Second-level Cache Organization for Data Prefetching

Sunil Kim
Advanced Processor Development
IBM Austin
Austin, TX

skim@austin.ibm.com

Alexander V. Veidenbaum
Department of EECS
University of Illinois
851 S. Morgan St.
Chicago, IL 60607-7053
alexv@eecs.uic.edu
312-413-9657 (phone)
217-244-1351(fax)

Key Words: Data Prefetching, Secondary Caches, Memory Hierarchy

February 23, 1996

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. We concentrate on stride-directed prefetching and study stream buffers and L2 cache prefetching. We show that proposed stride-directed organizations/prefetching algorithms do not work as well in L2 caches and describe a new stride-detection mechanism. We study an L2 cache prefetching organization which combines a small L2 cache with our stride-directed prefetching algorithm. We compare our system with stream buffer prefetching and other stride-detection algorithms. Our simulation results show this system to perform 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 a certain domain of applications. However, programs wth large data sets or low data reuse, such as scientific applications, often have high miss rates. Combined with long 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 execution of these instructions initiates data prefetching, and as a result, the data is fetched into caches or prefetch buffers. Software prefetching can, in theory, prefetch data with any access pattern if the pattern can be determined at compile time. In practice, it has its limitations due to imprecise compiler analysis, lack of run-time information, and extra instruction overhead.

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: variations on these schemes include *always prefetching, prefetching on misses*, and *tagged prefetching.* Another prefetching method is stream buffers, proposed in [12]. In this method, when a cache miss occurs, a stream buffer is assigned, and the stream buffer prefetches successive cache lines starting at the miss address. Both of these methods cannot deal with 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 deal with arbitrary constant strides and are discussed in more detail below.

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 three main reasons for performing prefetching only at second-level caches:

1. On-chip primary cache is outside system designer's control while off-chip secondary cache very often is not and can be made to support prefetching.

2. On-chip primary cache has to be rather small and very simple to maintain low access time, typically 2 to 3 clocks, as system clock rates reach or exceed 200MHz in DEC, Intel, MIPS, Sun processors and head for over 500MHz. Prefetching is likely to increase the primary cache access time and thus should avoided at this cache level.

3. Secondary caches are migrating onto the CPU chip or the same multi-chip module (e.g. Alpha21164 and Pentium Pro)) and their latency becomes significantly lower. At the same time on-chip primary caches are becoming lock-up free. This may allow us to tolerate the lower L2 cache latency through instruction scheduling or dynamic execution while prefetching into second-level caches.

The small, fast, on-chip primary (L1) caches have a high L1 cache miss rate, especially for scientific applications. Half of the benchmarks in the suite used in our simulations have 10 % or higher L1 miss rates for an 8KB L1 cache (see Table 2). This demands a low miss rate L2 cache, but again this is not the case for our benchmarks: 4 out of 10 have 30% or higher L2 miss rate for 512KB L2 cache.

It is very important to achieve a low L2 cache miss rate as well as to reduce an L2 miss service time, especially when using a small, fast L1 cache. Both of these goals can be achieved by prefetching at the L2 cache. The goal of this paper to find an L2 cache architecture to effectively handle this case.

Of the various hardware data prefetching architectures proposed in literature [18, 1, 8, 11, 12, 16] only the last two concentrate on L2 prefetching. The rest focus on primary, on-chip (L1) prefetching. [12] proposes one L2 prefetching approach using stream buffers. [16] improves on it and compares it with a non-prefetching L2 cache. Our research was motivated in part by the need for more studies of L2 prefetching.

Simple hardware prefetching such as tagged prefetching [18] and stream buffers [12] 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 prefetching methods are effective only for memory accesses with strides shorter than a cache line, not sufficient for some scientific applications.

Stride-directed prefetching [1, 8, 11, 16] is much more appropriate for our needs. In stride-directed prefetching, a *memory reference stream*, which is a sequence of data addresses generated by a given memory reference instruction, is detected by hardware and its stride calculated. The calculated stride is used to predict and prefetch future memory accesses. Stride-directed prefetching is effective for memory references with any constant stride. In most previous studies [1, 8, 11], a memory reference stream is detected by using an instruction address of the memory reference. This approach, which we refer to as *instruction-address-based* prefetching, is quite successful in L1 prefetching.

Another approach to L2 prefetching described in [16] combines stream buffers with stride detection. This is the only method specifically aimed at L2 prefetching. It has been further examined in [7]. This approach does a very good job in limiting the memory requirements of prefetching by applying a "filtering" mechanism to L1 misses before generating a prefetch. It uses a heuristic approach to stride detection which is not as powerful as the *instruction-address-based* method. In addition, this method does not use an L2 cache but achieves performance comparable to a large non-prefetching L2 cache with a very small number of stream buffers.

The questions we want to answer are

- what should be the prefetching architecture for L2 cache?
- Which stride detection method should be used?
- How well can it be made to work?
- What is the best overall L2 organization with prefetching: stream buffers and how many of them, a "real" cache with prefetching and how large a cache, or simply a large, non-prefetching cache?

This paper attempts to answer these questions. First, we discuss stride detection in L2 caches and propose our stride detection mechanism in order to see how well stride detection works. Next, we study the performance of stream buffer prefetching, in particular the effect of a large number of stream buffers, with and without stride-detection. Finally we compare a traditional cache (cache-only), prefetching without a cache (prefetching-only), and prefetching with a small cache.

2 Difficulties in L2 stride detection

Instruction-address-based prefetching can be difficult to apply to L2 prefetching because instruction addresses may not be readily available at L2. Asumming it is available and can be

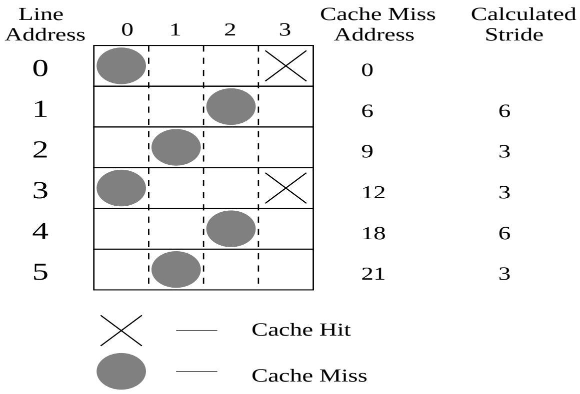

Figure 1: Stride Detection Scenario Based on L1 Cache Misses: Cache Line Size = 4 Words, Access Stride = 3

used to distinguish reference streams, a constant stride is not easily detected at L2 cache even when a given memory reference stream has a constant stride. This happens because not all data accesses are visible to the L2 stride detection unit, rather it sees only L1 cache miss addresses. Since the stride detector constantly checks consecutive data addresses to make sure it sees the predicted stride, missing addresses (which hit in L1 cache) may lead to aborting and starting over with a wrong stride prediction. This happens both within a given stream and across reference streams.

For example, consider a single reference stream with a stride of 3 and a cache line size of 4 words. A stride calculation based on two consecutive miss addresses alternates between 3 and 6. Figure 1 illustrates such a case.

Between different reference streams the problems arise due to spatial locality or due to cache conflicts in L1 cache. The former causes a hit address to be invisible to the L2 stride detector. The latter causes "extra" misses to be seen by the stride detector. To our knowledge, L2 cache *instruction-address-based* stride detection has not been studied in the literature.

Another method for L2 stride detection was proposed in [16]. It used allocation filters to eliminate random and stride-1 references. The remaining references went to a stride detector based on the following heuristic. A program address space is partitioned to separate references into different streams. References addressing the same partition are considered to be in the same stream. The problem is choosing partition size which accomplishes this. The authors propose that it should be supplied by either a programmer or a compiler. This does not seem an easy task and calls for a new solution. A second heuristic they proposed was "minimal" distance which detects a stream by finding its address distance with all active prefetch streams. They found it worked as well but needed more hardware. It was also used by [7]. Given these

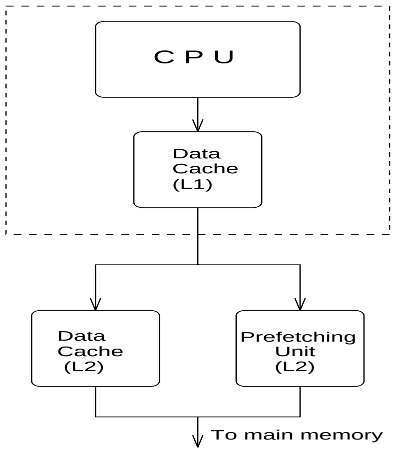

Figure 2: System Architecture for Read Access

difficulties we propose a new stride-directed prefetching scheme, called *loop-based prefetching*, to address the problem of memory reference stream detection at L2. This scheme does not rely on instruction addresses or address space partitions. We apply it in an approach similar to the one used by [16] as described below.

3 Prefetching Architecture

A general system organization for L2 prefetching used in this study is shown in Figure 2. It includes a CPU, an L1 and an (optional) L2 caches, and an L2 prefetching unit. We investigate three prefetching unit architectures:

1. *Stream buffers* and a miss table unit, which is similar to the unit-stride filter [16], is our baseline prefetching architecture.
2. *Loop-based prefetching architecture* uses our new stream detection unit to detect memory reference streams and compute their strides based on the knowledge about program loop iterations. It is added to the baseline architecture.
3. *Instruction-address-based architecture* uses an instruction address (PC) to identify memory access streams and compute their strides for prefetching.

The baseline architecture is chosen because of its simplicity, because it does not use memory reference stream and stride detection, and has been shown effective in L2 prefetching in [16]. The loop-based stride detection unit replaces the heuristic non-unit stride filter used in [16]. Figure 3 shows the baseline prefetching architecture (inner dotted box) and the loop-based

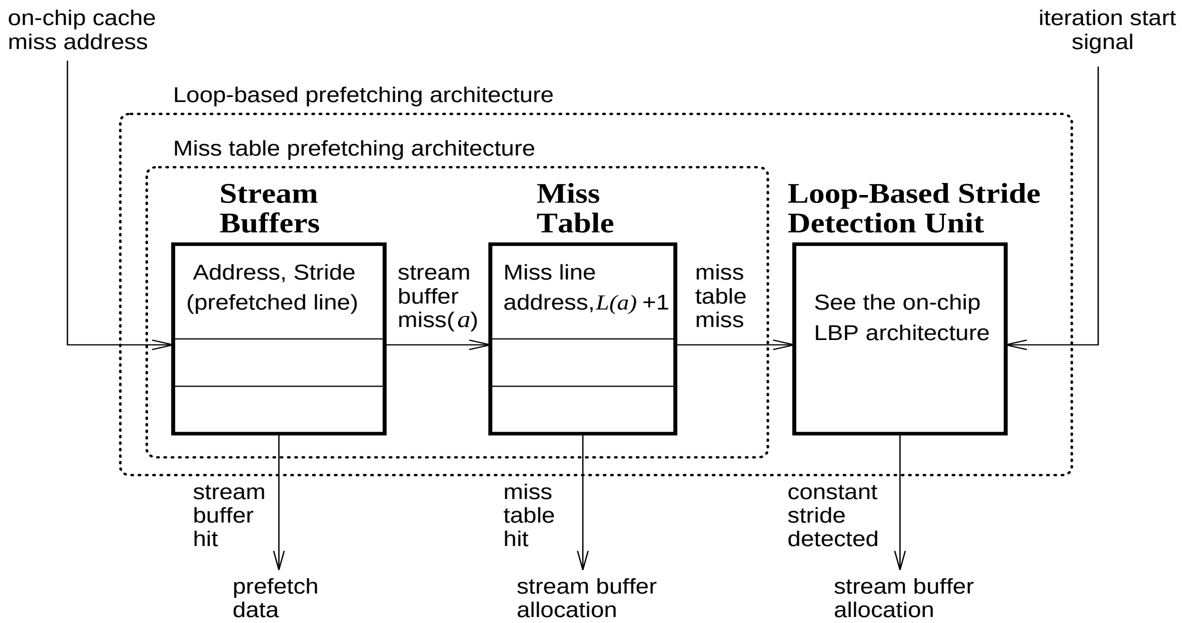

Figure 3: L2 Prefetching Architectures

prefetching architecture (outer dotted box). Stream buffer and miss table hits filter out stride-1 references which otherwise "confuse" the stride detector.

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, the stream buffers only generate and test address information for prefetching. For instruction-address-based prefetching architecture data is stored in the additional storage associated with each Reference Prediction Table (RPT) entry. Thus RPT is similar to stream buffers in this sense.

The stream buffers, the miss table, loop-based prefetching and instruction-address-based prefetching architectures are described in the next four sections.

3.1 Stream Buffers

Stream buffers were initially proposed in [12] and were also used in [16]. We use a similar stream buffer architecture.

Our stream buffer stores a data address (DA) and stride (St) information and (optionally) the cache line itself. The data address is used for tag compare to determine hit or miss. While a stream buffer can store several consecutive lines (referred to as *stream buffer depth*) in this study we use the depth of one.

On an L1 cache miss all the stream buffers are checked concurrently and in parallel with the L2 cache, if one is present. A prefetch request is issued for data address $DA + St$, and the DA is incremented by the St if there is a tag match in a stream buffer (*a stream buffer hit*). The actual cache line is returned to L1 cache either by the stream buffer or the L2 cache. If there is more than one matching stream buffer, only one of them generates a prefetch request.

If an L2 cache is present it is checked before a prefetch address generated by a stream buffer (note that this is not an L1 miss address) is issued to memory. If the cache already has the line containing the data to be prefetched, the prefetch request is discarded. Otherwise, the prefetch request fetches the line into the cache from the main memory. In this case the stream buffers do not store the cache line. If there is no L2 cache, the stream buffer stores the prefetched line.

A stream buffer is allocated to a reference stream on an L1 miss by a separate memory stream detection unit, such as *miss tables* or *loop-based stream detection units* (described below). The initial values of DA and St are assigned at this point. The stream buffer can be re-allocated to a new stream after if all the buffers are active. Stream buffers are replaced based on the LRU replacement policy.

3.2 Miss Table Prefetching (MTP) Architecture

This architecture uses a *miss table* to detect memory accesses with short strides or with good spatial locality and allocates stream buffers for such accesses. A miss tables stores line addresses. The table is fully-associative and indexed by line addresses. It is similar to the filter-based unit stride stream method proposed in [16],

A data address (M_a) missing in the stream buffers is processed by the miss table as follows ($L(M_a)$ is the line address of data address M_a.

if $L(M_a)$ hits in the miss table **then**

  allocate a stream buffer $DA \leftarrow$ the first word address of line $L(M_a) + 1$,

    $St \leftarrow$ the line size

  issue a prefetch for DA.

else

  allocate a new entry in the miss table with line address $L(M_a) + 1$.

endif

New entries are allocated in a circular FIFO order. An entry is not freed or invalidated on a hit. This method of managing the miss table is chosen to simplify its architecture.

3.3 Loop-Based Prefetching (LBP) Architecture

Memory accesses with stride longer than a cache line size cannot be detected and prefetched by the miss table method. To do so requires identifying individual memory reference streams and computing the strides. We propose loop-based prefetching in order to do so and improve on the stride detection heuristics of [16].

The loop-based prefetching (LBP) architecture contains a *loop-based stream detection unit* in addition to the MTP unit. The LBP unit detects memory reference streams without the use of load instruction addresses as described next.

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 issue order of load/store instructions inside the loop body remains constant from one iteration to the next (assuming in-order issue and absence of branches). If the beginning 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 4.

Each iteration of a loop starts by providing 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 number of a load within an iteration is used by the prefetching unit as an index into the RPT. An entry at this address 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. 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. The *iteration start* signal can generated by a special instruction or by an instruction unit detecting a backward branch.

Other RPT operations, such as updating memory access history and computing strides, are implemented in a way similar to the proposals in [1, 8, 11].

This method can be extended to handle multiply-nested 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 by a load in a loop. However, the prefetching unit handling the information becomes very complex. We use the simple RPT mapping hardware mechanism ignoring branches and concentrating on inner loops only which account for most of the memory accesses in a program (see Table 2). Branches in a loop are not handled in any special way and may cause misses. Prefetching of streams affected by branching is restarted if and when a "stable stride" is detected after the branch.

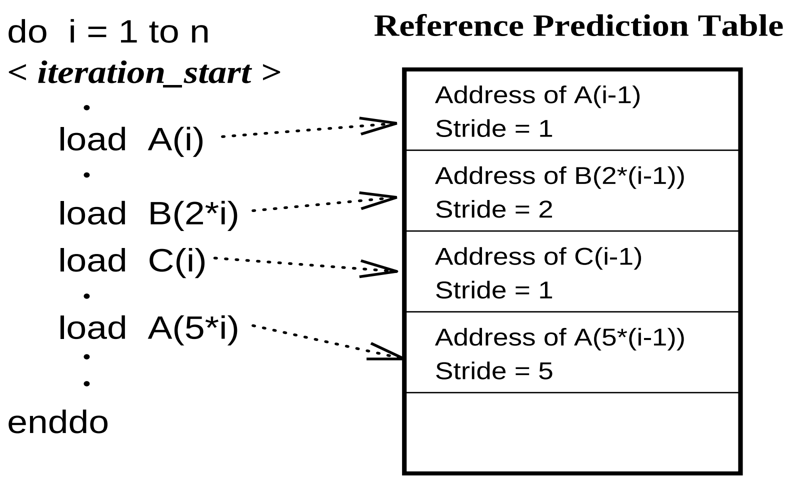

Figure 4: Reference Prediction Table Mapping in Loop-Based Prefetching

The organization of the loop-based stride detection unit is shown in Figure 5. 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. The unit is accessed with an addresses (M_a) that missed in both the stream buffers and the miss table unit. An RPT entry pointed by the RPT index counter is accessed with the miss address, and St_l is computed. As soon as a stable stride is detected, that is, a newly calculated stride is equal to the old stride, a new stream buffer is allocated with the stride and the data address equal to the miss address plus the stride. A prefetch request is issued for the data address as well. The following algorithm describes the operation of the prefetching unit.

if $M_a = DA_l$ and Stable **then**
  allocate a stream buffer
    $DA \leftarrow DA_l + St_l$,
    $St \leftarrow St_l$.
  issue a prefetch for DA.
else
  $Stable \leftarrow St_l = M_a - (DA_l - St_l)$. /* Set Stable bit */
  $St_l \leftarrow M_a - (DA_l - St_l)$. /* calculate a new stride */
endif
$DA_l \leftarrow M_a + St_l$.

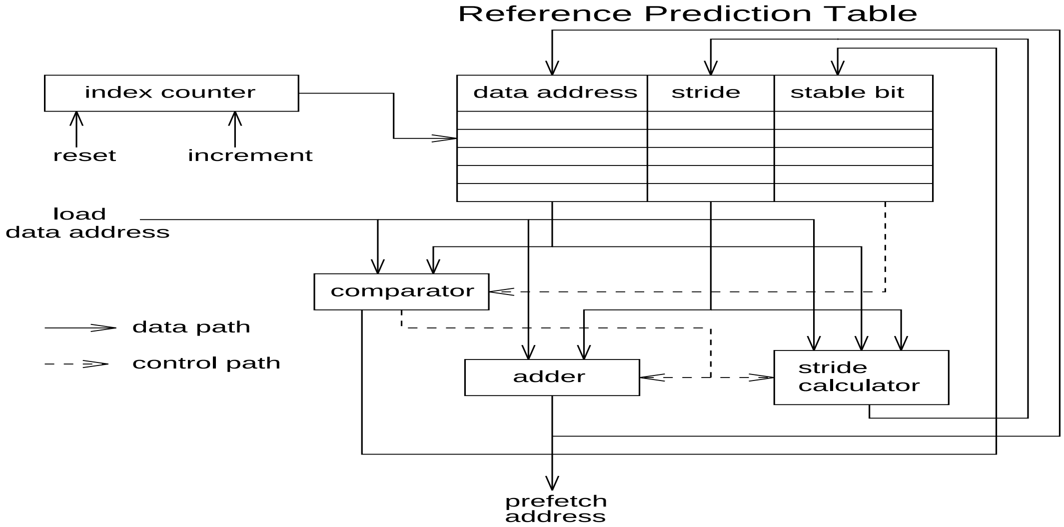

Figure 5: Loop-Based Stride Detection Unit

3.4 Instruction-Address-Based Prefetching (IAP) Architecture

The IAP architecture consists of an RPT, a comparator, an address adder and a stride calculator, as shown in Figure 6. Each RPT entry contains an *instruction address*, a data address (DA_p), a stride (St_p), and a stable bit (Sb). The IAP architecture we simulate is very similar to those proposed in [8, 11].

The RPT is a fully-associative table indexed by an instruction address. When a load instruction is executed and the load misses the L1 cache, the instruction address is supplied to the prefetching unit along with the miss address (M_a). The instruction address is used to find a matching RPT entry. If a matching RPT entry is not found, a new entry is allocated for the load instruction, replacing an RPT entry using the Least Recently Used (LRU) replacement policy. The initial stride and stable bit are set to 0. The prefetching unit operates by having each entry make a state transition between the stable state and the unstable state, as shown Figure 7. This state transition is required to obtain a stable stride. In the two states, the prefetching unit reacts differently to an incoming L1 cache miss address (M_a). The following algorithm describes the operation of the prefetching unit in each state.

Unstable State ($Sb = 0$) :

  if $M_a = DA_p$ **then**

    $Sb \leftarrow 1$.   /* transition to the stable state, a stable stride detected. */

    issue a prefetch for $DA_p + St_p$.

  else

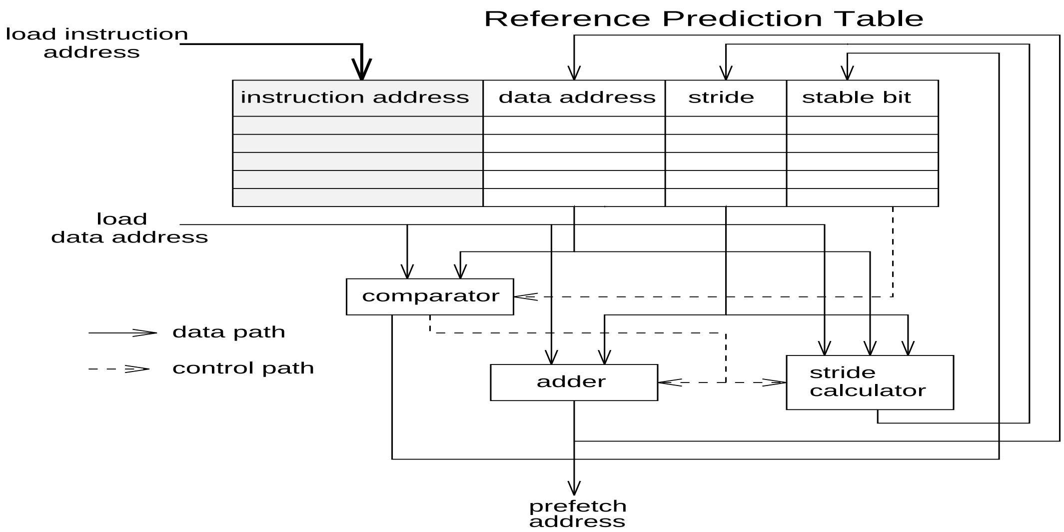

Figure 6: Instruction-Address-Based Prefetching Architecture

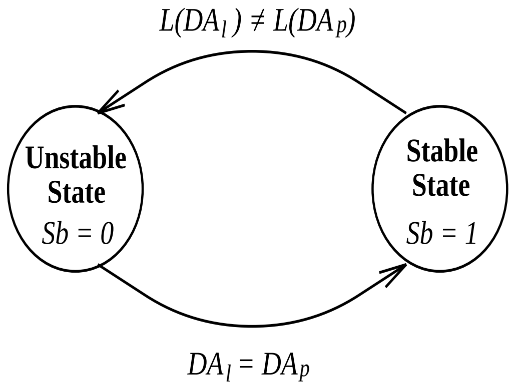

Figure 7: State Transition Diagram

$St_p \leftarrow M_a - (DA_p - St_p)$. /* calculate a new stride */

endif

$DA_p \leftarrow M_a + St_p$.

Stable State ($Sb = 1$):

if $L(M_a) = L(DA_p)$ **then**

issue a prefetch for $M_a + St_p$.

else

$Sb \leftarrow 0$. /* transition to the unstable state */

$St_p \leftarrow M_a - (DA_p - St_p)$. /* calculate a new stride */

endif

$DA_p \leftarrow M_a + St_p$.

4 Performance Evaluation Methodology

4.1 Simulation

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

Instructions fetches are ignored by assuming an instruction cache with one cycle access time and a perfect hit rate. The first and second-level data caches are direct-mapped and write-through. A cache line size is 4 words, and a word is 8 bytes in each case.

We use an 8 Kbyte L1 data cache throughout our experiments. A processor stalls on a read miss and does not stall on a write miss. On a write miss, no cache line is allocated and all writes are put in a separate write buffer of infinite size. A read hit and a write buffer access take one cycle to complete.

Only reads 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. When a prefetch request is issued, a cache line is *reserved* for the prefetched line and kept *reserved* until the prefetched line is returned by memory. A processor stalls on a read hit to the *reserved* line. A write hit on a reserved line proceeds without any delay, taking one cycle. If the reserved line contains a word accessed by an already pending write request, the line is updated accordingly.

The *iteration start* signal for a loop is obtained by inserting a special command inside the loop. From this command the MIPS trace tool generates a special instruction that is recognized as an iteration start by the simulators during program execution.

4.2 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,

Benchmark	Description	Read Access (M)	Write Access (M)	Total Memory Access (M)
APPBT	BT simulated CFD	17.47	6.51	23.98
APPSP	SP simulated CFD	16.49	5.31	21.8
ARC3D	3-D CFD	20.33	9.61	29.95
CG	Conjugate gradient	14.4	7.33	21.73
DYFESM	Structural dynamics	18.08	6.75	24.84
FLO52	2-D fluid dynamics	15.03	6.79	21.83
MDG	Molecular dynamics	20.70	8.96	29.66
MGRID	Multigrid kernel	14.53	8.6	15.39
OCEAN	2-D fluid dynamics	13.21	11.69	24.91
TRFD	Molecular dynamics	15.39	7.98	23.38

Table 1: Benchmark Characteristics

and MGRID are NAS benchmarks.

Table 2 shows the read characteristics of each benchmark. The third column of the table shows the L1 cache hit ratio of each benchmark. The fourth column shows the ratio of the total number of innermost reads to the total number of reads executed in each program. The innermost reads are the reads generated inside the innermost loops. The fifth column shows the ratio of constant stride read accesses to total read accesses. When a load generates a data address, a stride is calculated between the address and the last data address of the load. If the stride is equal to the previously calculated stride of the load, we consider it as a *constant stride* and count the access as a constant stride read access. These ratios were measured dynamically during program execution. In the last column, the ratio of the constant *long* stride read accesses to the total constant stride read accesses is shown. When a stride is larger than a cache line size (or negative), it is considered as a long stride in the statistics. Notice the large number of long stride accesses for some benchmarks.

For programs other than MDG, the innermost reads account for the most read accesses of a program. MDG shows a noticeably lower innermost read percentage, 65.4 %, as compared to the other benchmarks. Note that CG and DYFESM have relatively low constant stride read 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.

Benchmark	Total Read (M)	Read Hit Ratio (L1)	Innermost Read Percentage	Constant Stride Read Percentage	Constant Long Stride Read Percentage
APPBT	17.4	93.2	86.6	80.51	57.88
APPSP	16.4	86.1	95.7	86.41	44.6
ARC3D	20.3	86.8	86.1	83.37	31.87
CG	14.3	67.5	95.5	61.55	0.05
DYFESM	18.0	87.0	89.3	71.92	5.97
FLO52	15.0	82.5	99.5	92.64	5.75
MDG	20.7	96.9	65.4	82.47	2.64
MGRID	14.5	94.8	99.9	92.49	0.40
OCEAN	13.2	76.3	99.3	96.93	31.9
TRFD	15.4	92.0	97.6	87.16	49.49

Table 2: Read Characteristics

4.3 Performance Metrics

Cache miss rates are used as the main performance metric. A read access on a *reserved* cache line for prefetched data is counted as a cache hit because it at least partially hides the memory latency.

Prefetch prediction miss rate (PPMR) is used to evaluate how well future data accesses are predicted by prefetching units. This metric is defined as

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

Finally, we measure the total memory traffic, that is, the total number of memory requests generated by cache misses and prefetch requests. In a cache-based system, a higher cache hit rate directly implies less main memory accesses and a reduction of memory contention. Prefetching, however, usually increases the total number of memory accesses and memory contention even though it improves the cache hit rate. It is necessary to look at the total memory traffic as well as the cache hit rate.

We did not use execution time as a performance metrics because it requires detailed timing information about the memory subsystem (as well as the processor), and the results may vary widely depending on memory subsystem characteristics. These measures also depend on how far in advance prefetches are issued before actual memory accesses take place, a different issue in prefetching studies, which is not addressed here.

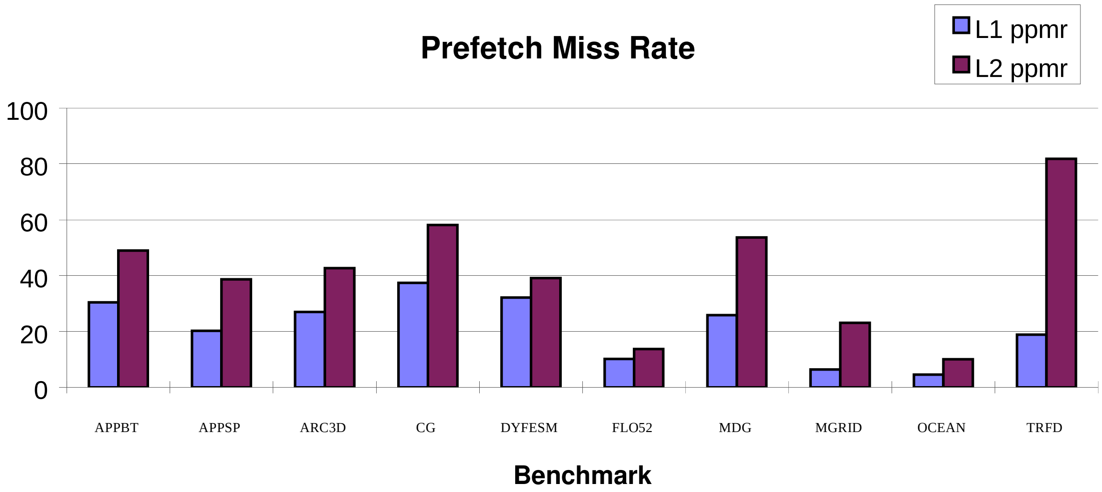

Figure 8: L1 and L2 Prefetching Prediction Miss Rate (PPMR), RPT size = 256

5 Experimental Results and Analysis

Let us start by examining the effectiveness of L2 prefetch methods. First, we compare the IAP prediction miss rates for L1 and L2 caches. Figure 8 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 disturb the miss address streams, which otherwise would be consistently visible to L2 prefetching units. The results indicate that stride-directed prefetching is not as effective at level-2 as at level-1, even if memory reference streams are accurately detected.

5.1 Stream Buffer Prefetching

In this section we evaluate the effect of increasing the number of stream buffers for all three prefetching methods: IAP, LBP and MTP in the absence of L2 caches. Data is stored in stream buffers or, for the IAP method , it is placed in the RPT, which has a storage associated with each entry. 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.

We present the results for 16, 64, and 256 stream buffers (RPT size). Even the smallest size is larger than what has been previously studied since we believed extra locality can be

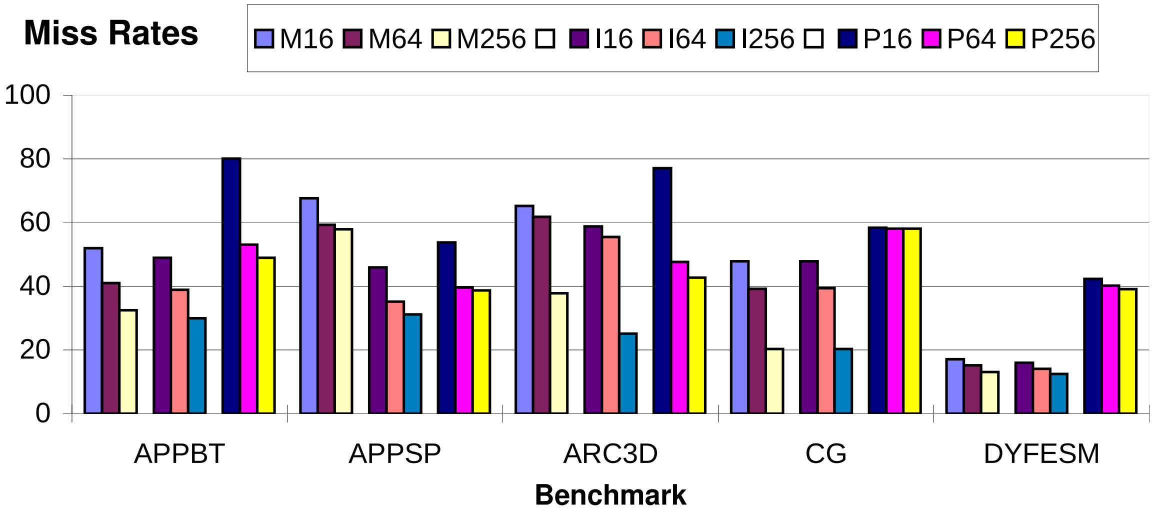

Figure 9: Stream Buffers Miss Rates. *Legend*: M=MTP, L=LBP, I=IAP followed by the number of stream buffers.

exploited with more buffers. The miss table size was determined in other experiments and set to 32. Stream buffers and miss tables need to be fully associative and it may not be practical to increase their size beyond 64. But the larger size confirms 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 the 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 9 and 10 show stream buffer miss rates for each benchmark. The stream buffer miss rate is measured as the number of stream buffer misses over the total number of L2 memory accesses, that is, L1 cache misses. As the number of stream buffers increases, the stream buffer miss rate is continuously improved in most cases. Increasing the number of stream buffers is beneficial since it increases storage for prefetched data and thus opportunity for reuse as well as providing more chances to initiate prefetches. For all benchmarks and most parameters, MTP and LBP perform better than IAP. LBP shows equal or lower stream buffer miss rates than MTP for all benchmarks and parameters.

Let us now discuss some specific cases. APPBT does not show much improvement with LBP because it has more than 60 % of long-stride misses with strides less than 2 cache line sizes. The Miss Table can also successfully capture such accesses and allocate stream buffers, not leaving

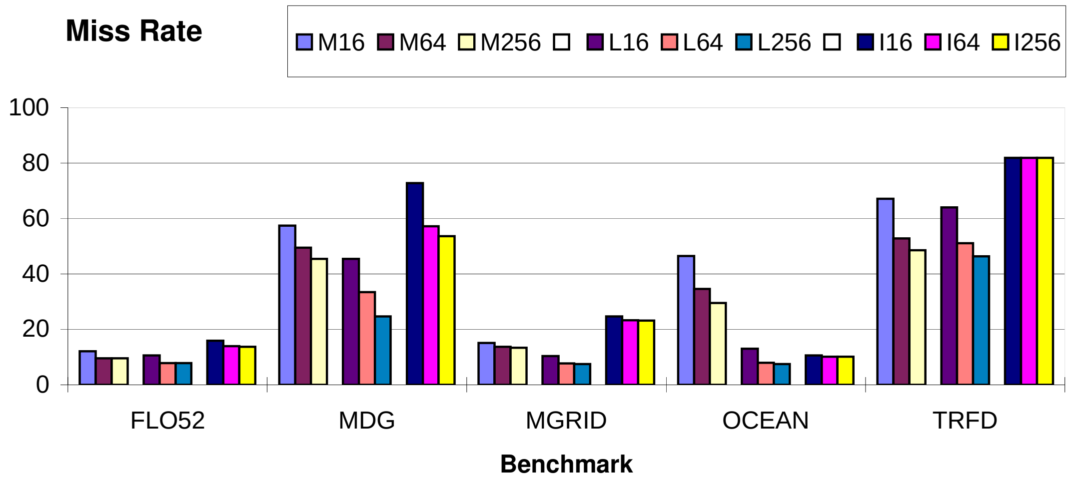

Figure 10: Stream Buffers Miss Rates. *Legend*: M=MTP, L=LBP, I=IAP followed by the number of stream buffers.

much room for improvement from LBP. TRFD has a large percentage of true long stride L1 cache misses, but shows only small improvements with LBP or IAP because of difficulty in obtaining a constant stride in part due to conflict misses in L1. TRFD performance improves noticeably with a 4KB, 2-way set-associative L1 cache.

One of the reasons for loss of performance in IAP is that data in the stream buffers is looked up by data addresses in MTP and LBP, whereas it is looked up by instruction addresses in IAP. A more important reason for the difference in performances is different prefetching styles. 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 even if they are observed across several memory reference streams. We quantify this more precisely in the next section.

5.2 Cross-stream locality

A series of L1 cache misses causing a stream buffer allocation by a miss table or a loop-based stream detection unit may or may not belong to the same 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 reference stream. We can now classify the 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.

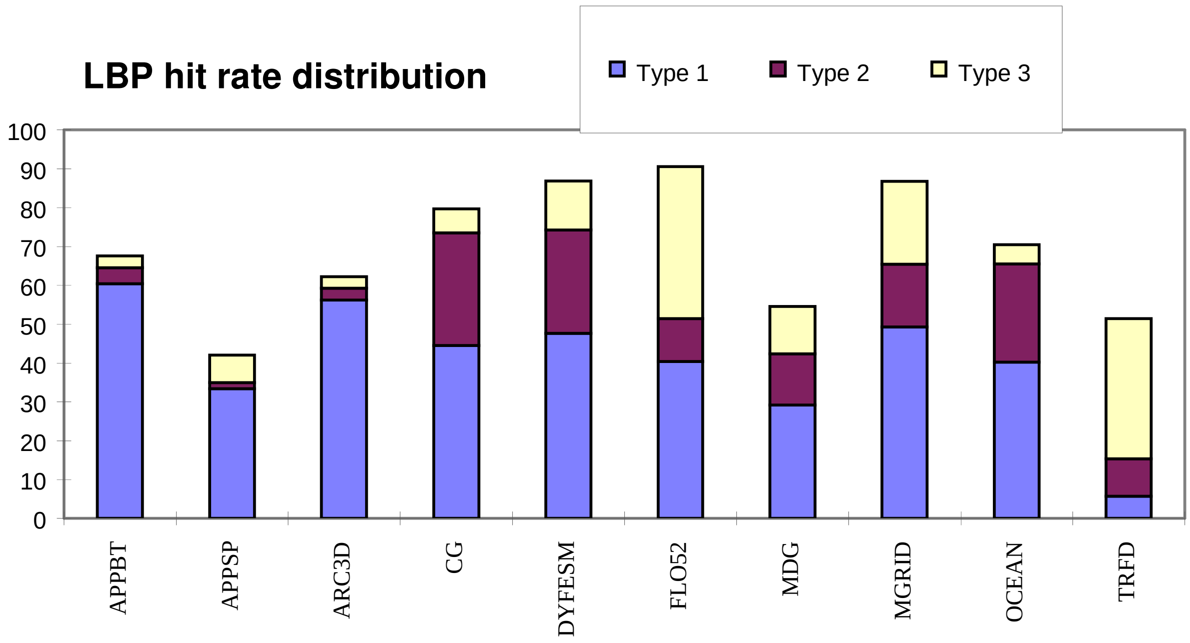

Figure 11: Distribution of Stream Buffer Hits for LBP: 256 Streams

- 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 successful prefetching based on memory access patterns with good spatial locality. The type 3 hit implies successful prefetching based on a memory access pattern of *one* memory reference.

The type distribution of stream buffer hits for LBP is shown in Figure 11. It shows that a large portion of the stream buffer hits are contributed by the type 1 and type 2 hits. A similar behavior was observed for MTP. Under IAP only type 3 hits are possible since the RPT is indexed by instruction addresses. Without the "real" L2 cache providing an opportunity for reuse, IAP-prefetched data cannot be used by other memory references. A large fraction of type 1 and 2 hits explains why more stream buffers help performance and calls for a use of additional storage. We will study how an addition of a small (by L2 cache standards) L2 cache helps to further improve performance.

5.3 Prefetching with Caches

In this L2 organization a small L2 cache is used together with a prefetching unit. The prefetched data is directly stored into the cache, and stream buffers only store data address and stride

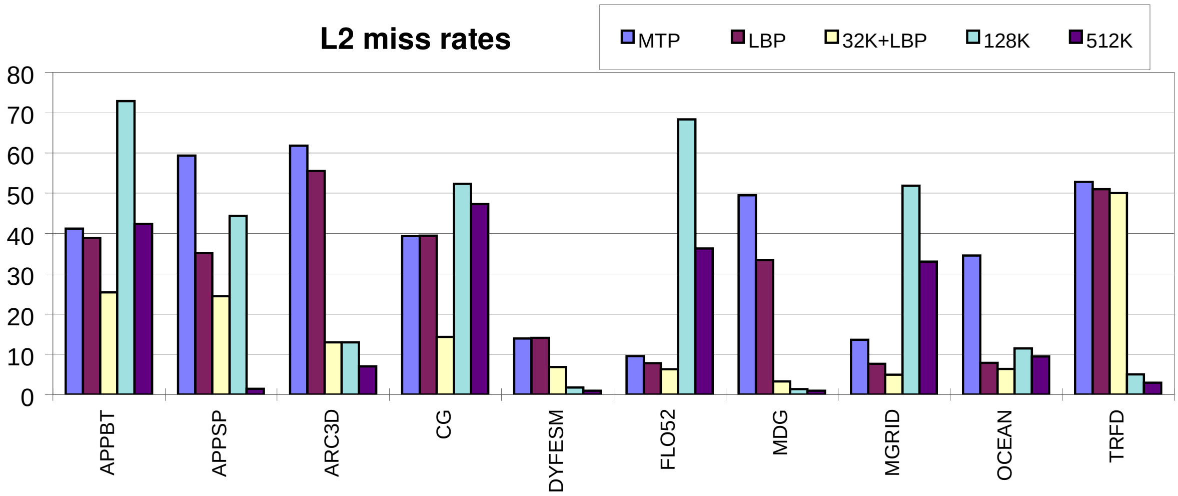

Figure 12: Miss rates for 64 stream buffers and L2 cache with and without prefetching

information. The L2 cache used in our simulations is 32 Kbytes, direct-mapped, and write-through. A line contains 4 words and the word size is 8 bytes. The number of stream buffers is fixed at 64 and the miss table size is fixed at 32.

The L2 cache miss rates of MTP and LBP prefetching methods, LBP with a 32KB L2 cache, and two sizes of non-prefetching L2 cache are shown in Figure 12. We did not show IAP or MTP with a small L2 cache because in both cases they perform worse than LBP, except for the case of ARC3D where IAP achieves a 10 % miss rate versus 13 % for LBP.

The addition of even a small L2 cache improves the performance for all benchmarks, significantly in half the cases. 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. The effect of larger non-prefetching L2 caches can also be seen in Figure 12. It shows that in half the cases largest size carries the day. However, prefetching can also be improved by increasing the "small" L2 cache size to "larger" but not "huge". In fact, DEC Alpha 21164 already has a 3-way set-associative L2 cache of 96KB. Adding prefetching to such a cache will make it look much larger.

For completeness, the hit rates of non-prefetching L2 caches and prefetching with a small cache are compared while varying the non-prefetching cache size. Figure 13 shows that the hit ratio difference between LBP with a 32K cache and L2 caches whose size varies as 32K, 128K, 256K, 512K, 1M, and 2M. A negative value means the LBP system has a higher hit ratio than the non-prefetching L2 cache by the amount shown. In general, as the L2 cache size increases,

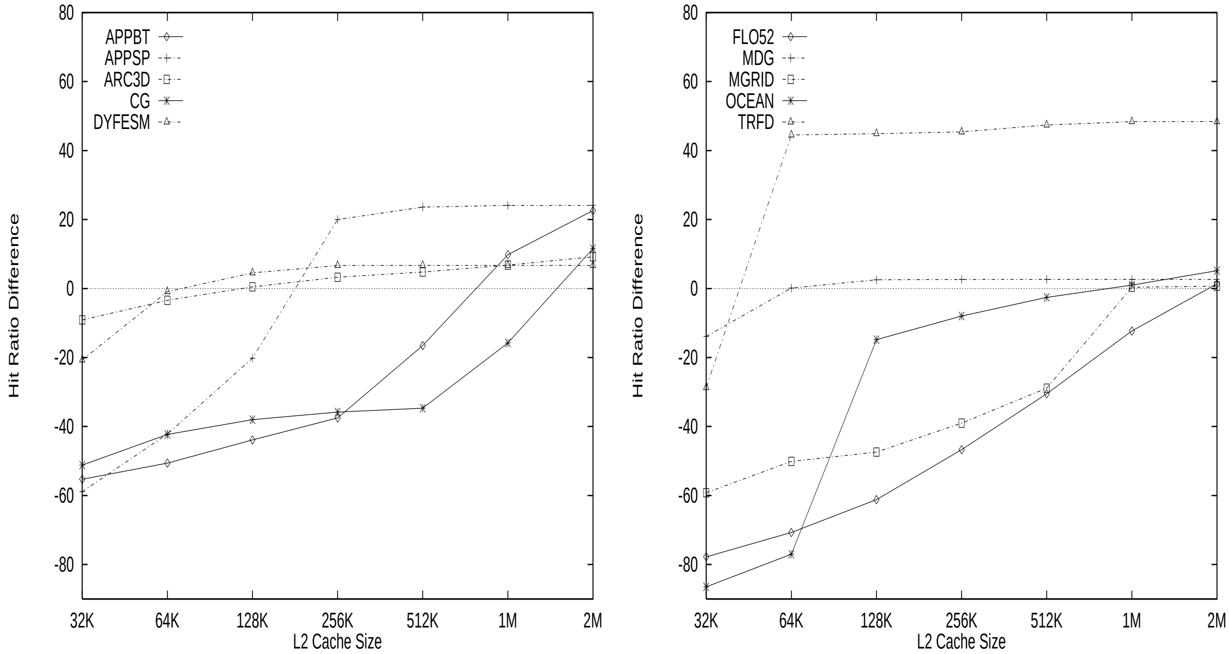

Figure 13: Hit Ratio Difference Between Loop-Based Prefetching versus Secondary Cache: LBP uses 64 Stream Buffers, a 32-entry Miss Table, a Loop-Based Stream Detection Unit and a 32K L2 Cache. L2 Cache-only uses 32K, 64K, 128K, 256K, 512K, 1M, and 2M Caches.

the hit ratios of non-prefetching L2 cache become higher than these 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).

For the benchmark with the worst LBP performance, TRFD, LBP suffered from cache conflict misses. We have verified that performance of LBP improves significantly when the associativity of the associated L2 cache is increased.

These results are consistent with [16] which already showed 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 a system with stream buffers significantly boosts the performance. It also requires a smaller number of stream detectors for most benchmarks. This combined organization is a better choice than prefetch-only and, in many cases, cache-only systems.

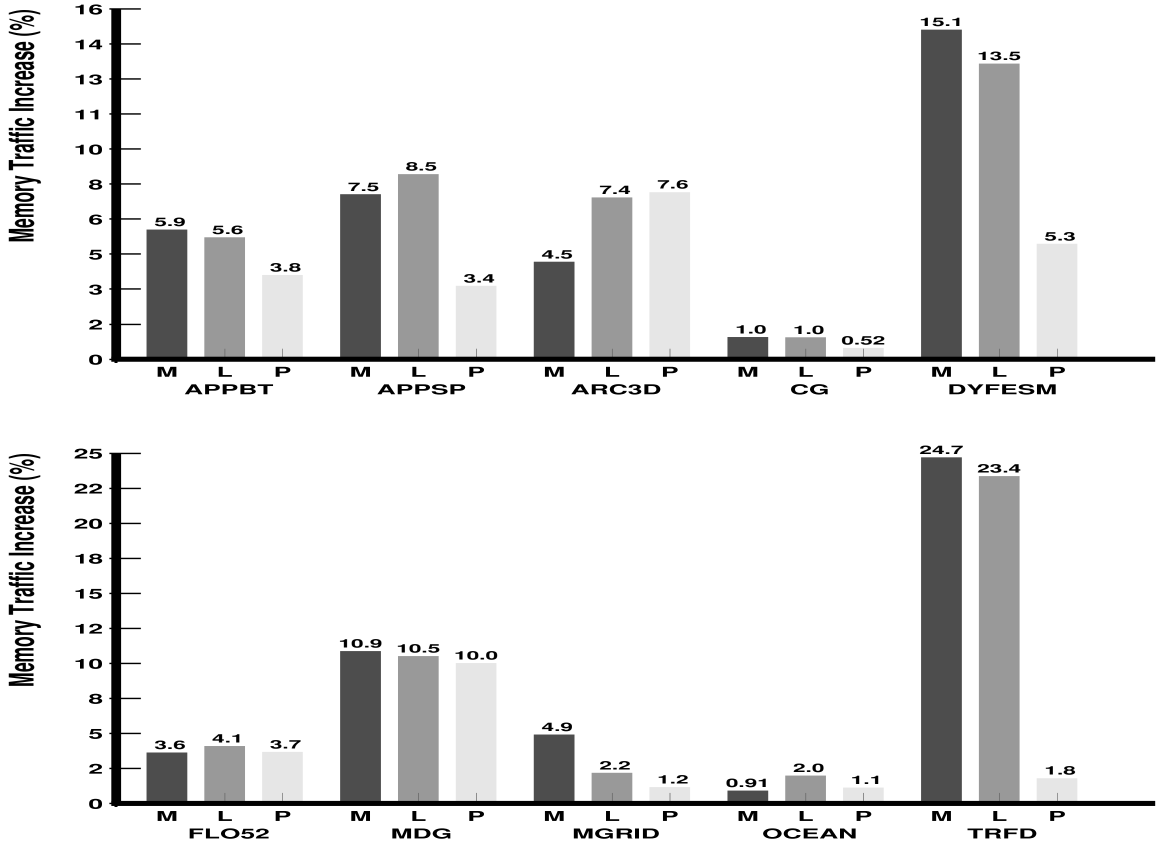

Figure 14: Prefetching Memory Traffic Increase over Cache-only System: 32 entries, L2 Cache Size = 32 Kbytes, **M** = MTP, **L** = LBP, **P** = IAP

5.4 Traffic increase under prefetching

One of problems associated with prefetching is the memory traffic increase due to unused prefetches. We compare memory traffic increase of MTP, LBP, and IAP with respect to the memory traffic generated with a 32 Kbyte L2 cache. Figure 14 shows the memory traffic increase for each prefetching method for all the benchmarks. In most cases the increase is very small, only DYFESM, MDG and TRFD show more than a 11 %, 15 %, and 25 % increase, respectively. This is due to the fact that constant stride detection is used, the stream buffer depth is only one, and because prefetch is not started until a stable stride is detected. For most benchmarks, IAP shows the smallest increase because it does not prefetch much data as its low stream buffer hit rates show. For APPBT, DYFESM, MGRID and TRFD, LBP has lower memory traffic increase than MTP. Considering the better cache miss rates and a relatively small memory traffic increase of LBP among the three prefetching methods, LBP with a small cache is the most efficient architecture among various L2 memory organizations studied.

6 Conclusions

We have shown that second-level (L2) prefetching has different characteristics compared to first-level (L1) prefetching. In particular, prefetching using instruction address information for stride detection was shown to be less effective. We proposed Loop-Based Prefetching (LBP) to perform stride-directed prefetching 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. We combined loop-based stride detection with MTP-based stride-one prefetching improving performance significantly. LBP was shown to be the best method improving over MTP or IAP prefetching.

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 even further. This organization was shown competitive to much larger non-prefetching caches in many benchmarks. This combined architecture exhibited only a small increase in memory traffic due to prefetching.

We have only discussed uniprocessor systems in this paper but loop-based prefetching architecture combined with a small L2 cache can be used effectively in large multiprocessor systems where external L2 caches of large size are expensive and do not perform as well as in uniprocessor systems.

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.