

Conflict Miss Elimination by Time-stride Prefetch ¹

Weiyu Tang Alexander Veidenbaum
Alexandru Nicolau Rajesh Gupta

ICS Technical Report

Technical Report #-00-15
March 2000

Center for Embedded Computer Systems
Department of Information and Computer Science
University of California, Irvine
{wtang, alexv, nicolau, rgupta}@ics.uci.edu

Department of Information and Computer Science
University of California, Irvine

¹This work was supported in part by DARPA ITO under DIS program.

Abstract

Many hardware cache prefetching mechanisms have been proposed to improve cache performance. Most of them rely on spatial locality prediction based on continuously monitoring miss addresses. While this kind of spatial locality prediction is useful in reducing the number of capacity and compulsory misses, it may not be effective in reducing the number of conflict misses. This paper presents the design, implementation and performance evaluation of a hardware cache prefetching mechanism to reduce the number of conflict misses. We introduce the concept of time-stride, the stride in time, which allows us to capture the characteristics of conflict misses. The stride is used to predict when an address will miss again and a prefetch scheduler is used to determine the time to issue a prefetch request to eliminate a future miss. Results obtained via trace-driven simulations show that our technique can eliminate as many as forty percent of total number of cache misses in some applications. Experiments show that our technique can function in tandem with spatial locality based hardware prefetch mechanisms such as stride-prefetcher to greatly improve cache performance. Comparison with victim cache, another hardware conflict miss elimination mechanism, shows that our technique and victim cache have comparable ability in conflict miss elimination.

Contents

1	Introduction	1
2	Related work	1
3	Time-stride	2
3.1	Motivation	3
3.2	Application analysis	5
4	Time-stride Prefetch	6
4.1	Hardware implementation	6
4.2	Hardware cost	9
5	Performance analysis	9
5.1	Simulation environment and benchmarks	9
5.2	Effect of MPB and MHT size	9
5.3	Effect of problem size	11
5.4	Results on complete benchmarks	12
5.5	Comparison with a stride-prefetcher	12
5.6	Comparison with a victim cache	13
6	Conclusion	14
	References	15

List of Figures

1	Matrix Multiply	3
2	Distribution of time-stride difference by memory accesses	4
3	Distribution of time-stride difference by cache misses	4
4	Matrix addition	5
5	System Design	6
6	Arrival of a miss address in TSP	8
7	Memory Access	8
8	Distribution of Time-strides	10
9	MPB Hit Rate	10
10	Miss Rate vs. Miss Elimination Rate	11
11	Miss Elimination Rate by TSP as MHT size is varied	12
12	Miss Eliminate by Stride-prefetch and TSP	13
13	Miss Elimination Rate by Victim Cache and TSP	14

1 Introduction

Cache becomes more important with the widening speed gap of processor and main memory. Many hardware cache assists have been proposed to improve cache performance. Normally, a cache assist consists of an on-chip fully-associative buffer and an assist policy. The on-chip buffer is often on the refill path of the cache for secondary lookup on a cache miss. The assist policy predicts memory access patterns and tries to avoid future cache misses for the predicted patterns.

Hardware cache assists have been actively researched for the following reasons:

- There is no change in the cache design. Cache hit time, which is on a critical path of a processor design, is not affected.
- The cost is small, just a small fully-associative on-chip buffer and some control logic for the assist policy.
- It is very effective for some applications whose memory access patterns can be predicted.

A commonly proposed cache assist is a hardware prefetcher. Some hardware prefetchers, such as stream buffer [7, 12], stride-prefetcher [2, 4] and Markov predictor [6] predict a miss address based on other temporally/spatially related addresses. Then the predicted address is prefetched immediately. Prefetching is useful in eliminating compulsory and capacity misses, but the effect on conflict misses is unpredictable.

In this paper, we propose a hardware prefetcher, the time-stride buffer, which is dedicated for conflict miss elimination. While other prefetchers only predict which addresses to prefetch and prefetch them immediately, our prefetcher predicts both which addresses to prefetch and when to prefetch them. If an address appears in the recently miss address history more than once, we predict this address will miss again. A prefetch scheduler determines when to issue a prefetch request so that the request can be issued on time.

For a conflict miss address which enters and leaves cache frequently, we conjecture that intervals between consecutive misses to the address may have small differences, especially for some scientific applications. For instance, a conflict between two array references in a loop may occur every N iterations. When an address misses in the cache, the "interval" between this miss and previous miss by this address can be calculated. Then the address is predicted to miss from cache again after the same "interval".

The rest of the paper is organized as follows. Section 2 reviews the related work. Time-stride, used to capture the characteristics of conflict misses, is proposed in section 3. The prefetch design based on the time-stride is described in section 4. Section 5 presents the simulation results. The conclusions of this work are summarized in Section 6.

2 Related work

A number of prefetch schemes for array and linked list prefetching have been proposed. All aim at identifying/predicting next element to be accessed and prefetching it.

Software prefetch [1, 8, 11] uses a special PREFETCH instruction to bring data into the cache in advance. It has the advantage of utilizing global information to identify memory addresses

most likely to miss from the cache. One disadvantage is that it will incur an execution cost even if the data is already in the cache. Another disadvantage is lack of run-time information, which makes it hard to detect conflicts. And code expansion may occur, which can have adverse effect on instruction cache.

Hardware prefetch has the advantages of utilizing run-time information. And there is no need for code recompilation. The disadvantage is the additional bandwidth requirement and the extra hardware.

Stream buffer [7] uses an array of cache line sized buffers to store prefetched data. On a cache miss, the stream buffer is searched to find whether the data is present. On a hit, the data is copied to cache and the stream buffer is refilled with successive addresses from lower memory hierarchy. On a miss, the successive addresses of the miss address will be prefetched into the buffer.

Stream buffer filter [12, 3] is proposed to improve the accuracy of prefetch and decrease the bandwidth requirement by allocating a stream buffer only when there are two cache misses to consecutive cache lines. It can also detect and prefetch striding data access by data space partitioning.

Stride-prefetcher [2, 4] can detect and prefetch striding data access by calculating the distance between two consecutive memory accesses by the same memory instruction. It requires access to instruction stream, which is not always available.

Markov predictor [6] prefetches based on miss history. It approximates a Markov process to correlate consecutive miss addresses for prediction. The implementation stores Markov model in main memory and is very costly.

Dynamic bypassing [5, 10] uses compilation-time or run-time information to eliminate conflict misses by caching only some addresses, especially those frequently used. [5] does dynamic bypassing at a coarse granularity and [10] needs access to instruction stream.

Victim cache [7] is a mechanism that is aimed specifically at conflict misses. It predicts that a line of data will be accessed again shortly and store the replaced data in a small fully-associative buffer on the refill path of the cache. On a cache miss, the victim cache is checked to see whether the data is present. If so, the data is copied from the victim cache to the cache. Victim cache is useful when two lines of data with spatial/temporal locality map to the same cache line and the accesses to both memory addresses are interleaved.

Blocking [9] eliminates capacity misses by partitioning one large dataset into many small datasets to exploit the temporal locality within the small datasets. But care should be taken when partitioning the dataset. Otherwise, many of the capacity misses will be transformed into conflict misses instead of being eliminated.

3 Time-stride

In this section, we describe our conjecture that the access pattern for a conflict miss address(CMA) can be predictable. Then time-stride is defined to capture the pattern of conflict misses. Simulations show that there are small differences between consecutive time-strides to a same address. Application analysis shows why conflict misses are predictable.

3.1 Motivation

In order to eliminate a conflict miss, the access pattern for the address should be predictable. Depending on the cache organization (cache size, line size, etc.), it is hard to predict which addresses will conflict with a CMA and when the conflicts will occur. However, as a CMA enters and leaves the cache frequently, we conjecture that a time-domain factor for a CMA can be used to capture the access pattern. We have the following definition to characterize the access pattern:

time-stride—the interval between two consecutive cache misses to a same address.

There are three ways to characterize the time domain: in cycles, memory accesses, or cache misses. Thus time-stride can be calculated in three ways: by cycles, memory accesses, or cache misses. Time-stride differences calculated by cycles is not very useful since it is highly dependent on the underlying processor model. We want to capture the characteristics of a CMA from the cache perspective.

```
#define N 127
for ( i=0; i < N; i++ )
    for ( j=0; j < N; j++ ) {
        a[i][j] = 0;
        for ( k=0; k < N; k++ )
            a[i][j] += b[i][k]*c[k][j];
    }
```

Figure 1: Matrix Multiply

On a cache miss to a CMA *addr*, a time-stride *ts* is calculated as the interval between the current miss and a previous miss to *addr*. We predict the future time-stride is equal to *ts*. Then the time for a future miss to *addr* can be calculated and prefetch can be done on time to eliminate the future miss. While stride prefetcher is our case, the time-stride does not need to be constant. This is because our time-stride is in time.

If the actual time-stride is much larger or smaller than the predicted one, then a CMA will be prefetched too early or too later to eliminate a future cache miss. The effectiveness of the prefetch depends on the differences between consecutive time-strides to the same address. Thus from the distribution of time-stride differences, we can see how good time-stride is in predicting future cache misses. The higher the percentage of time-stride differences close to 0, the better time-stride can be used to predict future cache misses.

Trace-driven cache simulation is used to collect time-stride differences calculated by memory accesses and by cache misses. A 32KB direct-mapped cache with line size of 32B is simulated. A commonly used scientific kernel—matrix multiply as shown in Figure 1 is used in the simulation.

Figure 2 and Figure 3 show the distribution of time-stride differences. Figure 2 uses memory accesses as the timescale. Figure 3 uses cache misses as the timescale. In Figure 2, 0 contributes 91 percent of all differences. In Figure 2, 0, 1, and 2 contribute 58 percent all differences. Time-stride calculated by memory accesses and cache misses is good for future cache miss prediction.

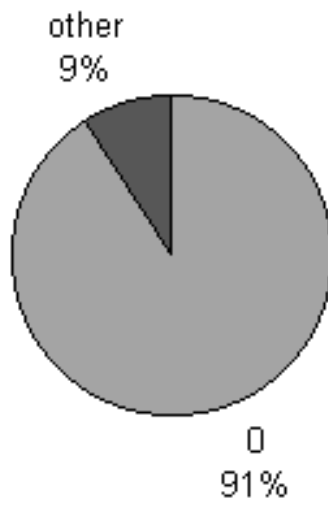


Figure 2: Distribution of time-stride difference by memory accesses

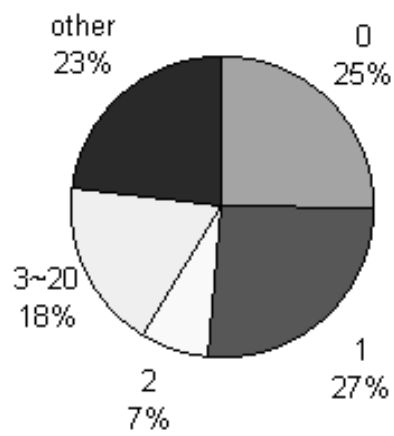


Figure 3: Distribution of time-stride difference by cache misses

3.2 Application analysis

Let's further understand why small values contribute a large portion of all time-stride differences. In scientific applications, computations in loops contribute more than 90 percent of all computations. In this kind of applications, memory (load/store) instructions in a loop can be classified into two categories: loop dependent and loop independent. A memory instruction is classified as loop independent memory instruction(LIMI) if it satisfies the following two conditions:

- the addresses accessed in each iteration are same.
- the order of accesses in each iteration is same.

Otherwise, the instruction is classified as loop dependent memory instruction (LDMI). As in the case of matrix multiply shown in Figure 1, $a[i,j]$ is a LIMI across loop k ; $b[i][k]$ is a LIMI across loop j ; $c[k][j]$ is a LIMI across loop i .

In a loop, on a cache miss for address $addr_1$, the address $addr_2$ currently in cache will be replaced. There are three possible relations between $addr_1$ and $addr_2$:

1. They are accessed by a same LIMI or two different LIMIs.
2. They are accessed by a same LDMI or two different LDMIs.
3. One is accessed by a LIMI and the other is accessed by a LDMI.

In scenario 1, $addr_1$ and $addr_2$ are accessed for every iteration of the loop. They will conflict and miss from the cache for every iteration of the loop. The time-strides calculated by memory accesses for both addresses are equal to the number of memory accesses in one iteration of the loop. The time-stride differences are 0, which contributes a large portion of all time-stride differences in Figure 2. For example in loop j of matrix multiply, if $b[i_1][k_1]$ and $b[i_2][k_2]$ map to the same cache line, then $b[i_1][k_1]$ and $b[i_2][k_2]$ will miss from the cache for every iteration of loop j . Time-strides for both addresses are equal to the number of memory accesses in one iteration of loop j .

Alternatively, time-strides calculated by cache misses for $addr_1$ and $addr_2$ are equal to the number of cache misses in one iteration of the loop, which tends to be fixed with small variations across the loop. Thus we can see a large portion of 0,1, and 2 in Figure 3.

```
for ( i=0; i < N; i++ )  
    a[i] = b[i] + c[i];
```

Figure 4: Matrix addition

Some cache misses in scenarios 2 and 3 can be eliminated by the use of time-stride. For example, Figure 4 shows the code for matrix addition. If $b[0]$ and $c[0]$ map to same address, $b[i]$ and $c[i]$ will conflict for every iteration of the loop. This conflict kills live spatial locality, thus

spatial prefetch can not eliminate cache misses here. Assume 32-byte cache line and each $b[i]$ or $c[i]$ takes 4 byte. After initial two misses to a cache line by $b[i]$ or $c[i]$, the time-stride of 1 can be calculated and following 6 misses to the same cache line are predictable and can be eliminated. Thus 75 percent of total number of cache misses can be eliminated by the use of time-stride.

For time-stride to be effective in predicting future cache misses, the access pattern of an application should have the following characteristics:

- existence of LIMIs
- the dataset accessed by LIMIs is relatively large

4 Time-stride Prefetch

In this section, we present the design of Time-stride Prefetch(TSP), a hardware prefetcher based on time-stride. Time-stride calculated by memory accesses needs number of memory instructions run in the processor, which is not easily available off-chip. We don't want to constrain TSP to be on-chip, so time-stride calculated by cache misses is used in the TSP design.

4.1 Hardware implementation

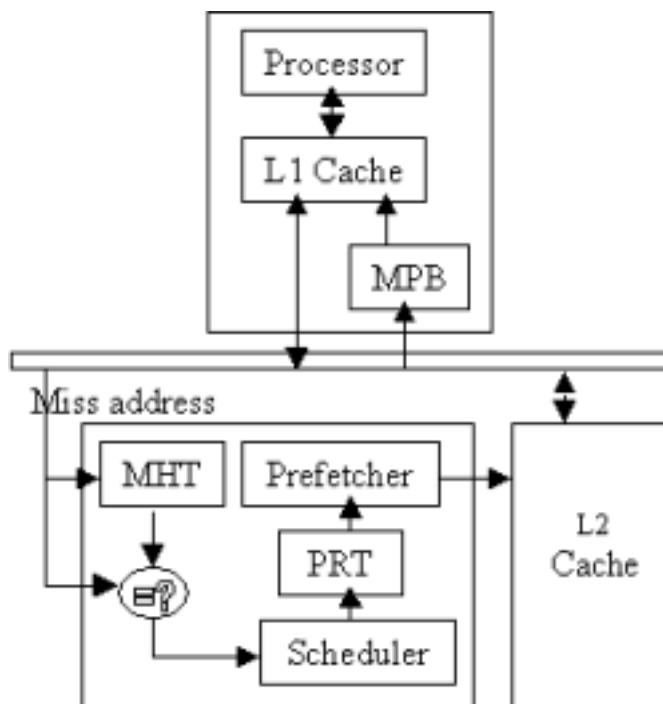


Figure 5: System Design

For TSP to eliminate conflict misses, it has to answer two questions: what is a CMA and when to prefetch it.

As a CMA will enter and leave the cache frequently, a miss address history is used to determine whether a miss address is a CMA. On a cache miss to an address $addr$, $addr$ is considered a CMA if a previous miss to $addr$ can be found in the miss address history. The longer the miss address history, the more CMAs can be identified. As a consequence, more time-strides can be calculated and more prefetches can be issued.

Once a CMA has been identified, there is a limited time-frame to prefetch this CMA so that a future cache miss can be eliminated. Scheduling is used to find an optimal time to issue a prefetch.

Figure 5 shows the design of TSP. It consists of 5 components:

- a miss prefetch buffer(MPB)
- a miss history table (MHT)
- a prefetch request table (PRT)
- a prefetcher
- a scheduler

The MPB is a small fully-associative cache on the refill path of the cache to store the prefetched lines. The line size of each entry in MPB is equal to the line size of the L1 cache. The MPB uses a FIFO replacement policy. The MPB is searched on a cache miss.

The MHT consists of an array of entries to keep recent miss addresses ordered in time. It uses a FIFO replacement policy.

The PRT consists of an array of entries to keep pending prefetch requests ordered in time. It uses a FIFO replacement policy. MHT and PRT are of the same size. The same *head* and *tail* pointers are used in both MHT and PRT to synchronize them.

The prefetcher prefetches data from L2 to MPB based on prefetch requests stored in the PRT.

The scheduler determines when to issue a prefetch request so that the prefetched data will arrive in MPB on time to avoid a future cache miss.

Figure 6 shows what happens on arrival of a new miss address. First, the address is filled in MHT to update the miss history. Then associative search is used to find the last occurrence of the miss address. We only prefetch for conflict miss addresses, which should appear in MHT more than once. When the last occurrence is found, a time-stride can be calculated as the difference between the two entries, which is equal to the number of misses between the two occurrence as every miss address occupies an entry in MHT.

A prefetch should be done not so early and not so late to eliminate a future miss. The scheduler determines the optimal entry in PRT to store a prefetch request for on time prefetch. For address $addr$, suppose a predicted time-stride is ts and the size of MPB is mpb_size , the scheduling window to store a prefetch request is from entry $(head + ts - mpb_size)$ to entry $(head + ts - 1)$. On next miss to $addr$, if the time-stride is equal to the predicted ts , the line for $addr$ has already been prefetched and is still in MPB. Thus a cache miss is eliminated.

However, the optimal entry to store the request is $(head + ts - \frac{mpb_size}{2})$ so that time-stride difference within $\frac{mpb_size}{2}$ of the predicted ts can be tolerated. If the optimal entry already has a

Given miss address *addr*

1. Fill entry *head* in MHT with *addr*.
2. Associative search for last occurrence of *addr* in MHT.
3. If *addr* found in entry *last*
4. Compute time-stride *ts* as $head - last$
5. Pass *head* and *ts* to scheduler to get an entry *en*.
6. Fill entry *en* in PRT with *addr* .
7. Endif
8. If entry *head* in PRT has a prefetch request
9. Use prefetcher to issue the prefetch.
10. Endif

Figure 6: Arrival of a miss address in TSP

prefetch request, the scheduler will find an empty entry close to entry ($head + ts - \frac{mpb_size}{2}$) and within the scheduling window.

Given memory access address *addr*

1. If *addr* is in L1 cache
2. Fetch data
3. Else if *addr* found in MPB
4. Fill data to L1 cache
5. Send *addr* to MHT
6. Else
7. Send *addr* to L2 cache and MHT
8. Wait for miss data to come
9. Endif

Figure 7: Memory Access

Figure 7 shows what happens on a memory access. When an address misses in both L1 cache and MPB, it is sent to both L2 cache and MHT. When an address misses in L1 cache and hits in the MPB, the data is copied from MPB into the L1 cache. Then the miss address is sent to the MHT. This is necessary so that a new time-stride for the miss address can be calculated and a new prefetch request can be issued to avoid a future cache miss by this address. Also, the arrival of a new miss address will trigger the prefetcher to issue a pending prefetch request so that the prefetched data can arrive in the MPB on time.

Sending an address to the MHT on a MPB hit doesn't incur additional traffic on the address bus. This traffic exists if prefetch is not used and miss fetch request will be issued through

address bus to L2 cache anyway.

4.2 Hardware cost

Assume a 32-bit address, a MHT of size n costs $4n$ bytes. And $n - 1$ comparators are needed for associative search in MHT. A PRT of size n costs $4n$ bytes. Cost of MPB is several (such as 8) cache line size buffers. Small amount of control logic is needed for the prefetcher and scheduler.

In current cache design, a typical L1 cache size is 32KB and L1 line size is 32B. The number of cache lines is 1K. A MHT with 1K entries can capture most conflict misses. A MHT and a PRT with 1K entries together cost 8KB. This is comparatively small considering that L2 caches often have sizes much larger than 256KB.

5 Performance analysis

5.1 Simulation environment and benchmarks

The benchmark suite consists of 7 programs from SPLASH-2 benchmarks (barnes, fmm, ocean, water, cholesky, lu, radix), 2 programs from SPEC92 benchmark (espresso, eqntott), dense matrix multiply and SPARSE 1.3, a sparse linear equation solver with matrix size of 200*200 and 5000 non-empty matrix elements. The programs are compiled on an SGI system with an R10K processor using MIPS and MIPSPro compilers with the following flags: -n32 (MIPS-III instruction set, 32b executable) and -O2 flag.

We use MINT-3 [13] simulator to model a single-issue, statically scheduled processor. Memory addresses generated by MINT-3 are fed into our memory hierarchy simulator. We don't model instruction cache. The memory hierarchy consists of an L1 cache, an L2 cache and main memory. The L1 cache is direct-mapped, 32KB with 32 byte lines. The L2 cache is multi-cycle, multi-bank, 2-way set-associative, lockup-free with cache size of 512KB with line size of 64B.

5.2 Effect of MPB and MHT size

We want to understand the effect of MPB and MHT size on the hit rate of MPB. For simplicity, matrix multiply with size of 127*127 is used in the simulations.

Figure 8 shows the percentage distribution of time-strides. Each point for a value of ts is calculated by dividing the number of time-strides with value of ts by total number of time-strides collected. We can see that locality exists in the distribution of time-strides. 86 percent of total number of time-strides are contributed by the following four groups of time-strides :

- from 2776 to 2780, 38.5 percent
- from 18 to 24, 24.9 percent
- from 2746 to 2751, 14.6 percent
- from 2761 to 2767, 7.9 percent

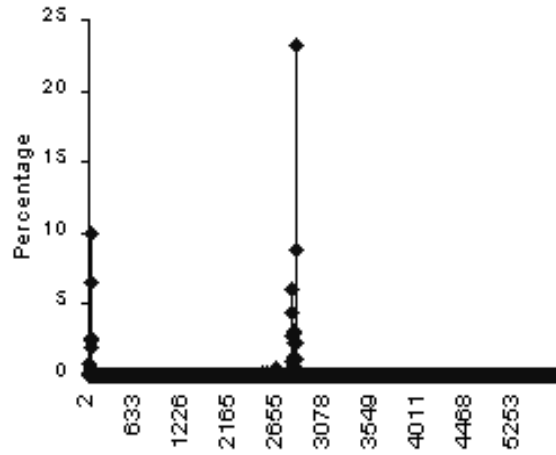


Figure 8: Distribution of Time-strides

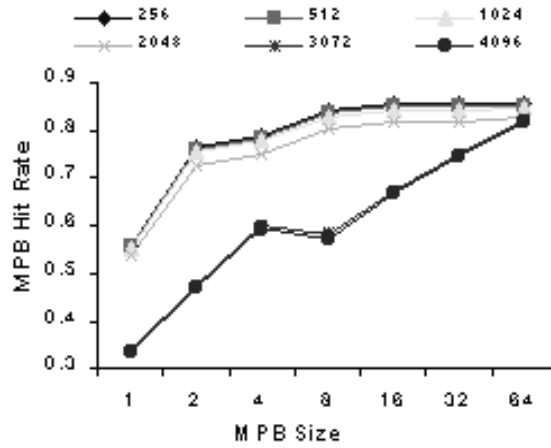


Figure 9: MPB Hit Rate

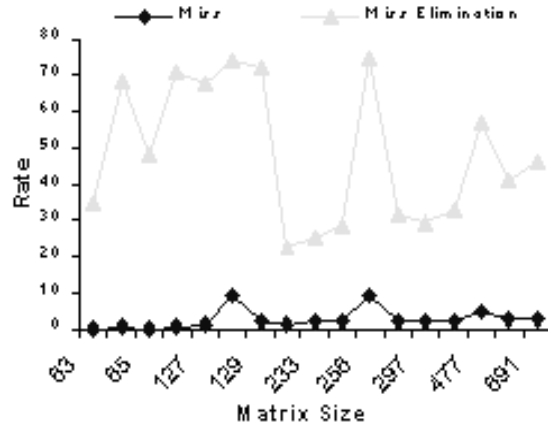


Figure 10: Miss Rate vs. Miss Elimination Rate

Figure 9 shows how the hit rate of MPB changes as we vary the size of MHT and MPB. The hit rate of MPB is calculated by dividing the number of hits in MPB by the number of prefetches issued by the prefetcher. The trends of MPB hit rate with MHT size from 256 to 2K and from 3K to 4K are almost same. The reason is that the number of prefetches is almost same from 256 to 2K and from 3K to 4K because of the time-stride locality shown in Figure 8.

With same MPB size, the configuration with small MHT size has a higher hit rate. As the size of MHT increases, the number of prefetches increases as well. Data in MPB will be flushed out of MPB more quickly and the hit rate in MPB will drop. With same MHT size, hit rate in MPB increases with larger size. The larger the size, the longer a prefetched data can stay in MPB, the larger time-stride differences can be tolerated.

5.3 Effect of problem size

We vary the size of matrix to see how the cache miss rate and miss elimination rate by TSP change with problem size. The miss elimination rate is calculated by dividing the number of hits in MPB by the number of total misses in L1 cache. MHT size of 4096 and MPB size of 8 are used in the simulation.

Figure 10 shows the relationship between the problem sizes and the miss rates. Some problem sizes, such as 63, 64 and 65, are chosen intentionally because they are either equal or close to a power of 2. Some problem sizes, such as 233 and 297 are chosen randomly. As can be seen in the figure, the miss rate increases sharply for some problem sizes. The miss rate for matrix with size 128*128 is much higher than that of matrix with size 129*129. As matrix with size 129*129 has more number of capacity misses with larger problem size, the extra cache misses in matrix with size 128*128 are conflict misses. This poses a problem for some blocking algorithms, which tends to use blocking size equal to the power of 2 for the benefits of scalability. Many capacity misses have been transformed into conflict misses instead of being eliminated. As can be seen in the figure, when the miss rate is high, miss elimination rate by TSP is also high. TSP can

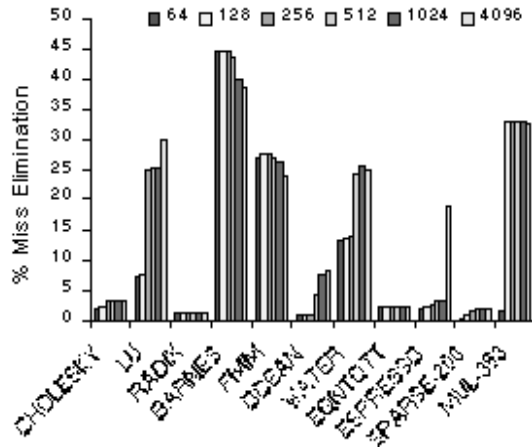


Figure 11: Miss Elimination Rate by TSP as MHT size is varied

help blocking algorithms by making the performance less sensitive to the partition of dataset. Blocking algorithms can help TSP as well. With blocking, large time-strides are transformed into small ones and more number of time-strides can be discovered by the MHT.

5.4 Results on complete benchmarks

Figure 11 shows the miss elimination rate by TSP across applications. The MPB size is 8. The MHT size is 64, 128, 256, 512, 1024 and 4096. The miss elimination rate varies widely, from as low as 3 percent in RADIX to as high as 45 percent in BARNES. The effectiveness of TSP depends on whether the access pattern in an application fits into our predicted miss pattern.

The existence of time-stride locality is evident in this figure. For many applications, the miss elimination rate increases sharply when the MHT size increases to a certain value. Then the rate keeps relatively unchanged with further increase of the MHT size. For example, in WATER, the miss elimination rate for MHT size of 64, 128 and 256 are almost same. There is a sharp rate increase as MHT size changes from 256 to 512. Then the miss elimination rate for MHT size of 512, 1024 and 4096 are almost same again.

In BARNES, FMM and WATER, miss elimination rate drops for some large MHT sizes. In these applications, a large number of time-strides can be discovered by the MHT and a large number of prefetch requests can be issued. On discovery of a time-stride, the window to schedule prefetch requests may be full. When the window is full, a new request will override a old request. Fewer misses will be eliminated if old prefetch addresses are more predictable than new prefetch addresses.

5.5 Comparison with a stride-prefetcher

To see how a TSP functions with other hardware prefetch mechanisms, we simulate a stride-prefetcher proposed in [2] and a TSP at the same time. The stride-prefetcher has access to the

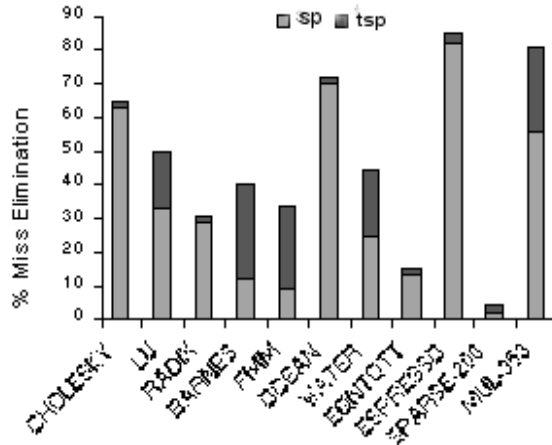


Figure 12: Miss Eliminate by Stride-prefetch and TSP

program counter. It has four streams and the depth of each stream is two. The MHT size of the TSP is 1024 and the MPB size is 8. On a cache miss, the stride-prefetcher is searched first. If the address can not be found in the stride-prefetcher, then TSP is searched.

As shown in Figure 11, when a TSP with MHT size of 1024 is used standalone, the TSP can eliminate 25, 40, 26, 25 and 32 percent of total misses for LU, BARNES, FMM, WATER and MUL-353 respectively. When a TSP is used with a stride-prefetcher, the TSP can still eliminate significant number of misses. As can be seen in Figure 12, 16, 28, 25, 19 and 25 percent of total misses for LU, BARNES, FMM, WATER and MUL-353 can be eliminated. A TSP can function in tandem with other hardware prefetch mechanisms to greatly improve cache performance.

As the miss elimination rate by a TSP decreases when a stride-prefetcher is used together with the TSP, we can see that they have shared coverage. Some consecutive miss addresses have constant spatial strides between them. Yet for each individual miss address, the consecutive time-strides have small differences. Thus cache misses by above addresses can be eliminated by both a stride-prefetcher and a TSP.

5.6 Comparison with a victim cache

The victim cache is another hardware cache assist used in conflict miss elimination. To compare the performance of victim cache and TSP, we simulate a victim cache and a TSP independently. The size of the victim cache is 16. For the TSP, the size of MHT is 512 and the size of MPB is 8.

In this experiment, the size of the victim cache is small as it is constrained by the cycle time of a processor. It is desirable that an address lookup in a victim cache can be done in one cycle so that the corresponding data can be transferred to the processor in the next cycle. If it takes more than one cycle for an address lookup, a victim cache will become another level in the memory hierarchy and the management of it will become complicated. In the case of a TSP, the size of MHT is constrained by the average time between two cache misses so that this size can be much

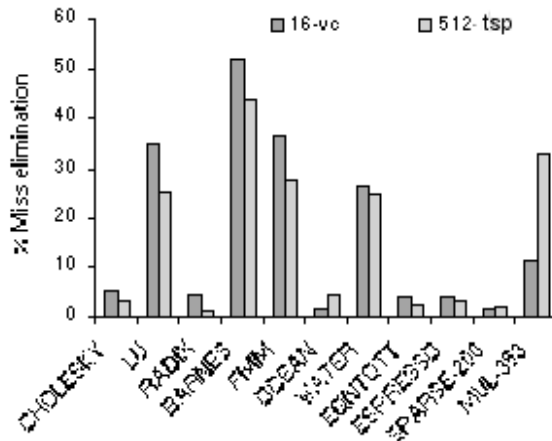


Figure 13: Miss Elimination Rate by Victim Cache and TSP

larger.

A victim cache also has higher implementation complexity than a hardware prefetcher. When a victim cache is used, following additional data paths are needed: read/write paths from a cache to a victim cache, a write path from a victim cache to a write buffer. These data paths may be on a critical path of the processor design. For a hardware prefetcher, the additional data path is a read path from a cache to a prefetch buffer, which is unlikely to be on a critical path of a processor design. And a prefetch buffer can be eliminated by prefetching data into a cache directly.

Figure 13 shows the miss elimination rate by a TSP and a victim cache. A TSP can eliminate more number of misses in 3 applications, OCEAN, SPARSE-200 and MUL-353. A victim cache can eliminate more number of misses in the rest 8 applications. But for applications with significant miss elimination rate, such as LU, BARNES, WATER and FMM, the miss elimination rate differences between the victim cache and the TSP are relatively small. In general, a victim cache and a TSP have comparable ability in conflict miss elimination.

6 Conclusion

In this paper, we have proposed a hardware prefetching mechanism to eliminate conflict misses. We have uncovered a new memory access pattern that makes some conflict misses predictable. Time-stride is used to describe the reoccurrence of conflict miss addresses and captures the characteristics of conflict misses. This prefetching mechanism is unique in that:

- We predict both which addresses to prefetch and when to prefetch. Other prefetchers only predict which addresses to prefetch and prefetch immediately .
- Which addresses to predict is based on the access pattern of an address itself instead of spatially/temporally correlated addresses.

A conflict miss can be eliminated by our mechanism while both addresses involved in the conflict miss are cacheable. This avoids the shortcomings of some conflict miss elimination techniques such as dynamic bypassing, which eliminate conflict misses by not caching one of the addresses involved in the conflict miss.

References

- [1] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Architectural Support for Programming Languages and Operating Systems-IV*, pages 40–52, 1991.
- [2] T.-F. Chen and J.-L. Baer. Effective hardware-based data prefetching for high performance processors. *IEEE Trans. on Computers*, 5(44):609–623, 1995.
- [3] K. I. Farkas and N. P. Jouppi. Complexity/performance tradeoffs with non-blocking loads. In *Int'l Symp. Computer Architecture*, pages 211–222, 1994.
- [4] J. Fu, J. Patel, and B. Janssens. Stride directed prefetching in scalar processors. In *Int'l Symp. Microarchitecture*, pages 102–110, 1992.
- [5] T. L. Johnson, M. C. Merten, and W. W. Hwu. Run-time adaptive cache hierarchy management via reference analysis. In *Int'l Symp. Computer Architecture*, pages 315–326, 1997.
- [6] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *Int'l Symp. Computer Architecture*, pages 252–263, 1997.
- [7] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Int'l Symp. Computer Architecture*, pages 364–373, 1990.
- [8] A. C. Klaiber and H. M. Levy. An architecture for software-controlled data prefetching. In *Int'l Symp. Computer Architecture*, pages 43–53, 1991.
- [9] M. S. Lam and E. E. Rothberg. The cache performance and optimizations of blocked algorithms. In *Architectural Support for Programming Languages and Operating Systems-IV*, pages 63–74, 1991.
- [10] S. McFarling. Cache replacement with dynamic exclusion. In *Int'l Symp. Computer Architecture*, pages 191–200, 1992.
- [11] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Architectural Support for Programming Languages and Operating Systems-V*, pages 62–73, 1992.
- [12] S. Palacharla and R. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Int'l Symp. Computer Architecture*, pages 24–33, 1994.
- [13] J. Veenstra and R. Fowler. MINT: A front end for efficient simulation of shared-memory multiprocessors. In *Int'l Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 201–207, 1994.