory reference pattern. A number of techniques have been proposed to prefetch linked memory references [5, 12, 15, 16].

Irregular: Memory reference patterns that do not fall into one of the previous categories are irregular. For example, we consider a triangular memory reference pattern (i.e., stride increases by a constant on each access) as irregular. Few prefetching methods are able to deal with irregular memory reference patterns. The notable exception is prefetching using Markov predictors [9] that can handle some reference patterns that fall into this category.

Stride-0, stride-1, and stride-n have previously been defined by Baer and Chen [13]. The linked category is documented by Roth et al. [12]. Luk and Mowry [5] refer to the data structures that are accessed by linked memory reference patterns as recursive data structures.

3.2. Prefetching within Loops

A common abstraction is to look at prefetching in the context of a loop such as the one shown in Figure 1. We will refer to this loop as the work loop. The variable `p` can be an element of a linked data structure in which case the function `next()` would return `p->next`. Or `p` can be a pointer to an array and the function `next()` would return `p+1` or zero when the computation is finished. Let us assume that an element `p` can fit within a single cache line.

```
while(p) {
  work(p);
  p = next(p);
}
```

Figure 1. An abstract view of a loop that iteratively processes a data structure with elements `p`.

A prefetch algorithm should be able to bring `p` close to the processor before the function `work()` is called on `p` as indicated in Figure 2. Let us assume for this discussion, that the `prefetch(next(p))` request could also have been issued by the hardware instead of by the software. For a hardware prefetching technique, the prefetch statement can be thought of as representing when an action is taken by the hardware.

```
while(p) {
  prefetch(next(p));
  work(p);
  p = next(p);
}
```

Figure 2. A loop with non-overlapping prefetching.

Let us now discuss several possibilities about the effect of this prefetching statement in terms of the hardware capabilities and the relative speed of the functional units and memory.

3.2.1. Faster Memory

If the memory latency is less than the time it takes to complete `work(p)`, the memory latency can be fully hidden. The processor would never incur a cache miss on `p`.

3.2.2. Slower Memory without Overlap

Let us consider what happens if the memory latency is larger than the time to complete `work(p)`. Now, the processor will incur a cache miss on `p` with every iteration of the loop. The cache miss, however, will have a reduced service time, because the data is already in transit from a lower level of the memory hierarchy.

If memory operations cannot be overlapped or pipelined, then nothing can be done to further hide the memory latency. While prefetching was able to reduce the cache miss latency somewhat, it cannot fully hide it.

There are several reasons for why it may not possible to pipeline memory request. One may be that the hardware is not able to pipeline memory requests. However, with the increase in instruction level parallelism (ILP) and the increasing gap in memory and processor performance, many modern processor architectures are not only able to pipeline memory requests, but they can sometimes even issue multiple memory requests per cycle. Data dependences can also make it impossible to pipeline memory requests. This is commonly the case for linked data structures such as a linked list where `prefetch(next(p))` cannot be executed until `p = next(p)` has completed. It is also possible that the prefetching technique itself is not able to look ahead across more than one iteration and therefore would not pipeline memory requests even when there are no data dependencies.

3.2.3. Slower Memory with Overlap

For array and linked data structures, prefetching techniques exist to overlap memory requests. This can further hide memory latency. Figure 3 shows the general approach when the memory latency is twice as long as the time to complete `work(p)`.

```
while(p) {
  prefetch(next(next(p)));
  work(p);
  p = next(p);
}
```

Figure 3. A loop with overlapping prefetching.

For an array data structure, `next(next(p))` is equivalent to `p+2`. For linked data structures, Luk and Mowry have proposed techniques to obtain the address of `p->next->next` without first having to reference `p->next` [5]. We will discuss their technique shortly. In both of these case, the memory system must have at least two pipeline stages or be able to handle two memory requests in parallel.

As memory latency continues to lag behind processor speeds, it will become increasingly important for memory systems to handle multiple requests simultaneously and for prefetching methods to exploit this with techniques that prefetch data from several iterations ahead of the current iteration.

3.3. Pure Prefetching Methods

Most prefetching methods are designed to exploit either one of the two non-zero strided or the linked memory reference patterns, but generally not a combination. We call such methods pure prefetching methods and will discuss them in this section. In the next section we will introduce some of the hybrid prefetching methods that can be a combination of several pure prefetching methods.

3.3.1. Consecutive prefetchers

The prefetch techniques targeting to cover the stride-1 reference pattern are the consecutive prefetchers. This is the oldest group of prefetching techniques because their hardware complexity is low enough to make it feasible [17].

Smith; Gindele An early technique is one-block-lookahead (OBL). With OBL, a prefetch is always issued to the next cache line in the address space and it is brought directly into the L1 cache. A prefetch can be triggered by a memory reference (always prefetch) or a cache miss (prefetch on miss) [17]. Always prefetch is a brute-force method that prefetches the right data for stride-1 memory references, but it will also generate a large number of overhead prefetch requests (p_{overhead}). Overhead prefetches are undesirable because a cache tag lookup must be done before they can be discarded. Prefetch on miss avoids this overhead, but it can only prefetch half of the stride-1 reference pattern. Every other memory reference will result in a cache miss, because prefetch on miss does not issue a prefetch following a prefetch hit. A slight extension, OBL with tagged prefetch, uses a tag bit per cache line to issue a prefetch only on the first reference to a cache line [18]. OBL with tagged prefetch has a coverage similar to the always prefetch method, but generates far fewer overhead prefetches [17].

Jouppi Jouppi proposed to prefetch into a separate prefetch buffer called a stream buffer [10]. His technique uses OBL with prefetch on miss, but instead of issuing a single prefetch, consecutive cache lines are prefetched and stored in the stream buffer. The stream buffer is organized like a queue that is filled by prefetching requests and emptied by cache miss requests. The data are copied to the L1 cache on a cache miss and also supplied to the processor. The stream buffer is flushed and prefetching is restarted with a new address when the item popped from the stream buffer does not match the next cache miss. A single stream buffer works well with an instruction cache, but not so well with a data cache. The sequence of cache misses generated by an L1 data cache are rarely consecutive for a long period of time and will cause the stream buffer to be restarted frequently. Jouppi found that the use of multiple stream buffers, operating in parallel, works much better with a data cache.

Stream buffers had two major contributions to prefetching. First, the use of a separate buffer avoids the pollution of the cache with data that are never referenced ($m_{\text{early2}} = 0$). Second, even with the absence of non-blocking caches and out-of-order execution, stream buffers overlap multiple prefetches from a single reference and are therefore able to hide memory latency better (Section 3.2.3) than other OBL techniques.

3.3.2. Stride prefetchers

Consecutive prefetchers are not able to efficiently prefetch for the stride-n memory reference pattern, especially for larger n. In addition, with the exception for a filtering technique proposed by Palacharla and Kessler [11], consecutive prefetchers naively assume that all memory references are consecutive. These effects hurt the prefetch accuracy of the consecutive prefetchers. Generally, stride prefetchers scan the memory references to find candidates for stride prefetching. If no candidate is found, no prefetching will be initiated. This improves the prefetching accuracy of stride prefetchers relative to consecutive prefetchers. In a comparison of prefetch accuracies between a variety of prefetching techniques, Joseph and Grunwald show this effect nicely. Whereas the stride prefetchers appear to have almost perfect accuracy, the number of useless misses for the consecutive prefetchers typically exceeds 10% of the number of cache misses [9].

Stride prefetchers fall into two general categories. One is a software prefetching technique that uses a compiler to analyze array references in the program loops and inserts prefetch instructions as needed. The other uses a hardware component that detects when the repeated execution of a particular memory instruction issues stride-n memory references and uses this information to predict and prefetch future references.

Mowry, Lam, and Gupta Mowry et al. [6] have developed a software prefetching technique for array references whose indices are affine (i.e., linear) functions of loop indices. While such a memory reference pattern does not need to fit a stride-n reference pattern, it typically can be described by a small number of stride-n reference patterns. This technique requires hardware support in the form of a prefetch instruction that works similar to a load instruction, but the data referenced are not actually stored in a register. Prefetch instructions are not allowed to cause exceptions (e.g., a page fault). In the indiscriminate version, the compiler will insert prefetches for every affine array reference. Software pipelining is used to carefully place prefetches early enough so that the prefetch completes just in time for the array reference. The selective version only inserts prefetches when the compiler predicts that the array reference instruction will cause a miss. It does so by considering the reuse of references and the loop iteration counts relative to the cache size. Loop splitting is used to insert prefetches into only those iterations where the compiler determines that misses are likely.

The importance of the selective software prefetching technique over the indiscriminate technique is that it can reduce overhead prefetches (p_{overhead}). Mowry et al. [6] observed that it reduced the average percentage of overhead prefetches from 79% to 29% with only a small loss in the prefetch coverage. Overhead prefetches are more costly to software prefetchers, because each prefetch occupies an instruction slot and instruction bandwidth that could have been used for executing something useful. In hardware prefetching, techniques the main cost of overhead prefetches is an increased pressure on the cache tag memory that can be overcome with banking or multiporting the tag memory.

Fu, Patel, and Janssens The hardware stride prefetchers use a hardware structure as shown in Fig. 4 for finding stride-n memory references. The simplest implementation of this scheme was proposed by Fu et al. [14]. They named the hardware structure the stride prediction table (SPT). The SPT only uses a valid bit for state information and has no Times and Stride entries. All memory reference instructions are looked up in the SPT and a new entry is created if none is found. If an entry is found, the difference in the Previous Memory Address and current memory address is used as a stride predictor. The stride predictor added to the current memory address is used as a candidate prefetch address. Finally, the Previous Memory Address is updated with the current memory address.

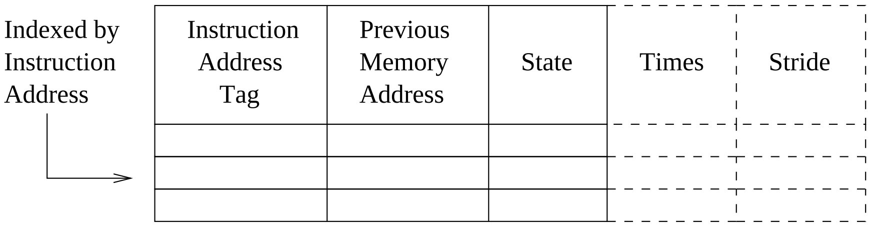

Figure 4. General form of the data structure used in hardware stride prefetchers to detect stride-n memory references.

A candidate prefetch address will be issued as a prefetch to the memory system 1) when the memory instruction causes a miss, 2) when the memory instruction causes a hit, or 3) always. Generally, the prefetch on miss has the same problem as the prefetch on miss OBL technique. Only every other memory reference is prefetched. Furthermore, the techniques mentioned so far will not issue a prefetch more than one memory reference ahead and are therefore unable to issue deeply pipelined memory requests (Section 3.2).

Chen and Baer Chen and Baer [4] proposed a scheme that adds two significant features. First, they store the last known stride in the reference prediction table (RPT, another name for the SPT) and they use a two-bit state field instead of a single valid bit. The state field contains information about how stable the stride has been in the past and affects the choice of prefetch addresses. Prefetches will not be issued when the strides between the last three memory addresses of a memory reference instruction are different. Their second innovation is a lookahead program counter (LA-PC) that is updated each cycle and uses a branch prediction table (BPT) to predict the direction of branches. The LA-PC is designed to run ahead of the PC. Lookups in the reference prediction table (RPT) are done by both the PC and LA-PC. RPT updates are controlled by lookups with the PC and prefetches are issued from lookups with the LA-PC. An additional entry Times in the RPT (Fig. 4) stores the number of iterations the LA-PC is ahead of the PC for a particular memory reference instruction. A candidate prefetch address is computed by adding the product of Times and Stride to the Previous Memory Address.

The use of the added state information is designed to improve the prefetch accuracy. The use of the LA-PC is designed to better control the prefetch distance and should reduce early and late prefetches. From the view of the work loop (Section 3.2), the LA-PC allows prefetches to be issued more than one loop iteration ahead of a memory reference and consequently can more aggressive pipeline memory references. They found that the smallest number of memory cycles per instruction (MCPI) occurs when the distance of the LA-PC ahead of the PC is not allowed to grow much beyond the number of cycles needed to access the next level in the memory hierarchy. Any larger distance would increase the number of early prefetches.

Pinter and Yoaz As superscalar processors had become the architecture design of choice for general-purpose processors in the 80s and 90s, a new challenge was recognized for cache prefetching. A superscalar processor can execute several instructions per cycle and issue possibly multiple memory requests per cycle. This increases pressure on the cache tags to determine if memory references hit or miss in the cache. Consequently, fewer idle cycles are available for discarding overhead prefetch requests and fewer prefetches can be issued without interfering with the normal processor execution. Pinter and Yoaz [7] addressed this problem with their Tango architecture by using a filter cache that caches the last few accesses to the tag memory. They also proposed a new LA-PC mechanism that is able to run ahead of the PC much more rapidly and is able to find more memory reference instructions per cycle than the scheme used by Chen and Baer [4].

The new LA-PC, referred to as a pre-PC, is enabled by extending the branch target buffer (BTB). Because the pre-PC advances from branch to branch, instead of from instruction to instruction as done by the LA-PC, it can advance much more rapidly. After having been advanced to the next branch, a specially indexed reference prediction table for superscalar processors (SRPT) is searched associatively with the pre-PC for all memory reference instructions from the current branch to the next branch to determine prefetch candidates. They simulated an SRPT that returns up to two memory reference instructions per cycle. For each memory reference instruction found, a prefetch is issued using the same state machine logic introduced by Chen and Baer [4]. After all memory references found in the SRPT have been processed, the pre-PC is advanced to the next branch. Just as done by Chen and Baer, the distance between pre-PC and PC is limited to roughly the memory latency to avoid early prefetching. Apart from early prefetching, predictions across multiple branches can also reduce the prefetch accuracy.

To allow branch-to-branch updates of the pre-PC, additional entries are needed in the BTB as shown in Fig. 5. The key entries to allow updates of the pre-PC are the T-Entry and NT-Entry fields. They contain indices back into the BTB for the first branch on the taken and not-taken paths, respectively. Because the BTB is also accessed by the regular PC for normal branch prediction, some fields must be dual ported as shown in Fig. 5. In addition, BTB replacements are more costly, because all BTB entries that point to the replaced BTB entry (via T-entry and NT-entry fields) must be updated. Pinter and Yoaz [7] solved this by performing a fully associative search on the T-entry and NT-entry fields on each BTB replacement. Since the update can be done locally within each BTB entry, the time penalty is tolerable, although there is certainly an increased hardware cost for full associativity.

The SRPT is identical to that used by Chen and Baer with one additional requirement. Since their goal is to be able to use the pre-PC to find several memory references in a single cycle, they implement a fully associative search on the SRPT using the BTB index of the current pre-PC. Although such a fully associative search would return potentially all memory reference instructions between the pre-PC and the previous branch, an index added to the

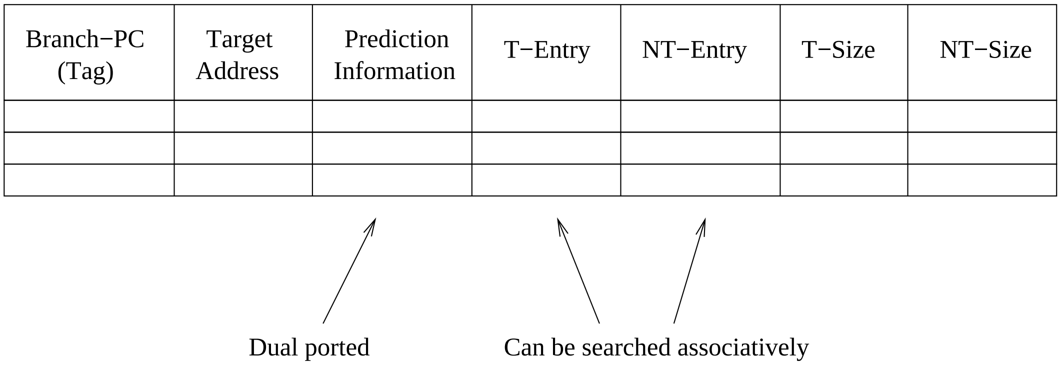

Figure 5. Branch target buffer (BTB) used for advancing the pre-PC.

tag limits it to only two per cycle. The only reason for this limit is to reduce the hardware cost of having to lookup and issue a larger number of prefetches.

The second key innovation of the Tango architecture is its use of a filter cache. It is a FIFO queue that keeps the address of the last six cache lines that produced a hit in the cache. A prefetch address that is found in this queue is an overhead prefetch and can be discarded without having to do a cache tag lookup. The filter cache can cut the number of cache tag lookups from overhead prefetches in half.

3.3.3. Recursive prefetchers

Recursive prefetchers that prefetch linked memory references have two primary challenges. First, like stride prefetchers, recursive prefetchers must be able to detect the memory reference pattern to be prefetched. Second, overlapped prefetching is difficult due to the data dependences inherent in the linked memory references.

The key technique used in detecting a linked memory reference pattern can be found by looking at the C statement `p = p->next;` which is commonly used for linked list traversals. When executed once, `p` will be loaded with an address. When executed again, the previous address serves as the base address for loading the next address. The important relationship is that the base address of one load was produced by a load from the previous loop. This is very different in a strided memory reference pattern where the base address is computed arithmetically from a base address that is constant for all memory references. All recursive prefetchers look for these dependencies across different load instructions to find the linked memory reference pattern. We will look at two hardware prefetching methods first and a software prefetching approach afterward.

Mehrotra and Harrison The linked list detection scheme proposed by Mehrotra and Harrison [15] is the simplest in design. They extended the reference prediction table, such as the one used by Chen and Baer [4], by two more fields as shown in Fig. 6. The linear stride field is the traditional field for detecting a stride-n reference pattern. The Indirect Stride and Previous Data Loaded fields are added for detecting linked memory references. The only linked memory references that can be detected by this technique are those generated by self-referential updates, such as in the C statement `p = p->next;`. The Previous Data Loaded field holds the actual memory contents loaded by the previous execution of the load instruction. During each cycle, the Indirect Stride value is updated with the difference between the Previous Data Loaded value and the memory address of the current execution of the load instruction. In our C statement example, this will equate to the constant offset of the `next` field within the structure pointed to by `p`. A linked memory reference pattern is detected when the indirect stride field is found to be constant between multiple executions of the load instruction.

Indexed by Instruction Address

Instruction Address Tag	Previous Memory Address	Previous Data Loaded	Linear Stride	Indirect Stride	State

Figure 6. Reference prediction table for detecting stride-n and linked memory reference patterns.

Roth, Moshovos, and Sohi Although simple and easily implemented, the method by Mehrotra and Harrison [15] has the disadvantage of only considering self-referential memory instructions. Roth et al. [12] proposed a different scheme that is able to detect linked memory references when the producer of the base address is not the same as the consumer of the base address, an example of which is shown in Fig. 7. It takes two separate hardware tables to accomplish this. Figure 8 shows these two tables. The values loaded by memory instructions are cached in the potential producer window (PPW) as possible candidates for memory instructions that load (produce) a base address. Also, for each memory instruction a lookup is done on the PPW indexing it with the base address of the current load. If a match is found, the PC of the current memory instruction is considered to be the consumer and the PC of the memory instruction of the matching PPW entry is considered the producer. The producer and consumer PCs are stored in the correlation table (CT) together with a template of the consumer memory instruction (Fig. 8). The template is necessary for generating candidate prefetch addresses. For typical memory instructions, the template would only include the offset of the memory instruction.[3]

```
while(p) {
  p = q->next;
  q = p->next;
}
```

Figure 7. An example control structure for RDS traversals.

Prefetching addresses are generated from the information stored in the CT. When the PC of a load instruction hits in the CT, it is known to have been a producer in the past. The template information from the CT and the

[3] The offset of the memory instruction is equal to the offset of the `next` field within the structure pointed to by `p`.

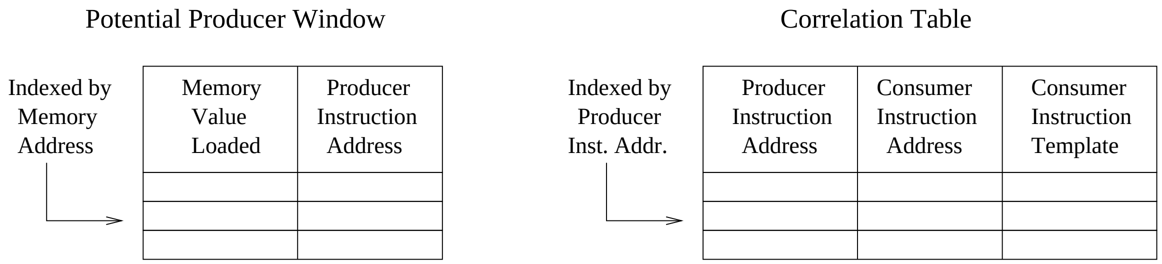
Potential Producer Window
Indexed by Memory Address
Memory Value Loaded
Producer Instruction Address
Correlation Table
Indexed by Producer Inst. Addr.
Producer Instruction Address
Consumer Instruction Address
Consumer Instruction Template

Figure 8. Potential Producer Window (PPW) and Correlation Table (CT) used for detecting linked memory references.

data returned from the current load instruction produce a memory address that is used as a prefetch address. If the prediction is correct, the consumer recorded in the CT will load this prefetch address at some point in the future. Lacking any timing information, the prefetch address is immediately entered into a prefetch requests queue (PRQ) and issued to the memory system during an idle cycle. As in Jouppi's stream buffers [10], prefetched data are stored in a buffer, which reduces the penalty of early prefetching.

Prefetched data may themselves be producers and could therefore also be looked up in the CT to generate additional prefetches. Roth et al. [12], however, did not perform such additional prefetches because it increases the risk of early prefetches. They also argued that there may be little benefit to prefetch further ahead. To understand this, we need to consider the whole loop encompassing the memory instruction that generates the prefetches. If the memory latency is less than the time to execute one iteration of this loop, then the memory latency can be fully hidden and there is no benefit of prefetching ahead further. If the memory latency is longer than the time to execute one iteration, then memory instruction and prefetch will complete at the same time and it makes no difference which of the two triggers the subsequent prefetch. Another way to look at this is that it is impossible to overlap prefetches due to the data dependencies. This places a limit on how many prefetches can be hidden (Section 3.2.2).

They noted that their argument against generating these additional prefetches ignores cases that may not fit the simple loop model. For example, not all loops may be of equal length, which can make it beneficial to prefetch ahead to effectively borrow time from one loop to use in a later loop.

Luk and Mowry Essentially, the same approach for finding linked memory references can be taken by a software prefetcher. However, since addresses are not known at compile time, the compiler instead looks at the data types. Linked memory references require that one data type structure contains an element that points to some other (or the same) data type structure. The recursive prefetcher, a software prefetcher, proposed by Luk and Mowry [5] calls these data types, recursive data structures (RDS). Their software prefetcher recognizes recursive data structures and also a set of control structures that traverse an RDS. Figure 1 is one example of such a control structure. Here `p` must be an RDS and the function `next()` can be both a function or an explicit dereference such as `p->next`. Their scheme can detect complicated control structures. For example, it can detect RDS traversals within recursive function calls and allow multiple recursive calls sites as they are needed in tree traversals. In addition, their algorithm can detect traversals through recursive data structures of varying types as shown in Fig. 7.

Detecting control structures of RDS traversals is comparable to finding producers of base addresses in the scheme proposed by Roth et al. [12]. Either scheme appears to have certain cases of finding base address producers that the other may not be able to detect. For example, Fig. 9 shows a control structure supported by the software prefetcher. The hardware scheme will not detect the dependence between producer and consumer if the addition of the base address with the offset of the `next` field in `p` is not embedded in the memory load. That is, the consumer instruction template cannot be specified and no correlation between producer and consumer could be found. Furthermore, the size of the PPW is limited and dependencies that are far apart may not be detected. The hardware scheme is also more likely to provider false matches when a correlation between producer and consumer occurs by chance without an actual relationship existing between the two. On the other hand, the hardware scheme may be able to detect certain base address producers that the software scheme misses because of a programming style that does not match any control structures for RDS traversals recognized by the compiler optimization stage in the software scheme.

```
while(p) {
  p = f(p);
}
```

Figure 9. Another example control structure for an RDS traversal.

As soon as the base address is computed, a prefetch instruction can be inserted for the next element. This is shown on a tree traversal example in Fig. 10. Luk and Mowry [5] call this greedy prefetch scheduling, and it is functionally very similar to the hardware prefetcher proposed by Roth et al. [12]. The primary difference is in the detection of base address producers as previously discussed and in the placement of prefetch calls. Whereas the hardware prefetcher will issue prefetches almost instantaneously, the software prefetcher will usually issue the prefetch at the top of the next loop iteration or function call as done in Fig. 10. The remaining differences are those that generally separate hardware and software prefetching schemes. The hardware schemes have a higher hardware cost and the software schemes have the additional instruction overhead. Also, the technique by Roth et al. [12] prefetches into a buffer whereas Luk and Mowry's scheme [5] prefetches directly into the L1 cache. This should make the software scheme more vulnerable to early prefetching, a particular concern with greedy prefetch scheduling, due to the lack of prefetch timing control.

Although functionally similar, Luk and Mowry [5] motivate their greedy prefetch scheduling from a different perspective than Roth et al. [12]. A difficulty for recursive prefetchers is the dependencies between successive prefetches that make it impossible to overlap prefetches, thus limiting the amount that memory latencies can be hidden. As mentioned in Section 3.2.3, in order to overlap prefetches, `p->next->next` must be computed without having to load the intermediate elements of the data structure. While this appears impossible, Luk and Mowry introduced three software prefetching mechanisms that support this with varying degrees of success. The first one is

```
f(treeNode *t) {                 f(treeNode*t) {
  treeNode *q;                     treeNode *q;
                                   prefetch(t->left);
                                   prefetch(t->right);
  if( test(t->data))               if( test(t->data))
    q = t->left;                     q = t->left;
  else                             else
    q = t->right;                    q = t->right;
  if( q!= NULL )                   if( q!= NULL )
    f(q);                            f(q);
}                                }
```

Figure 10. The left side shows the code before and the right side after insertion of prefetch calls.

the greedy prefetch scheduling.

Greedy prefetching cannot overlap memory references for linked list traversals. However, for tree traversals, there can be some overlap. In particular, in a k-ary tree, up to k prefetches can be issued in parallel as illustrated by Fig. 10, assuming that all pointers are located in one cache line. Contrary to the argument by Roth et al., therefore, Luk and Mowry found conditions under which it can be beneficial to let prefetched data themselves initiate further prefetches.

Figure 11 duplicates a figure from Luk and Mowry's paper, which shows the sequential ordering of prefetches on a binary tree using greedy prefetching. The nodes in bold are those covered by greedy prefetching and result in a prefetch hit as long as they are not evicted from the cache due to early prefetching or conflicts. Nodes 6, 10, 12, and 14 resulted in cache misses. However, if the prefetch of node 3 causes the prefetches of nodes 6 and 7, then node 6 would not be missed subsequently. This effect would be difficult to achieve in Luk and Mowry's greedy prefetcher because initiating prefetches from other prefetches cannot be done with their software prefetcher. However, it could be possible with the prefetched proposed by Roth et al.

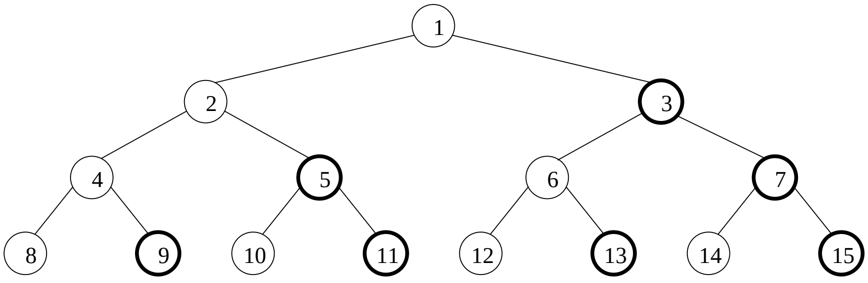

Figure 11. Ordering of prefetches in greedy prefetching using a pre-order traversal of a binary tree. Bold tree nodes are those that are prefetched and should result in a prefetch hit. All other nodes will be missed on.

A remaining question is how to prevent an explosion of prefetches that can occur due to the fanout present in a tree. Some mechanism is needed to prevent prefetches to be issued beyond a certain depth. There can also be problems with early prefetching. For example, node 3 in Fig. 11 would be prefetched almost at the beginning of the tree traversal, but it would not be referenced until about half-way through the tree traversal (assuming pre-order traversal). If the prefetch buffer is small, chances may be high that node 3 will be replaced before it is referenced.

With a lack of better timing control over prefetches and the possibility of prefetching data that are never referenced, Luk and Mowry found greedy prefetching to have a relatively bad prefetching accuracy. Coverage was generally good, except with applications where a larger number of the misses were caused by scalar and array references. The poor coverage was an important reason for why roughly one half of the applications had only about a 2% reduction in execution time. The other half had reductions in execution time from 4% to 45%.

Luk and Mowry proposed two additional prefetching techniques that are targeted at improving the prefetching accuracy over greedy prefetching. The first is history-pointer prefetching that keeps track of the traversal ordering of an RDS and stores this information with the RDS by adding to it a new pointer. For each node in the RDS, the history pointer provides the value of `p->next->next->next` assuming a history depth of three and a linear traversal of a linked list. It takes a full traversal of the RDS for all history pointers to be initialized. If the traversal ordering does not change during subsequent traversals, then a prefetch on the history pointer will effectively break the dependence chain and make it possible to prefetch `p->next->next->next` without having to visit intermediate nodes. History pointers can be updated on subsequent traversals to accommodate slight changes to the traversal ordering. Drastically different traversal orderings will cause the wrong data to be prefetched most of the time. With a depth of three, up to three prefetches can be pipelined. The cost of this approach is the time and space required to maintain the history pointers.

Luk and Mowry only tested the performance of history-pointer prefetching on an application that did not change its tree traversal over the course of the program execution. Their compiler technique is currently not automated and must be applied by hand. Under these conditions, history-prefetching performs significantly better than greedy prefetching. Coverage and accuracy improved, but there are still a significant number of overhead prefetches because data are indiscriminately prefetched. The locality analysis of the strided software prefetcher developed by Mowry et al. [6] cannot as such be applied to the recursive data structure, making selective prefetching difficult.

The final technique is data-linearization prefetching. Luk and Mowry demonstrated its potential, but did not seem to have an automated compiler optimization for this technique. The technique cannot be applied to all applications. The basic idea is to map the recursive data structures in the same sequence to memory, in which it will be traversed. This reduces the problem to prefetching a sequentially accessed array. There is no solution offered for how to handle different traversal orders. Luk and Mowry optimized the application to the dominant traversal order. They do not discuss how to handle accesses to the RDS that are not in a simple forward-traversal order. Compared to greedy prefetching, data-linearization prefetching has a lower coverage on both applications that were tested. This is because only the dominant traversal order is optimized. Overhead prefetches are, however, removed and execution time is improved over greedy prefetching.

3.3.4. Markov prefetcher

The Markov prefetcher remembers past sequences of cache misses [9]. When it finds a miss that matches a miss in a remembered sequence, it will issue prefetches for the subsequent misses in the sequence. This gives Markov prefetching the ability to prefetch any sequence of memory references as long as it has been observed once. However, it also means that the Markov prefetcher cannot deal well with dynamic memory reference patterns as they can appear in programming languages that frequently allocate and deallocate memory (e.g., Java, C++, Scheme).

Joseph and Grunwald [9] implemented a hardware prefetcher that approximates a Markov model. The main data structure is a state transition table (STAB) indexed by a miss address that stores a number of prefetch address predictors (1, 2, 4, and 8 are modeled). This data structure is large. They evaluated their prefetcher with a 1-Mbyte STAB with a 2-Mbyte L2 cache. They compared their performance to an architecture without prefetching, but with a 4-Mbyte L2 cache instead.

During program execution, a history of the past n miss addresses is maintained. When a new miss is encountered, two actions take place. First, the oldest miss address in the history list (n misses in the past) is used to index the STAB and the current miss is stored there as a new prefetch address predictor. Entries in the STAB are replaced using an LRU scheme. The parameter n can be used to adjust the prefetch distance, and it represents the maximum overlap of prefetches that can be achieved. Second, the current miss address is also looked up in the STAB. All prefetch address predictors found in this table entry are sent to the prefetch request queue. Each prefetch address predictor has a different priority based on its position in the STAB. Lower priority requests may be discarded from the prefetch request queue if it becomes full. The Markov prefetcher uses a prefetch buffer to store recent prefetches. When an L1 cache miss hits in the prefetch buffer, the data will be copied to the L1 cache, but will not be removed from the prefetch buffer. Instead, the data in the prefetch buffer are moved to its head. The prefetch buffer is managed like a FIFO and prefetched data arriving from the L2 cache will replace the least recently used entry. Keeping data that are transferred to the L1 cache in the prefetch buffer should have an effect similar to a victim cache [10]. If the data are evicted soon from the L1 cache, they can be repeatedly restored from the prefetch buffer. Unfortunately, no evaluation was done to show if this is more effective than Jouppi's organization, where data that are copied to the L1 cache are also removed from the stream buffer.

The number of prefetch address predictors modeled greatly affects coverage and accuracy. Accuracy becomes worse quickly as the number of prefetch address predictors is increased, whereas coverage increases steadily. The coverage of the Markov prefetcher obtained on their benchmark applications was about 50% when using two or four prefetch address predictors.

Perfect coverage is difficult to achieve with the Markov prefetcher because a sequence of misses must be seen first before it can be predicted. This is a disadvantage that most consecutive, strided, and recursive prefetchers do not have.

3.4. Hybrid Prefetching Methods

Amdahl's law [1] tells us two limitations of cache prefetching. First, application speedup is limited by the portion of execution time caused by cache miss stalls. Moreover, for pure prefetching methods, only those cache misses that are covered by the prefetch domain can be hidden. This further limits potential speedup. Clearly, the first limitation is not a limitation really, but the reality of why prefetching has become a performance improving technique in the first place. The second limitation, however, clearly limits all pure prefetching techniques. It would be quite unsettling for a processor architect to have to choose between a stride prefetcher or recursive prefetcher, knowing very well that what really is needed is both designs.

For example, a stride prefetcher will not be able to prefetch misses caused by linked memory reference patterns, unless the linked data structure is laid out sequentially in memory, in order of the memory accesses. What is needed are techniques that are effective for all cache misses or methods for combining pure prefetching techniques. The Markov prefetcher is an example of the first class, but because Markov prefetchers must learn, they are generally less effective at prefetching strided and linked memory reference patterns than the other pure prefetching techniques. We call the second class the hybrid prefetchers. In this section we will give examples of and describe techniques for building hybrid prefetchers.

3.4.1. Instruction Prefetchers

Instruction miss rates are traditional thought of as less important to processor performance than data miss rates. Recent research by Maynard et al., however, shows that they may be just as important or maybe even more important than data references [19]. While memory references of instructions can be treated just as data references from a prefetching perspective, they exhibit a more predictable access pattern that can be exploited. Fortunately, many L1 caches are separated between data and instructions making it easier to incorporate different prefetch techniques for data and instructions.

Instructions are generally fetched from memory using a stride-1 memory access pattern. A consecutive prefetcher would work very well. The trouble occurs, however, at branches, where the instruction sequence may continue at a different location. Luk and Mowry introduced a nice example of a hybrid prefetcher that is designed to prefetch instructions consecutively, but also handles branches well [20]. Their cooperative prefetcher uses two separate prefetchers.

The first prefetcher is a next-N-line hardware prefetcher [17] which is similar to OBL, but instead of fetching the next cache line, it fetches the next N cache lines (N being typically in the range of 2 to 8). The next-N-line prefetcher may issue useless prefetches when it crosses branch points. To avoid useless prefetches, Luk and Mowry designed a prefetch filter in the L2 cache that recognizes when prefetched cache lines are not being referenced. The prefetch filter will stop prefetching at those cache lines in future accesses.

The second prefetcher is a software prefetcher. The compiler will insert prefetch instructions ahead of branch points if they are not covered by next-N-line prefetching and if they are estimated to cause cache misses [21].

Luk and Mowry compare their hybrid technique to next-N-line prefetching on its own and Markov prefetching. Next-N-line prefetching performs well on sequential instruction sequences, whereas Markov prefetching can handle sequential sequences and branch points well. Markov prefetching, however, must learn a sequence before it can prefetch it. On the benchmark applications, cooperative prefetching generally reduces the execution time by about 10%. The better of next-N-line prefetching and Markov prefetching improve execution time by only about 5%.

3.4.2. Data Prefetchers

Mehrotra and Harrison came up with an idea of extending the stride prediction table to also be able to detect linked references [22]. We discussed their prefetcher in Section 3.3.3. While the linked memory reference detection scheme is not as general as those proposed by Roth et al. and Luk et al., it is simple and integrates stride-n and linked prefetching with little overhead. Unfortunately there are not enough results available to evaluate the effectiveness of their technique [15].

Joseph and Grunwald introduce a more general approach of creating hybrid prefetchers [2]. The idea is to use multiple unmodified pure hardware prefetchers. They define two modes of operation, serial and parallel. In parallel operation, all prefetchers get access to the system resources and issue prefetches independently from the other prefetchers. The only modification made is to prevent prefetchers from issuing identical prefetches to the memory system. In serial operation, the most accurate prefetcher is allowed to issue a prefetch first. If it cannot issue a prefetch, then the next accurate prefetcher is queried. Prefetcher accuracy is determined statically and Joseph and Grunwald consider stride prefetchers to be most accurate, then Markov prefetchers, and finally consecutive prefetchers.

It appears that a little more work is needed to effectively use a serial hybrid prefetcher. First, all pure prefetchers should be able to have access to all memory references and cache misses (and any other information they need) to be able to build their database contents. Joseph and Grunwald do not describe if they use this technique or perhaps some other method. Furthermore, it must be clearly defined what it means that a prefetcher is able to issue a prefetch. For a stride prefetcher this can be if there is a hit in the SPT or RPT with the state indicating that a stride-n reference pattern was found. For a recursive prefetcher, finding an actual base address producer can constitute that it is able to make a prefetch. For Markov prefetching a hit in the STAB gives little certainty about the accuracy of the prefetch address predictors. When used in conjunction with a hybrid prefetching scheme, it may be beneficial to use only the most likely prefetch address predictor(s) instead of all of them. Finally, consecutive prefetchers, will always claim to have a prefetch address. Clearly there is a problem if multiple such prefetching techniques are put in series (only the first would ever issue a prefetch). In addition, with the use of a stride prefetcher, there is certainly some question about the benefit of a consecutive prefetcher.

Joseph and Grunwald found a remarkable improvement in prefetch coverage when using parallel and serial hybrid prefetchers compared to stride, Markov, and consecutive prefetchers on their own. Unfortunately, the parallel prefetcher is also highly inaccurate. This is perhaps not such a surprising result. The serial prefetcher had mixed results with prefetch accuracies ranging from nearly as bad as the parallel prefetcher to better prefetch accuracies than Markov and consecutive prefetchers alone.

4. MULTIMEDIA PROCESSING

Multimedia applications have become an important workload on modern personal computers and various embedded applications. To give some examples, multimedia applications fall into categories such as graphical user environments, modern television and communication systems, computer games, and medical visualization and diagnosis systems. We will restrict our discussion mostly to the image processing domain, because it is an area the author of this paper is more familiar with and because other work in this area tends to focus on image processing, too. Many basic image processing operations, such as convolution filters and discrete cosine transformations, are also widely used in other areas of multimedia processing (e.g., audio processing), therefore, our discussion may apply equally well to those areas, too.

We will look at two basic classes of multimedia architectures, the general purpose processors with multimedia extensions and the dedicated mediaprocessors. While there are also hardwired architectures that can perform a particular multimedia processing task, these systems are not programmable and contain special purpose memory systems.

General purpose processors with multimedia extensions and dedicated mediaprocessors both make use of the multimedia functional unit. Two basic characteristics of multimedia processing has created the need for a new functional unit. First, multimedia processing contains a large amount of data-level parallelism. For example, in pixel-wise image addition, in theory, all pixels-pairs of the two source images could be added in parallel to create the destination image. Second, for many multimedia applications, a precision of 8 or 16 bits is sufficient. The multimedia functional unit exploits the parallelism by bundling multiple data elements into a wide machine word and operating on the data elements in parallel in the style of single instruction, single data (SIMD) processing. Lee refers to this type of architecture as the microSIMD architecture and argues that it is the most cost-effective parallel architecture choice for multimedia processing [23].

While multimedia functional units appear very effective, there is less confidence in the memory system. Generally, the memory system remains unchanged when a general purpose processor is extended for multimedia processing. Mediaprocessors take a somewhat different approach that we will describe later. First we will motivate why caches, a key component in the memory system of a general purpose processor, are not a perfect match for multimedia processing. Taking the image addition function as an example, we find that the memory access pattern has no temporal locality, but an abundance of spatial locality. Little temporal locality is quite common in multimedia functions because the input data is often needed for only a brief period of time. This implies that during the program execution, a large amount of data is loaded into the cache. While a cache exploits the spatial locality quite well, filling it with multimedia data that will never be referenced again, does not make efficient use of the cache space. In addition, other useful data, such as local variables and constant data, might be evicted from the cache by the multimedia data.

4.1. General Purpose Processors with Multimedia Extensions

Most high-performance general purpose processors are extended for multimedia processing or plans to do so have been announced (e.g., 3DNow! [24], AltiVec [25], MAX-2 [26], MMX [27], VIS [28]). Extending a general purpose processor for multimedia processing means that one or more multimedia functional units have been added to the processor. The multimedia functional units may operate on the general purpose registers, floating point registers, or their own set of multimedia registers. The memory hierarchy generally remains unchanged. Multimedia data is loaded to the processor with memory reference instructions and it passes through the same memory hierarchy as all other memory references.

Effective use of multimedia function units is difficult because current compilers for general purpose processors do not automatically use multimedia function units. The programmer can rewrite the program to call special assembly libraries that utilize the multimedia function units. For best performance improvements, however, parts of the program must be rewritten in assembly to make explicit use of the multimedia instructions [29].

A study by Ranganathan et al. showed that many common image and video processing kernels exhibit cache miss stall times equal to over half of the execution time on general purpose processors enhanced with multimedia extensions [30]. Cache prefetching designed to hide those cache misses, can therefore provide quite significant improvements in execution time.

4.2. Mediaprocessors

Dedicated mediaprocessors are usually organized as VLIW processors [31]. Compared to a superscalar design, this reduces the hardware complexity because instruction scheduling is shifted from the hardware to the compiler. Reduced hardware complexity translates to reduced cost and power consumption. Reduced cost and power consumption is important for mediaprocessors because they are predominantly targeted at the embedded market instead of the personal computer market. In addition, mediaprocessors with over twenty functional units are not uncommon [31], making scheduling a particularly complex task. While mediaprocessors also contain traditional integer function units, they usually contain a larger percentage of multimedia function units than multimedia-extended general purpose processors. Mediaprocessors also typically support multiple memory operations per cycle and, despite their low cost, have relatively large on-chip memories.

Some mediaprocessors use the same memory hierarchy that can be found on general purpose processors, although typically they can afford only a single cache level. Other mediaprocessors replace the caches with addressable on-chip memories. The most important distinguishing factor, however, is the addition of a direct memory access (DMA) engine that can be used for scheduling data transfers in parallel with computation. A double buffered transfer pattern is a common form to program the DMA engines [32]. Here, the next block of the data input is transfered to one buffer in the on-chip memory, while the current block, stored in a second buffer, can be used for the computation. When processing of the current block is completed, the buffers are switched. This technique for overlapping memory transfers and computation is very effective for hiding memory latency and efficiently using the on-chip memory area. Double buffering can also be used by mediaprocessors that have multi-way associative on-chip caches. A special hardware feature allows replacements to be disabled for a section of a cache set. The programmer maps the buffers to these areas of the cache and programs the DMA engine to transfer data directly into the cache. The remaining cache area continues to be available for caching of local variables and other data.

So far, we have given an argument for why cache prefetching is unnecessary in mediaprocessors. The use of DMA engines already is very effective in hiding memory latency. The problem of DMA engines is programmability. Programming DMA engines is quite challenging, because the programmer must write programs for both the computation and the data-flow and properly synchronize them. Moreover, such applications are not at all portable to other mediaprocessors and neither are trained programmers for one mediaprocessors easily moved to a project using a different mediaprocessor. This makes it difficult for companies to develop products using mediaprocessors. Therefore, instead of performance improvements, cache prefetching for mediaprocessors promises to reduce programming complexity by making it unnecessary to write a data-flow program.

Cache prefetching on multimedia extended general purpose processors and mediaprocessors has the primary challenge of properly detecting and effectively prefetching the memory reference patterns exhibited by multimedia applications. This requirement is no different than what cache prefetching is expected to do on general purpose processors. A secondary focus is the effective utilization of the available on-chip memory area that is more challenging due to the low temporal locality in some multimedia applications.

4.3. Data Prefetching

The first observation is that recursive prefetchers do not seem well suited for a typical multimedia memory reference pattern. An important memory reference pattern in multimedia processing is that generated by lookup tables. Lookup tables are used, for example, in generalized warping where the position of each pixel in an image is changed based on the result of a table lookup of the pixel's coordinates. Lookup tables are also popular for computing transcendental functions. While table lookups do have memory reference pairs that fall into the producer and consumer category of the recursive prefetchers, there are no chains of producers and consumers. Also the number of producers and consumers is larger and their separation short. Overall, this makes prefetching hard to apply. While it may of course be possible that a linked list or tree data structure exists in some form in a multimedia application, we are not aware of any application group where such accesses could be categorized as a dominant multimedia reference pattern.

The large off-chip memory requirements of a Markov prefetcher make it probably infeasible for today's mediaprocessors where cost is a critical factor. However, memory size may be less of an issue for mediaprocessors of the future. We are not aware of any studies that have evaluated a Markov prefetcher on typical multimedia applications. When table lookups are involved the sequence of misses should be quite random and may not be effectively prefetched by a Markov prefetcher. On the other hand, data-flow in multimedia applications is otherwise typically quite static which may also translate into a repeated sequence of cache misses over the course of the program execution. In such cases Markov prefetching could be quite effective.

The only prefetchers that have been evaluated in the context of multimedia applications are the consecutive and stride prefetchers. Zucker et al. evaluate the effectiveness of several prefetching techniques on MPEG encoding and decoding applications running on a general purpose processor [33]. They found that a stream buffer could remove on the order of 50% of the cache misses. A stride prefetcher was able to remove around 80% of the misses for cache sizes larger than 16 kbyte, however, with smaller cache sizes the stride prefetcher was no better than the consecutive prefetcher. The problem they observed is that useless prefetches were removing too much useful data from the cache. When they modified the stride prefetcher to use a prefetch buffer, this problem went away. Zucker et al. proposed to use a prefetch buffer to directly supply data to the functional units, without also copying it to the L1 cache. This was supposed to overcome the cache polluting effect that can be observed in multimedia applications. The number of cache misses removed did not change much compared to the regular stride prefetcher. Unfortunately, their results do not show comparisons in execution time and quantitative results of the prefetch accuracy to judge other possible effects of not copying prefetched data to the L1 cache.

Ranganathan et al. [30] evaluate the use of the software prefetching technique proposed by Mowry et al [6], although they insert prefetching instructions by hand and perform their own analysis for when to insert them. Here the simulation environment is a multimedia enhanced general purpose processor. They found reductions in execution time on image processing applications of 50% in many cases, eliminating most of the cache miss stall times. On the other hand, JPEG and MPEG coding applications showed only little benefit because for these applications only a small component of execution time was spent waiting on L1 cache misses. This illustrates a shortcoming of the study by Zucker et al. [33] who only model the memory latency, but not the instruction execution. In comparison, Ranganathan et al. model an out-of-order processor which shows that for these applications memory latency does not yet impact performance severely.

5. CONCLUSIONS AND FUTURE WORK

To improve performance from prefetching in a broad class of applications, a hybrid prefetching scheme must be employed. We have reviewed some prefetching techniques with a clever design that combine within one prefetching technique the ability to prefetch a wide variety of memory reference patterns. We also reviewed the technique by Joseph and Grunwald that describes in a general way how to combine two different hybrid prefetching techniques. Most promising here appears the serial prefetching method, where prefetchers are placed in series with the most accurate first. An interesting question to answer is how well a stride prefetcher and recursive prefetcher would perform together and furthermore what kind of cache misses would remain. Perhaps some other simple memory reference pattern emerges that could be exploited by yet another pure prefetcher. Important, too, are the effective hardware (or software) integration of multiple prefetch methods of which Mehrotra and Harrison's prefetcher is a nice example (although their performance impact is unproven).

Apart from the hybrid prefetchers, the performance of the pure prefetching techniques remains important because their performance directly affects the performance of the hybrid prefetchers. The stride prefetchers are most mature with accurate schemes to detect stride-n reference patters and sophisticated techniques to accurately time prefetches.

Linked prefetchers are fairly accurate, too, although there is a lack of good prefetch timing, making the prefetch buffer a popular choice. In particular, the inability to overlap prefetches may limit how far cache misses can be hidden on superscalar processors. We illustrated that there is a potential for the prefetcher by Roth et al. to be more effective with tree data structures than their current design permits.

Additional prefetch techniques may still be necessary to cover memory reference patterns that do not fall into the stride-n and linked category. Here, the Markov prefetcher may fill in, although its hardware cost makes it an expensive choice and its slow reaction time and difficulty with dynamic memory access patterns may limit it. Also important to evaluate at this point is if the point of diminishing return has not already been reached. If 50% of execution time is due to cache miss stalls and 90% of this penalty has been hidden with stride and linked prefetchers, then additional improvements in execution time are very hard to achieve.

Finally, prefetching on mediaprocessors competes with the current practice of writing data-flow programs for DMA engines and may be an important feature for reducing programmer complexity. Either method can improve performance significantly, however, prefetching is currently not explored well on mediaprocessors. The most promising prefetching technique for multimedia applications is the stride prefetcher. More work is needed in evaluating how well these work on multimedia applications, though. Little temporal locality in multimedia applications may make it beneficial to develop prefetching techniques that can isolate prefetched multimedia data into a separate on-chip memory region to prevent cache pollution.

REFERENCES

1. D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, San Francisco, CA, 1996.
2. D. Joseph and D. Grunwald, "Prefetching using Markov predictors," *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 252–263, June 1997.
3. D. Callahan, K. Kennedy, and A. Porterfield, "Software prefetching," *Proceedings of the 4th Annual International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 40–52, April 1991.
4. T.-F. Chen and J.-L. Baer, "Effective hardware-based data prefetching for high-performance processors," *IEEE Transactions on Computers* **44**(5), pp. 609–623, 1995.

5. C.-K. Luk and T. C. Mowry, “Compiler-based prefetching for recursive data structures,” *Proceedings of the 7th Annual International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 222–233, October 1996.
6. T. C. Mowry, M. S. Lam, and A. Gupta, “Design and evaluation of a compiler algorithm for prefetching,” *Proceedings of the 5th Annual International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 62–73, October 1992.
7. S. S. Pinter and A. Yoaz, “Tango: a hardware-based data prefetching technique for superscalar processors,” *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pp. 214–225, December 1996.
8. D. F. Zucker, R. B. Lee, and M. J. Flynn, “An automated method for software controlled cache prefetching,” *Proceedings of the 31st Hawaii International Conference on System Sciences*, pp. 106–114, January 1998.
9. D. Joseph and D. Grunwald, “Prefetching using Markov predictors,” *IEEE Transactions on Computers* **48**(2), pp. 121–133, 1999.
10. N. P. Jouppi, “Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers,” *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 364–373, May 1990.
11. S. Palacharla and R. E. Kessler, “Evaluating stream buffers as a secondary cache replacement,” *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pp. 24–33, April 1994.
12. A. Roth, A. Moshovos, and G. S. Sohi, “Dependence based prefetching for linked data structures,” *Proceedings of the 8th Annual International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 115–126, October 1998.
13. J.-L. Baer and T.-F. Chen, “An effective on-chip preloading scheme to reduce data access penalty,” *Proceedings of the 1991 Conference on Supercomputing*, pp. 176–186, November 1991.
14. J. W. C. Fu, J. H. Patel, and B. L. Janssens, “Stride directed prefetching in scalar processors,” *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pp. 102–110, December 1992.
15. S. Mehrotra, *Data Prefetch Mechanisms for Accelerating Symbolic and Numeric Computation.* PhD thesis, University of Illinois, Urbana, IL, 1996.
16. M. H. Lipasti, W. J. Schmidt, S. R. Kunkel, and R. R. Roediger, “SPAID: Software prefetching in pointer- and call-intensive environments,” *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pp. 231–236, November 1995.
17. J. E. Smith, “Decoupled access/execute computer architectures,” *Proceedings of the 9th Annual International Symposium on Computer Architecture*, pp. 112–119, July 1982.
18. J. D. Gindele, “Buffer block prefetching method,” *IBM Technical Disclosure Bulletin* **20**(2), pp. 696–697, 1977.
19. A. M. G. Maynard, C. M. Donnelly, and B. R. Olszewski, “Contrasting characteristics and cache performance of technical and multi-user commerical workloads,” *Proceedings of the 6th Annual International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 145–156, October 1994.

20. C.-K. Luk and T. C. Mowry, “Cooperative prefetching: compiler and hardware support for effective instruction prefetching in modern processors,” *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pp. 182–194, November 1998.

21. T. C. Mowry and C.-K. Luk, “Compiler and hardware support for automatic instruction prefetching: A cooperative approach,” Tech. Rep. CMU-CS-98-140, School of Computer Science, Carnegie Mellon University, June 1998.

22. S. Mehrotra and L. Harrison, “Examination of a memory access classification scheme for pointer-intensive and numeric programs,” *Proceedings of the 1996 International Conference on Supercomputing*, pp. 133–140, May 1996.

23. R. B. Lee, “Efficiency of microSIMD architectures and index-mapped data for media processors,” *SPIE Proceedings*, vol. 3655, pp. 34–46, 1999.

24. S. Oberman, F. Weber, N. Juffa, and G. Favor, “AMD 3DNow! technology and the K6-2 microprocessors,” *Proceeding Notebook for Hot Chips 10*, pp. 245–254, August 1998.

25. M. Phillip, “A second generation SIMD microprocessor architecture,” *Proceeding Notebook for Hot Chips 10*, pp. 111–121, August 1998.

26. R. B. Lee, “Subword parallelism with MAX-2,” *IEEE Micro* **16**(4), pp. 51–59, 1996.

27. A. Peleg and U. Weiser, “MMX technology extension to the intel architecture,” *IEEE Micro* **16**(4), pp. 42–50, 1996.

28. M. Tremblay, J. M. O’Connor, V. Narayanan, and L. He, “VIS speeds new media processing,” *IEEE Micro* **16**(4), pp. 10–20, 1996.

29. R. Bhargava, L. K. John, B. L. Evans, and R. Radhakrishnan, “Evaluating mmx technology using dsp and multimedia applications,” *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pp. 37–46, November 1998.

30. P. Ranganathan, S. Adve, and N. P. Jouppi, “Performance of image and video processing with general-purpose processors and media ISA extensions,” *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pp. 124–135, May 1999.

31. S. G. Berg, W. Sun, D. Kim, and Y. Kim, “Critical review of programmable mediaprocessor architectures,” *SPIE Proceedings*, vol. 3655, pp. 147–156, 1999.

32. I. Stotland, D. Kim, and Y. Kim, “Image computing library for a next-generation VLIW multimedia processor,” *SPIE Proceedings*, vol. 3655, pp. 47–58, 1999.

33. D. F. Zucker, M. J. Flynn, and R. B. Lee, “A comparison of hardware prefetching techniques for multimedia benchmarks,” *Proceedings of the International Conference on Multimedia Computing and Systems*, pp. 236–244, June 1996.