Stride-directed Prefetching for Secondary Caches *

Sunil Kim
IBM (MS 4305)
11400 Burnet Road
Austin, TX 78758
skim@austin.ibm.com

Alexander V. Veidenbaum
Department of EECS
University of Illinois at Chicago
Chicago, IL 60607-7053
alexv@eecs.uic.edu

Abstract

This paper studies hardware prefetching for second-level (L2) caches. Previous work on prefetching has been extensive but largely directed at primary caches. In some cases only L2 prefetching is possible or is more appropriate. By studying L2 prefetching characteristics we show that existing stride-directed methods [1, 8] for L1 caches do not work as well in L2 caches. We propose a new stride-detection mechanism for L2 prefetching and combine it with stream buffers used in [16]. Our evaluation shows that this new prefetching scheme is more effective than stream buffer prefetching particularly for applications with long-stride accesses. Finally, we evaluate an L2 cache prefetching organization which combines a small L2 cache with our stride-directed prefetching scheme. Our results show that this system performs significantly better than stream buffer prefetching or a larger non-prefetching L2 cache without suffering from a significant increase in the memory traffic.

1 Introduction

Advances in processor architecture and its implementation technology make memory latency an ever increasing problem for high-performance systems. A two-level cache hierarchy is typically used to deal with the problem and the approach is quite successful for many types of applications. However, programs with large data sets or low data reuse, such as scientific applications, often have high miss rates. Combined with longer access times for the secondary caches and main memory this results in poor cache performance.

Several solutions have been advanced as a way to tolerate these long latencies. A solution we will concentrate on in this paper is data prefetching. Prefetching attempts to predict future data accesses and move the data to the upper levels of the memory hierarchy before it is referenced. This either eliminates or reduces the memory latency.

Both hardware and software prefetching approaches have been studied extensively. Software prefetching, which has been investigated in [3, 9, 13, 15], uses a compiler to analyze memory references and insert prefetch instructions accordingly. The prefetch instructions fetch data into caches or prefetch buffers for future memory accesses.

In hardware prefetching, special-purpose hardware monitors data access patterns and prefetches data according to the observed memory access patterns. The simplest hardware prefetching schemes are the *one block lookahead* (OBL) schemes [18], where only cache line $l + 1$ is considered for prefetching when line l is accessed. These schemes differ in ways a prefetch of a line is triggered, namely *always prefetching, prefetching on misses*, and *tagged prefetching*. Another prefetching method is stream buffers, proposed in [12]. In this method a stream buffer is assigned on a cache miss and the stream buffer prefetches successive cache lines starting at the miss address. Both of these methods cannot handle strides longer than a cache line.

More sophisticated hardware prefetching schemes use *stride-directed* prefetching and are described in [1, 8, 11, 16]. These can handle arbitrary constant strides and are discussed in more detail later.

Finally, it is possible to attack the problem through a combination of hardware and software. A general prefetch engine using software instructions to supply the stride and start prefetching for a stream is proposed in [4]. A different integrated hardware/software prefetching approach is proposed in [10].

We will focus on hardware-based prefetching in this paper. We further limit our study to secondary cache prefetching only while assuming no prefetching in the primary cache. There are several reasons for performing prefetching only at second-level caches (L2 prefetching). First, system designers have an opportunity to add prefetching hardware to an off-chip secondary cache which may not be possible in an on-chip primary cache. Prefetching logic added to primary cache increases complexity and may increase processor cycle time. Finally, a larger L2 cache can tolerate more cache pollution caused by prefetching than a small primary cache can.

Simple hardware prefetching such as tagged prefetching and stream buffers can be used for L2 prefetching. These methods are attractive for L2 prefetching because of their simplicity and/or when only L1 cache miss addresses are available. However, these methods are not effective for memory accesses with strides longer than a cache line which occur in most scientific applications. Stride-directed prefetching is much more appropriate in such programs.

In stride-directed prefetching, a *memory reference stream*, i.e. a sequence of data addresses generated by a given memory access instruction, is detected by hardware and its stride calculated. The calculated stride is used to predict and prefetch future memory accesses.

In most previous studies [1, 8, 11], a memory reference stream is detected by using instruction addresses. This approach, denoted an *instruction-address-based* prefetching, is quite successful in L1 prefetching. However, it has not been shown whether instruction-address-based or stride-directed prefetching is effective at L2 cache level. There

*This work was supported in part by the National Science Foundation under Grant No. NSF 89-20891.

are several possible reasons why they may not be. First, the stride detector at L2 does not see all data addresses, just L1 cache misses. This makes it difficult to detect a constant stride due to locality or due to cache conflicts in L1 cache. The former makes a hit address invisible to the stride detector, the latter causes *extra* misses to be seen by the stride detector. We refer to this as an L1 cache *screening* effect. Second, instruction addresses may be difficult or costly to obtain outside of a processor chip for system-level L2 cache implementation. Thus it is desirable to have an L2 prefetching method capable of detecting memory reference streams without instruction addresses.

L2 prefetching proposed in [16] combines stream buffers with non-unit stride detection. This is the only method specifically aimed at L2 prefetching and it does not use instruction addresses. Two heuristic approaches to non-unit stride detection are proposed. In one a program address space is partitioned into separate *concentration zones*. References falling into the same zone are considered to be in the same stream and are used to calculate the stride. Prefetching performance is shown sensitive to the zone size. The choice of the right zone size is delegated to either a compiler or a programmer. A second heuristic is a *minimal delta* scheme which detects a stream by finding two memory references with the smallest address distance among last few memory references. It was found to work as well in [7] but needed more complex hardware. However, these methods are not as accurate as L1 instruction-address-based method in detecting memory reference streams and can stand improvement.

In this paper, we propose a new stride detection method for L2 caches, called *loop-based prefetching*. This scheme does not need instruction addresses to identify a memory reference stream and requires only modest hardware. We compare loop-based prefetching with stream buffers and instruction-address-based prefetching. We also analyze L2 prefetching characteristics which profoundly affect the relative performance of different prefetching methods.

L2 prefetching has been proposed with and without caches. Only stream buffers and other prefetching hardware are used in [16], whereas a small L2 cache is added to prefetching hardware in [14]. However these two different architectures have not been directly compared. In this study we add a small cache to prefetching hardware and compare it with a traditional cache (cache-only), stream buffer prefetching without a cache (prefetching-only), and prefetching with a small cache.

The rest of this paper is organized as follows. The next section describes loop-based prefetching. Section 3 presents our evaluation methodology and prefetching architectures. Section 4 discusses our results. Section 5 concludes this paper.

2 Loop-Based Prefetching and Its Implementation

Our goal is an L2 prefetching which can detect memory references streams without using instruction addresses. We will use it in conjunction with stride calculation units to perform stride detection and prediction. We call our proposed scheme *loop-based prefetching*. The method incurs only a modest hardware increase compared to other proposed methods.

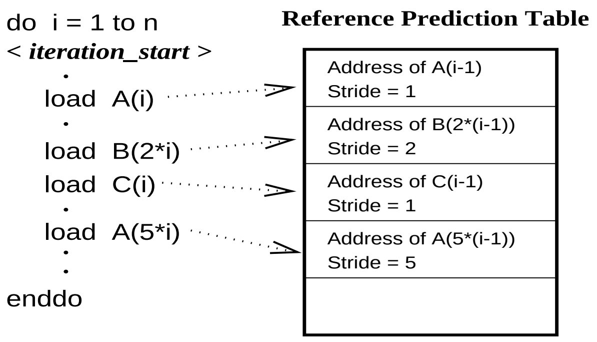


Figure 1: Memory reference stream detection in Loop-Based Prefetching

2.1 The approach

Memory references with a constant stride are usually generated by load/store instructions which are repeatedly executed in a loop. During the execution of a loop body, the execution order of load/store instructions inside the loop body remains constant from one iteration to the next (assuming in-order execution with respect to other loads/stores and absence of branches). If a start of an iteration can be detected, it is possible to identify a memory reference stream by its appearance relative to the start of an iteration as shown in Figure 1.

An iteration of a loop provides the prefetching unit with an *iteration start* signal. Subsequent loads are identified by their relative position with respect to the iteration start signal. The relative position of a load within an iteration is used by the prefetching unit as an index into the *Reference Prediction Table(RPT)*. An entry at the indexed position contains a stride and prefetch address for the load instruction. The prefetching unit resets the index to zero on each iteration start signal. Thus a simple counter is sufficient to identify a stream and access an RPT entry. The stride is used to prefetch ahead of the L1 requests once it is computed.

An overflow occurs if there are more loads than the total number of entries in the RPT. Such a case is easily detected as the counter overflows, and subsequent loads until the next iteration start signal are ignored and are not prefetched. Other RPT operations, such as updating memory access history and computing strides, can be implemented in ways similar to previous proposals [1, 8, 11]. The iteration start signal can be generated by a memory-mapped instruction accessing a particular memory address, a new instruction, or by an instruction unit detecting a backward branch. In either case, only one such instruction is needed per iteration in contrast to some schemes requiring an instruction per memory reference.

In L2 loop-based prefetching load accesses that hit in the L1 cache are not seen by prefetching hardware. Therefore, the L2 prefetching unit uses only L1 misses to increment the index to RPT and keeps trying to acquire a constant stride from L1 misses mapped to the same RPT entry. Thus loop-based prefetching may suffer from the L1 screening effect just as other stride-directed prefetching methods do. But as we will show, this simple approach is quite effective for stride detection.

The method can be extended to handle multiply-nested

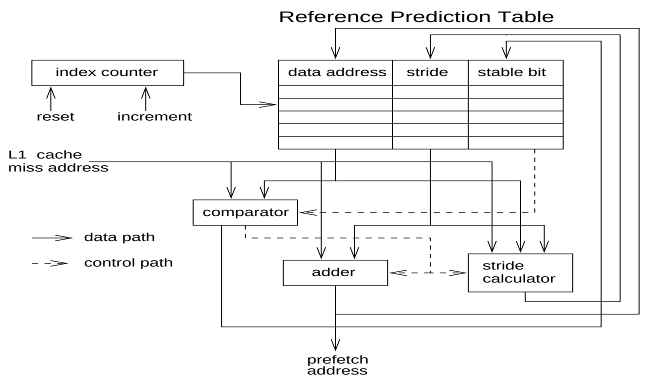


Figure 2: Loop-Based Stride Detection Unit

loops and branches inside loops. By providing additional loop execution information, such as *loop start*, *loop exit*, and *branch taken* it is possible to accurately identify any memory reference stream generated in a loop. However, the prefetching unit becomes more complex. In this study we use a simple RPT hardware mechanism ignoring branches and concentrating on *innermost* loops only because they account for most of the memory accesses in a program (see statistics in Table 1).

2.2 Implementation

The organization of a stride detection unit for loop-based prefetching is shown in Figure 2. We refer to it as *loop-based stride detection unit.* It consists of a Reference Prediction Table (RPT), an RPT index counter, a comparator, an address adder, and a stride calculator. Each RPT entry contains a data address (DA_l) and a stride (St_l) for a reference stream.

An RPT entry pointed by the RPT index counter is accessed with an L1 miss address (M_a), and the stride is computed. As soon as a *constant stride* is detected, that is, a newly calculated stride is equal to the old stride, the information (data address and stride) is sent to a prefetching engine such as stream buffers. The stream buffer will generate a prefetching request for the stream. The following algorithm describes the operation of the prefetching unit.

```
if $M_a == DA_l$ and Stable then
    /* constant stride detected */
    allocate a stream buffer with
        $DA \leftarrow DA_l + St_l$; $St \leftarrow St_l$.
    issue a prefetch for $DA$.
else
    /* set stable bit and calculate a new stride */
    $Stable \leftarrow St_l == M_a - (DA_l - St_l)$.
    $St_l \leftarrow M_a - (DA_l - St_l)$.
endif
$DA_l \leftarrow M_a + St_l$.    /* update data address */
```

3 Performance Evaluation Methodology

3.1 Simulation

We used trace-driven simulation for performance evaluation. Benchmark programs were executed and memory addresses referenced by loads were obtained via program instrumentation. Programs were instrumented by a tracing tool [5] we developed for MIPS R3000 processors, compiled with a MIPS optimizing Fortran compiler, and executed on a MIPS M120 system with a R3000/3010 processor. The address traces drive prefetching architecture simulators developed for this study.

Instructions fetches are ignored by assuming an instruction cache with a perfect hit rate. The first and second-level data caches are direct-mapped, write-through, and no-write-allocate. A cache line size is 4 words, and a word is 8 bytes in each case (32-byte line). We use an 8 Kbyte L1 data cache throughout our experiments while L2 cache size and organization vary.

Only loads are prefetched. An L2 cache, if present, is searched before a prefetching unit issues a prefetch request to memory. The prefetch request is discarded on hit. We do not employ timing simulation and therefore assume that prefetched data is available to subsequent memory references without any delay.

The *iteration start* signal for a loop is obtained by inserting a special command inside the loop. From this command the MIPS tracing tool generates a special instruction that is recognized as an iteration start by the simulator.

3.2 Benchmarks

Table 1 shows the characteristics of the benchmarks used. All are computationally intensive numerical applications. These applications usually exhibit large data set sizes. Except for ARC3D, they are selected from among the PERFECT [6] and NAS benchmarks [2]: DYFESM, FLO52, MDG, OCEAN and TRFD are PERFECT benchmarks, and APPBT, APPSP, CG, and MGRID are NAS benchmarks.

Benchmark statistics were collected dynamically during program execution. They include percentage of innermost loads, e.g. loads generated inside innermost loops, and of loads with constant stride. Percentage of constant *long* stride loads, e.g. strides longer than a cache line size (positive or negative), in the total number of constant stride loads is also shown. Notice a large number of long stride accesses for some benchmarks.

For programs other than MDG, the innermost loads account for most of the load accesses in a program. MDG shows a noticeably lower innermost load percentage, 65.4 %, as compared to the other benchmarks. Note that CG and DYFESM have relatively low constant stride load access ratios, and APPBT, APPSP, ARC3D, OCEAN and TRFD have higher long stride ratios. CG and DYFESM showed such low constant stride ratios because they have a large number of array indirections.

3.3 Performance Metrics

A cache miss (hit) rate is our main performance metric. In addition, prefetch prediction miss rate (PPMR) is used to evaluate how well future data accesses are predicted by prefetching units. This metric is defined as

$$\mathrm{PPMR} = \left(1 - \frac{\begin{array}{c}\text{The number of memory accesses}\\ \text{that hit in a prefetching unit}\end{array}}{\begin{array}{c}\text{The number of prefetch}\\ \text{requests generated}\end{array}}\right) \times 100$$

Benchmark	Description	Load Access (M)	Load Hit Rate (L1) (%)	Innermost Load (%)	Const. Stride Load (%)	Const. Long Stride Load (%)
APPBT	BT simulated CFD	17.47	93.2	86.6	80.51	57.88
APPSP	SP simulated CFD	16.49	86.1	95.7	86.41	44.6
ARC3D	3-D CFD	20.33	86.8	86.1	83.37	31.87
CG	Conjugate gradient	14.4	67.5	95.5	61.55	0.05
DYFESM	Structural dynamics	18.08	87.0	89.3	71.92	5.97
FLO52	2-D fluid dynamics	15.03	82.5	99.5	92.64	5.75
MDG	Molecular dynamics	20.70	96.9	65.4	82.47	2.64
MGRID	Multigrid kernel	14.53	94.8	99.9	92.49	0.40
OCEAN	2-D fluid dynamics	13.21	76.3	99.3	96.93	31.9
TRFD	Molecular dynamics	15.39	92.0	97.6	87.16	49.49

Table 1: Benchmark Characteristics

Finally, we measure memory traffic, that is, the total number of memory requests generated by cache misses and prefetch requests. Prefetching may increase the memory traffic and lead to a higher overall memory latency.

We do not measure execution time because it requires detailed timing information about the memory subsystem and the processor. The above metrics, on the other hand, are timing independent. The absence of timing evaluation may produce optimistic results for prefetching when compared with caches. It demonstrates the upper bound of performance improvement with prefetching. However, PPMR and total memory traffic are sufficient performance metrics for uniform comparison of prefetching architectures.

3.4 Prefetching Architecture

A general system organization for L2 prefetching used in this study includes a CPU, an L1 and an (optional) L2 caches, and an L2 prefetching unit. An L2 cache and/or an L2 prefetching unit are directly connected to an L1 cache. We investigate three prefetching unit architectures:

1. *Stream buffers* and a *miss table unit* (similar to the unit-stride filter [16]) is our baseline prefetching architecture called *miss table prefetching (MTP)* architecture.

2. *Loop-based prefetching architecture (LBP)* uses our new stride detection unit to detect memory reference streams and compute their strides based on the knowledge of program loop iterations. The unit is added to the baseline architecture.

3. *Instruction-address-based architecture (IAP)* uses an instruction address (PC) to identify memory access streams and compute their strides for prefetching.

The baseline architecture is chosen because it is simple, does not use memory reference stream and stride detection, and has been shown effective in L2 prefetching [16]. The loop-based stride detection unit replaces the heuristic non-unit stride filter used in [16] and is mainly used for long stride detection. Note that we do not use the loop-based stride detection unit alone, which will be explained later. Figure 3 shows the baseline prefetching architecture (inner dotted box) and the loop-based prefetching architecture(outer dotted box).

The stream buffers are based on the model used in [16]. The stream buffers are fully-associative, and LRU allocation is used. A stream buffer is allocated with *data address* and *stride* supplied by the miss table unit or the loop-based stride detection unit, and a line containing the data address is fetched into the stream buffer. When an L1 miss hits on a stream buffer, the stream buffer fetches the next line that contains a word at the data address plus stride and increases its data address by the stride.

When no stream buffers are found to contain the L1 miss address, the L1 miss address is sent to the miss table unit. The miss table is fully-associative and contains a *next line* address for each previously seen L1 miss. The miss table sees only L1 misses that miss the stream buffers. If the L1 miss address hits in the miss table, a stream buffer is allocated with a data address and a stride equal to the line size. The data address is the starting address of the next line to the L1 miss. If the L1 miss address misses in the table, an entry is allocated for the miss in FIFO order. The miss table detects short stride memory accesses.

The loop-based stride detection unit is added to MTP to detect long stride memory accesses. The operation of the detection unit is described in Section 2.2. Note that the detection unit only sees L1 misses that miss in both stream buffers and miss table unit. Stream buffer and miss table hits filter out unit-stride references which otherwise may *confuse* the stride detector.

The instruction-address-based architecture is based on the model proposed by Baer and Chen [1]. L1 misses are directly sent to the reference prediction table (RPT) with their instruction address. The RPT is fully-associative and indexed by the instruction address. If the instruction address misses in the RPT, an LRU entry is allocated with the L1 miss address and the instruction address. If it is a hit, the RPT stores the miss address and compares it with a previously stored miss address to compute the stride. If a *constant* stride is obtained, the unit will prefetch the line containing the miss address increased by the stride.

We use two different L2 memory organizations: one with an L2 cache and one without an L2 cache. In the latter case only stream buffer storage is used for data. In the former case data is stored in the L2 caches itself, and the

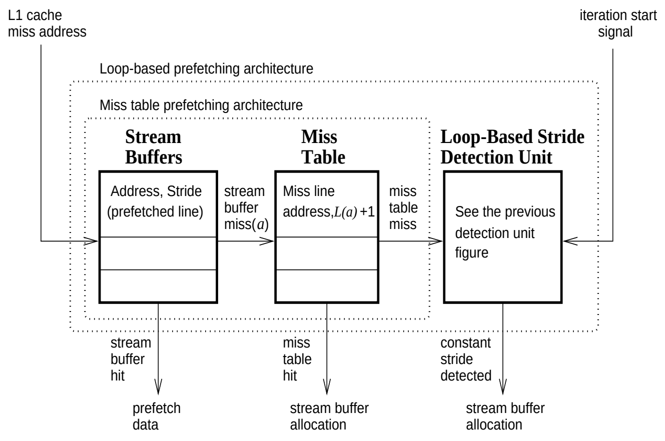


Figure 3: L2 Prefetching Architectures

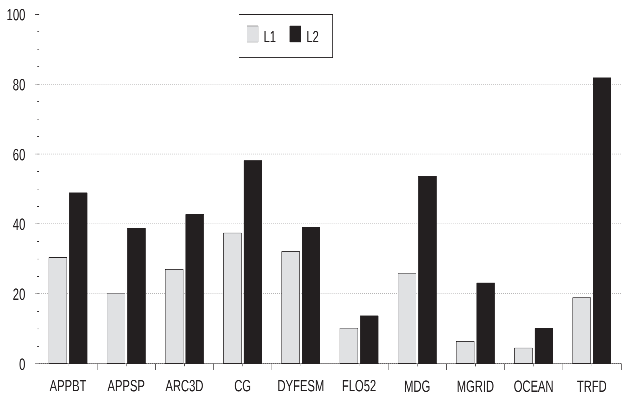


Figure 4: IAP L1 and L2 Prefetching Prediction Miss Rate (PPMR), RPT size = 256

stream buffers only generate and test address information for prefetching. For instruction-address-based prefetching architecture data is stored in separate additional storage associated with each Reference Prediction Table (RPT) entry. RPT is thus similar to stream buffers in this sense. The difference is that data is accessed with instruction addresses in RPT.

4 Experimental Results and Analysis

Let us start by examining the effectiveness of L2 stride-directed prefetching. First, we compare the IAP prediction miss rates for L1 and L2 caches. Figure 4 shows the prefetch prediction miss rate (PPMR) for the two cases using a large RPT (256 entries). Although we use the same prefetching method by supplying instruction addresses to the L2 IAP unit, for most benchmarks the prefetch prediction miss rate of the L2 IAP is much higher than that of the L1 IAP method. This high PPMR is caused by the difficulty in detecting a constant stride at L2 cache. The screening effect of L1 caches and conflict misses perturb miss address streams which otherwise would be consistently visible to L2 prefetching units. The results indicate that stride-directed prefetching may not be as effective at L2 as at L1, even if memory reference streams are accurately detected.

4.1 Stream Buffer Prefetching

The effect of the number of stream buffers on performance for all three prefetching methods (IAP, LBP, and MTP) is evaluated next. Data is stored in stream buffers for the MTP and LBP and in the RPT data space in each table entry for the IAP. Thus the RPT and stream buffers are similar in their overall function and organization and we use the term *stream buffer* for both, unless stated otherwise. There is no conventional L2 cache in this case.

The results for 16, 64, and 256 stream buffers (RPT size) are presented. Even the smallest size used here is larger than what has been previously studied since we believe that extra locality can be exploited with more buffers. The miss table size was determined in other experiments and set to 32 entries. Increasing the size beyond 32 did not improve the prediction miss rate significantly but hurt the performance in some cases. Stream buffers and miss tables need to be fully associative and it may not be practical to increase their size beyond 64. The larger size is used to confirm our locality conjecture.

In addition to the stream buffers and the miss table, the LBP method needs an RPT for loop-based stream detection. In the experiments, the size of this additional RPT is fixed at 64 entries. We observed that no more than 33 entries were used in any of our benchmarks. Note that the RPT for loop-based prefetching is used only for stream detection and does not issue prefetch requests as done by the stream buffers or the RPT in the IAP method.

Figures 5 shows stream buffer miss rates for each benchmark. The stream buffer miss rate is the number of stream buffer misses over the total number of stream buffer accesses. The latter is the same as L1 cache misses. The stream buffer miss rate steadily decreases, in most cases, as the number of stream buffers is increased. Increasing the number of stream buffers is beneficial because it provides more storage for prefetched data and thus the opportunity for reuse. In addition, it allows more opportunities to initiate prefetches.

LBP shows equal or lower stream buffer miss rates than MTP for all benchmarks and parameters. LBP improves noticeably over MTP by detecting long stride accesses for APPSP, ARC3D, and OCEAN and by detecting streams more accurately for MDG. These benchmarks except for MDG have a relatively large portion of long stride accesses as shown in Table 1. This result shows that the loop-based stride detection unit is effective at L2 prefetching for long stride accesses and in some cases for short stride accesses.

Two other benchmarks, APPBT and TRFD, with many long stride accesses do not have much improvement with LBP. More than 60 % of long-stride misses in APPBT have strides less than 2 cache line sizes. The miss table can also successfully capture such accesses and allocate stream buffers not leaving much room for improvement from LBP. TRFD has a large percentage of true long stride accesses, but shows only small improvements with LBP or IAP because they fail to detect a constant stride. This is due, in part, to conflict misses in L1. We confirmed in additional experiments that TRFD performance improves noticeably with a 4KB, 2-way set-associative L1 cache.

Previous results showed that stride-direct prefetching

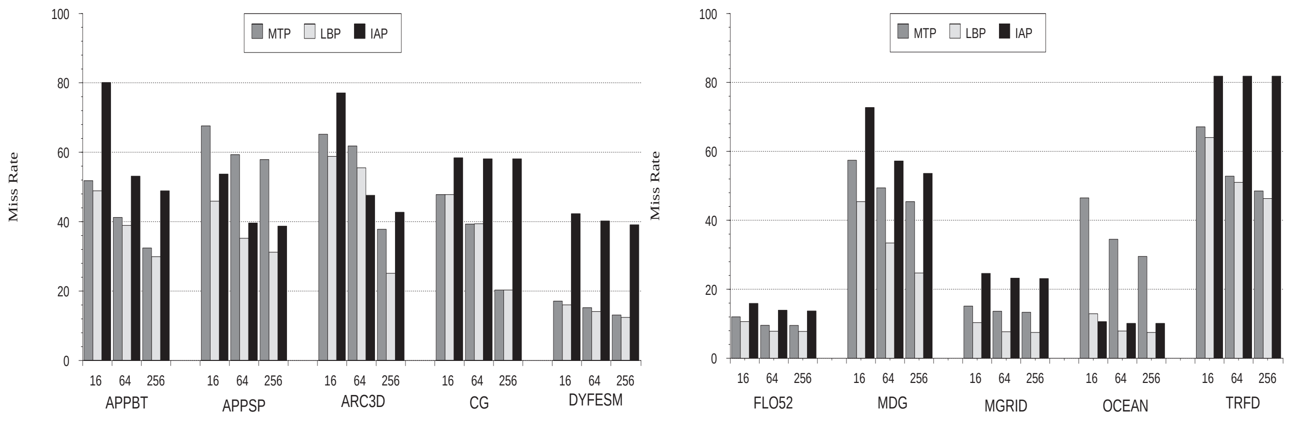


Figure 5: Stream Buffers Miss Rates. The X-axis numbers represent the number of stream buffers.

can be effective at L2. However, IAP while theoretically capable of detecting memory streams more accurately than the loop-based stride detection unit performs worse than MTP and LBP in most cases. One of the reasons for loss of performance in IAP is that data in the stream buffers is looked up by a data address in MTP and LBP, whereas it is looked up by an instruction address in IAP. A more important reason is a different approach to prefetching. IAP is based on memory access patterns of each memory reference stream, whereas MTP and LBP can exploit memory access patterns with good spatial locality across several memory reference streams. We quantify this more precisely in the next section.

4.2 Cross-stream locality

A stream buffer allocation by a miss table or a loop-based stream detection unit is caused by a series of L1 cache misses. The misses causing the allocation may actually belong to the same or a different memory reference stream. If they do belong to the same memory reference stream we say that a stream buffer is *exactly allocated* for the memory reference stream and *belongs to* the memory reference stream. When a miss address hits on a stream buffer, the miss address and stream buffers may belong to the same or a different memory reference stream, or the stream buffer is not even exactly allocated. Based on these observations we classify stream buffer hits as follows:

- Type 1 hit: the stream buffer is *not* exactly allocated.
- Type 2 hit: the stream buffer is exactly allocated, but the L1 cache miss and the stream buffer belong to different memory reference streams.
- Type 3 hit: the stream buffer is exactly allocated, and the L1 cache miss and the stream buffer belong to the same memory reference stream.

The type 1 and type 2 hits indicate a *cross-stream* locality, the locality among different memory reference streams. The type 3 hit implies successful prefetching based entirely on memory access patterns of a *single* memory reference stream.

The type distribution of stream buffer hits for MTP and LBP is shown in Figure 6. It shows that a large portion of

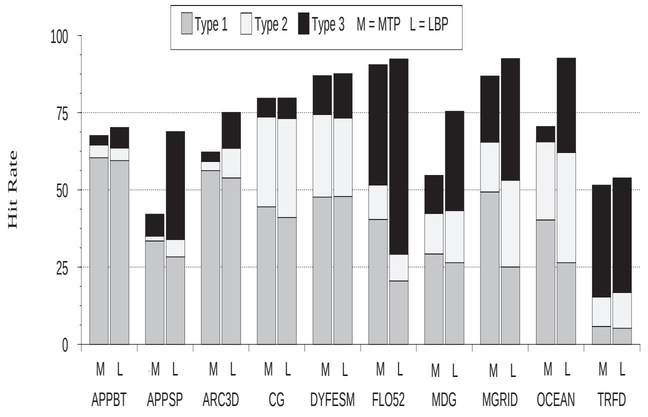


Figure 6: Distribution of Stream Buffer Hits for MTP and LBP with 256 Stream Buffers

stream buffer hits is contributed by type 1 and type 2 hits. A similar behavior is observed for MTP except that type 3 hit contribution is not as large as for LBP. Under IAP only type 3 hits are possible since the RPT is indexed by an instruction address both when memory access patterns are detected and data is accessed. Since IAP can have only type 3 hits, IAP showed worse performance than MTP and LBP. Furthermore, the results in the next section show (see Figure 7) that when IAP is used to prefetch into an L2 cache and type 2 hits are exploited the performance for most benchmarks is still worse than that of MTP and LBP. This implies that an ability to exploit type 1 hits is very important to the success of L2 prefetching.

Stride detection is more difficult at L2 and not as effective as at L1 because L1 misses which arrive at L2 may not have a constant stride. Therefore, prefetching methods which can take advantage of good cross-stream spatial locality should be used for L2 prefetching.

4.3 Prefetching with Caches

Previous experiments showed that a large number of stream buffers improved the performance by exploiting memory reference locality. In this section, we add a small L2 cache to the prefetching architectures to take advantage of the

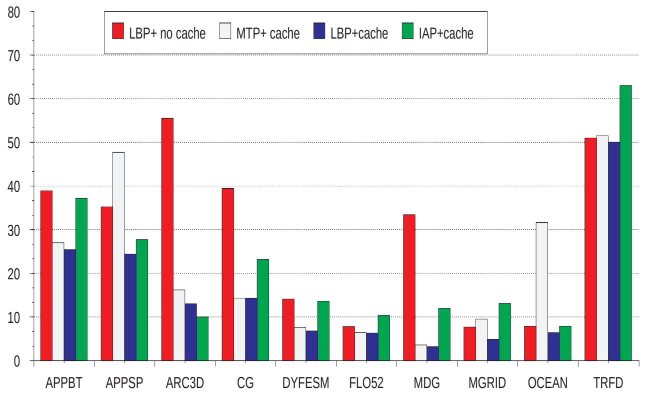


Figure 7: Miss rates for Prefetching with and without 32KB L2 caches: 64 Stream Buffers for LBP, MTP and IAP

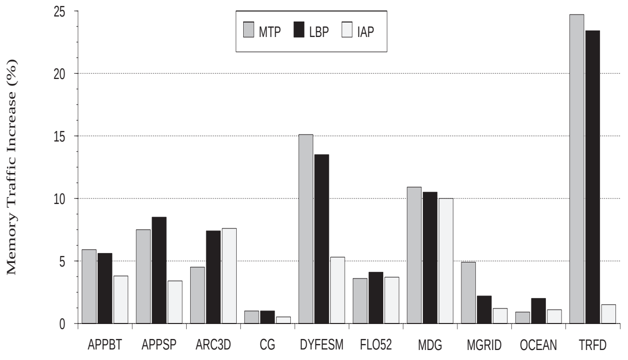


Figure 9: Prefetching Memory Traffic Increase over Cache-only System: MTP, LBP, and IAP use a 32KB L2 Cache and 64 Stream Buffers. The Cache-only System uses a 32KB L2 cache.

locality and study the effect on performance. In this organization, the prefetched data is directly stored into the L2 cache. Stream buffers only store data address and stride information and issue prefetch requests. A 32KB, direct-mapped, write-through L2 cache is used in this study. A line contains four 8Byte words. The number of stream buffers and the RPT size of IAP is 64, and the miss table size is 32.

The L2 miss rates of LBP without a cache and the L2 cache miss rate of MTP, LBP, and IAP with a 32KB L2 cache are shown in Figure 7.

The addition of even a small L2 cache improves the performance for all benchmarks, significantly in more than half the cases. The cache is most beneficial for IAP because IAP can now have type 2 hits. However, MTP and LBP with a cache still show better performance than IAP with a cache for most benchmarks due to their ability to utilize type 1 hits. Although not shown here, the number of stream buffers is not as important in this case, even a smaller number is very effective with a small L2. Also not shown is the fact that the small L2 cache suffers from conflict misses in some benchmarks. Once the set-associativity of the small L2 cache is increased, the cache miss rate is dramatically reduced.

Next, the hit rates of non-prefetching L2 caches and prefetching with a small cache are compared while varying the non-prefetching cache size. Figure 8 shows that the cache hit rate difference between LBP with a 32KB L2 cache and conventional L2 caches of size 32K, 128K, 256K, 512K, 1M, and 2M Bytes. A negative value in the figure means the LBP system has a higher hit rate than the non-prefetching L2 cache by the amount shown. In general, as the L2 cache size increases, the hit rate of a non-prefetching L2 cache becomes higher than that of the LBP system with a small cache. For all benchmarks except TRFD, L2 caches need to be two to four times larger (64 to 128K) than the cache used with LBP to match the hit rate. The LBP system can attain up to 40 % higher hit rates in half of the benchmarks than a 16-times larger non-prefetching L2 cache (512K).

TRFD has the worst LBP performance because LBP suffers from L2 cache conflict misses. We have verified that performance of LBP improves significantly when the associativity of the L2 cache is increased.

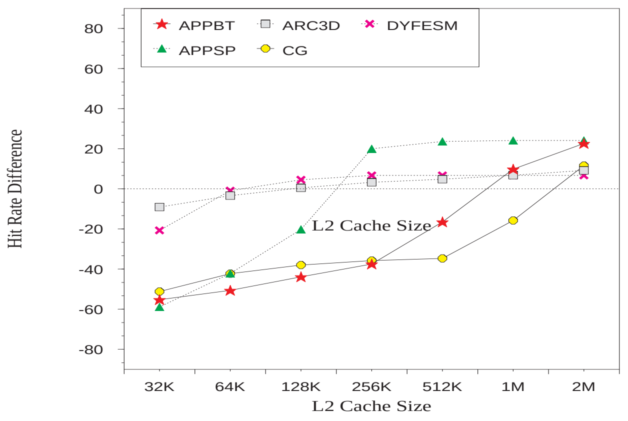
These results are consistent with [16] which has shown that stream buffers alone can perform better than an L2 cache for some scientific codes with large data sets. Our results show that adding a small L2 cache to LBP can significantly boost the performance. It also requires a smaller number of stream buffers for most benchmarks. This combined organization is a better choice than stream buffers alone and, in many cases, non-prefetching L2 cache alone.

4.4 Traffic increase under prefetching

One problem associated with prefetching is a memory traffic increase. We compare memory traffic increase of MTP, LBP, and IAP with respect to the memory traffic generated with a 32 Kbyte L2 cache. Figure 9 shows the memory traffic increase for each prefetching method and benchmark. In most cases the increase is less than 10%. Only DYFESM, MDG and TRFD show a 15 %, 11 %, and 24 % increase, respectively. The low increase is due to the fact that prefetches are not started until a stable stride is detected and only one cache line is prefetched each time. For most benchmarks, IAP shows the smallest increase because it does not prefetch much data due to low stream buffer hit rates. LBP has a lower memory traffic increase than MTP for APPBT, DYFESM, MGRID and TRFD. Lower miss rates and a relatively small memory traffic increase makes LBP with a small cache the most efficient architecture among the various L2 memory organizations we studied.

5 Conclusions

We have shown that second-level (L2) prefetching has different characteristics compared to first-level (L1) prefetching. In particular, prefetching that relies solely on constant stride detection of a memory reference stream was shown to be less effective. An L2 prefetching architecture should take advantage of cross-stream locality to be effective. Stream buffers and the miss table prefetching (MTP) are one of such L2 prefetching methods. However, these methods still leave room for improvement for long stride accesses.



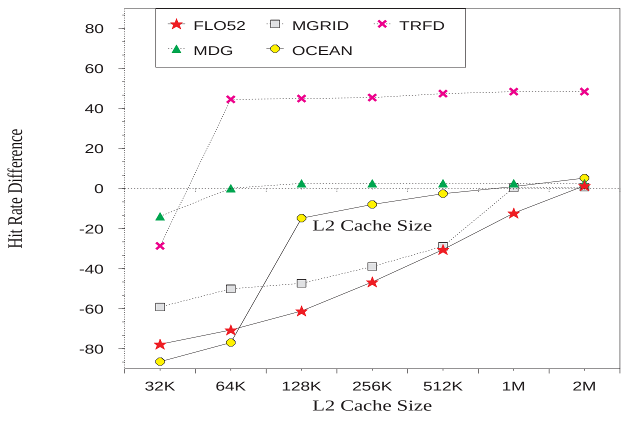


Figure 8: Hit Rate Difference Between Loop-Based Prefetching versus Secondary Caches: LBP uses 64 Stream Buffers, a 32-entry Miss Table, a Loop-Based Stream Detection Unit and a 32K L2 Cache. L2 Cache-only uses 32K, 64K, 128K, 256K, 512K, 1M, and 2M Caches.

We proposed loop-based prefetching (LBP) to perform prefetching for long stride accesses at L2 as a way to improve on existing heuristics. Loop-based prefetching relies on loop execution information and does not need instruction addresses to identify memory reference streams. When we combined loop-based stride detection with MTP prefetching, it significantly improved performance particularly for applications with long stride accesses. This combined architecture (LBP architecture) was the best method improving over both MTP and IAP architectures.

We have also shown that additional locality can be exploited by increasing the number of stream buffers when prefetched data is store in the buffers only. Finally, by using a small L2 cache for prefetched data storage we were able to improve the performance of LBP architecture even further. This organization was competitive with much larger non-prefetching caches in many benchmarks. The combined LBP architecture exhibited only a small increase in memory traffic due to prefetching compared to other prefetching architectures we investigated.

References

[1] Jean-Loup Baer and Tien-Fu Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Supercomputing*, pages 176–186. IEEE, November 1991.

[2] D. Bailey, H. Simon, J. Barton, and T.Lasinski. The NAS parallel benchmarks. Technical Report RNR-91-02, NASA Ames Research Center, 1991.

[3] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *Architectural Support for Programming Languages and Operating Systems*, pages 40–52, April 1991.

[4] Tien-Fu Chen. *An Effective Programmable Prefetch Engine for On-Chip Caches.* In *International Symposium on Microarchitecture (Micro-28)*, pages 237–242, November 1995.

[5] Yung-Chin Chen. *Cache Design and Performance in a Large-Scale Shared-Memory Multiprocessor System.* PhD thesis, University of Illinois at Urbana-Champaign, 1993.

[6] G. Cybenko, L. Kipp, L. Pointer, and D. Kuck. Supercomputer performance evaluation and the perfect benchmarks. In *International Conference on Supercomputing*, 1990.

[7] Keith Farkas and Norman Jouppi. How Useful Are Non-Blocking Loads, Stream Buffers and Speculative Execution in Multiple Issue Processors? In *High-Performance Computer Architecture*, pages 78–89, January 1995.

[8] John W.C. Fu, Janak H. Patel, and Bob L. Janssens. Stride directed prefetching in scalar processors. In *International Symposium on Microarchitecture*, pages 102–110, December 1992.

[9] Edward H. Gornish, Elana D. Granston, and Alexander V. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *International Conference on Supercomputing*, pages 354–368, June 1990.

[10] Edward H. Gornish and Alexander V. Veidenbaum. An Integrated Hardware/Software Data Prefetching Scheme for Shared-Memory Multiprocessors In *International Conference on Parallel Processing*, Aug. 1994.

[11] Y. Jegou and O. Temam. Speculative prefetching. In *Supercomputing*, 1993.

[12] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *International Symposium on Computer Architecture*, pages 364–373, May 1990.

[13] Alexander C. Klaiber and Henry M. Levy. An architecture for software-controlled data prefetching. In *International Symposium on Computer Architecture*, pages 43–53, May 1991.

[14] K. Krishnamohan. Applying rambus technology to desktop computer main memory subsystems, March 1992.

[15] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Architectural Support for Programming Languages and Operating Systems*, pages 62–73, 1992.

[16] Subbarao Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *International Symposium on Computer Architecture*, pages 24–33, May 1994.

[17] Steven A. Przybylski. *Cache and Memory Hierarchy Design.* Morgan Kaufmann Publishers, Inc., 1990.

[18] Alan Jay Smith. Cache memories. *Computing Surveys*, 14(3):473–530, September 1982.