# A scheme of predictor based stream buffers

Bill Hodges, Guoqiang Pan, Lixin Su

April 24, 2002

## 1 Introduction

Cache misses, especially data cache misses are still one of the big factors degrading modern microprocessor performance. The degradation of performance caused by data cache misses can vary from a few to tens of percent depending on which application is run by the microprocessor.

Two major approaches are used to reduce data cache misses. The first approach involves redesigning the memory hierarchy by adding additional units or modifying a specific unit already existing in the memory hierarchy. This includes additional levels of cache like the L2 cache and even L3 cache, victim cache, pseudo cache, stream buffer, data value predictor, etc. Different microprocessors may choose to implement different type of units or a combination of several types of units. People can also improve cache performance by building a highly associative cache or a non-blocking cache. The works by reducing misses and results in increasd overall performance. The second approach usually doesnt deal with redesigning of the memory system. The microprocessor implements aggressive out-of-order issue and tries to explore instruction level parallelism. This hardware reordering is also called dynamic instruction scheduling. The compiler can also perform static instruction scheduling like boosting a load before a store to change the data access sequences. Instruction scheduling and data access order change can also help lower data cache misses.

The focus of our research is in trying to lower data cache misses and improve microprocessor performance by adding a special unit, esp. a specially designed stream buffer into the memory hierarchy. We believe that the data accesses of applications have their own patterns, and cache alone do not capture all the patterns appeared. By prefetching data into stream buffer based upon special predictions, the data cache misses can be lowered and the microprocessor performance can be increased.

Our hypothesis is that data accesses to memory hierarchy have certain patterns, like strides. There exist localized data accesses that may have frequently used data with smaller strides for consecutive memory accesses. There also exist the other group of non-localized data accesses. These global data accesses are prone to have large strides for consecutive data accesses. The reuse rate of fetched data with large strides are low. Besides strides, we believe there may exist other data access patterns that may need to be identified. Our work serves as a motivation for others to keep searching these existing but still not identified patterns.

In concept of stream buffers is introduced in 1990 by Jouppi[3], where a stream of data blocks is prefetched into a stream buffer by fetching the data consecutive to a miss address and shifted into the cache as used. The buffer handled further prefetches as it is emptying into the cache. However, the stream buffer proposed by Jouppi still can be improved in many aspects. In particular, large bodies of the work are targetted at avoiding agressively flushing the stream buffer. Both Palacharla and Kessler[2] and Farkas et al.[5] used stride length infomation and multiple buffers to achieve this. Recently, Sherwood et al.[1] implemented a new form of stream buffer called the Predictor-Directed Stream Buffer (PSB) as an extension to Farkas by predicting the next fetch address in a stream.

Another area of study is the mechanism of how the data in the stream buffer is kept. In usual implementations, when there is a hit in the stream buffer head entry, the data block inside this entry will be saved into the L1 cache. However, if it is streaming data that is only used once, it will cause cache pollution. A possible solution may need to add a filter between the data stream buffer and the L1 cache.

In this project we implement a stream buffer with decoupled predictor. The stream buffer is implemented as a side storage to L1 cache, while the predictors correspond to the pages active in the TLB. This allows a more complex predictor to be added without excessive overhead, and allows possible persistant predictor behaviour. The stream buffer is also seperate from the L1 cache to ease implementation.

## 2 Previous Work

There exist diverse prefetching strategies proposed by different researchers. Basically, the requirements of a good prefetching strategy include timing, which means the data prefetching has to be timely and the data will be available when needed, and accuracy, which means the prefetched data should be

useful instead of waste memory bandwidth and pollute data caches. All the prefetching strategies fall into two categories: static data prefetching and dynamic data prefetching. We only concentrate on dynamic data prefetching techniques.

Dynamic data prefetching is implemented by hardware without the assistance of compilers and preserves binary compatibility. It concentrates on discovering the dynamic execution behaviour of a program to do a better job then static prefetching in hope of dynamic behaviour can present more features then static analysis.

Jouppi implemented a stream buffer as a special storage unit consisting of several entries, with a tag, an available bit and a data line in each of the entry. Whenever a miss occurs, the stream buffer starts prefetching. The stream buffer prefetches the data consecutive to the miss address. The entries of the stream buffer models a FIFO queue. The first entry will be checked at a cache miss to see if it is the right data the operation needs. If there is a hit in that entry, the data in this entry will be removed and stored into L1 data cache and the data contained in the following entries will be sequentially shifted one entry ahead of their original positions, and the stream buffer filles itself by further prefetching.

From the prefetch block address calculation methods, dynamic prefetching can be divided into prefetching of sequential block, prefetching of a block with a fixed stride from missed address, and predictor-based prefetching. The predictor-based prefetching usually implements a predictor which predicts the next prefetching address based upon previous prefetching address predictions.

Providing stride information to the stream buffer can allow the microprocessor to know how many blocks it should skip before starting prefeching a new block into the stream buffer. The stride information can be obtained either according to the to-be-prefetched data address (Palacharla and Kessler) or according to the PC address of the instruction that requests a data prefetching into the stream buffer (Farkas et al.). Palacharla and Kesslers approach employs an algorithm called minimum-delta. Specifically, when a cache miss occurs, a stride calculator implemented along with the stream buffer will calculates the difference between the current data address and the previous prefetched data address. This difference is called stride which can either positive or negative and decides the prefetch direction. When the stride is less than the size of a block, a unit-stride is used to prefetch new data block into the stream buffer. Farkas et al. in contrast used a small fully associative buffer. Each entry of the buffer contains the previous miss address of a certain load and a tag to identify this certain load

instruction. They calculate a stride specifically for a load instruction and prefetch new data block into the stream buffer according to this calculated stride.

Sherwoods' stream buffer has a general next prefetch address predictor and per-stream history buffer implemented along with the stream buffer. When a cache miss occurs, the address predictor fetches the correspondent data streams previous prediction results from the per-stream history buffer. Then the predictor will give the next prefetch data address based on previous prediction information. After the prediction, the predictor saves the current prediction result into the per-stream history buffer for the next prefetch prediction. The stream is identified by instruction PC addresses.

## 3 Our Implementation

This section describes an architectural model for the stream buffer and prediction scheme. Figure 1 is a block diagram of our overall scheme. We implemented a decoupled predictor and stream buffer. We hypothesized that the behaviour of a stream can be easily gathered and used by observing the data accesses to a block of memory between the size of a line and a page. The size must be big enough to capture and efficiently reuse the stream infomation, while small enough to minimize interference between multiple streams. This leads itself to easily to implementing the preditor together with the TLB, and keeping track of the same segment of memory as the tlb[1]. The stream buffer is also seperate from the L1 cache and there is no communication between them besides checking whether a piece of data is already in the cache. When a miss is generated, the predictor corresponding to the miss is queried on whether the miss address should be fetched into the cache or stream buffer. Background prefetching is also applied in idle bus cycles. We have not implemented any cache prefetching schemes since they are orthogonal to our approach. For simulation purposes, the predictor table is implemented as a 2-level table similiar to the page table and the stream buffer is implemented in the cache. We check prefetches on every cycle, and a prefetch is issued to the stream buffer if the bus to L2 cache is free. (Of course, the stream buffer is allowed to ignore it as specified above.) Our current implementation do not prefetch to cache, which will pose new problems for filtering and replacement schemes.

[1] Sadly, this do not lead itself well to a superpage technique.

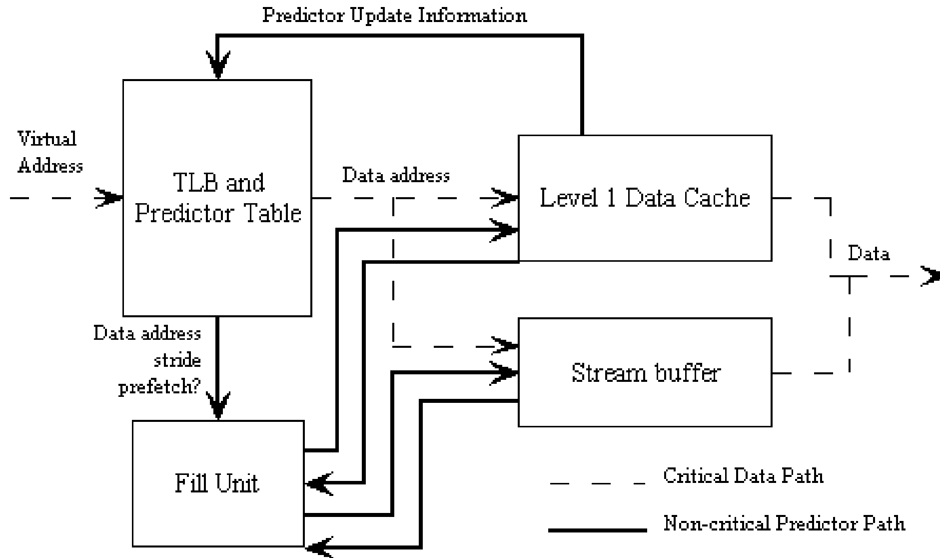


Figure 1: Block diagram

### 3.1 Predictor Tables

Our predictor is implemented as a state machine with four states as in Figure 2. The four states can further be classified into two types, namely predictable states and non-predictable states. The non-predictable states are UNINIT and RANDOM. When a predictor is first instantiated, it is set to the UNINIT (uninitialized) state. When the cache gives a predictor update, UNINIT always goes to the RANDOM state. RANDOM is the state in which a predictor sits if it exhibits no particularly predictable behavior, based on our prediction methods. On the other hand, STRIDE and SEMISTRIDE are predictable states and request prefetches is queried.

The predictor collects infomation as the memory associated to the preditor is accessed. On each memory access, the predictor entry associated with the address is looked up and updated. We also store the last address we had, and the cycle we last updated. On updates, a stride variable is calculated by subtracting the previous memory address accessed from memory address accessed. This current stride is compared with the stride of the previous two addresses. If the strides are exactly equal, the predictor enters the STRIDE state. If the strides are within a certain threshold, then the predictor enters the SEMISTRIDE state.

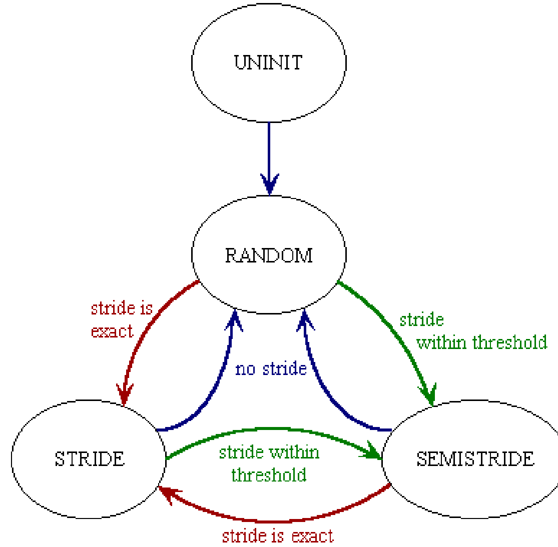


Figure 2: State Diagram for Predictor

If the predictor is in the STRIDE state, then it will return the predicted cycle until next line in the stream is accessed. To Compute this, the predictor requires a guess on the number of cycles between accesses. We use the average of the last two cache access updates. If the stride is greater than a certain threshold, then the predictor also specifies to use the stream buffer, rather than prefetching into cache. When the predictor is in the SEMISTRIDE state, behavior is almost exactly the same in these regards. The only difference in querying is that the SEMISTRIDE state adds an offset to the predicted number of cycles until the next line is needed. This gives a lower priority to SEMISTRIDE predictions compared to STRIDE predictions.

The SEMISTRIDE state allows the streamed data access without fixed stride to be prefetched to the extent where prefetching will be useful. When a predictor in the SEMISTRIDE state receives a cache update, if the new stride is different from the stored stride beyond the above mentioned allowable threshold, then the predictor is kicked into the RANDOM state again. If the new stride is the exact same as the previous stride, then the predictor jumps in the STRIDE state, giving it a higher priority. When a predictor in the STRIDE state receives a cache update, however, it is more prone

to stay in the STRIDE state. When two consecutive inexact strides occur, the predictor move into the SEMISTRIDE state, if the stride difference is within the threshold, or the RANDOM state if otherwise. This tendency to remain in the STRIDE state was motivated by a desire to keep good strides from needing to be rediscovered if we reach the horizontal bound of a two dimensional array, and wrap around, or in other such situations.

Because there are no data movement between stream buffer and cache, it is important to decide which data to keep in cache and which data to keep in stream buffers. We used the percieved stride length to make such a decision. Data that have unit stride can be left in cache, and those have longer stride can be left in the stream buffer, as a more tempory storage of streamed data.

### 3.2 Stream Buffer

We implemented our stream buffer as a set of fully associative units, each containing some number of cache-line sized entries. Because large fully associative lookups cannot be implemented efficiently, we limit ourselves to look at not too big ($< 16$) way associative lookups.

Each stream buffer unit consists of some number of tagged lines, just like a cache entry. The lines are organized in a circular fashion, but replacement is handled differently from the standard FIFO scheme. Instead of always evict the oldest line and replace it with a new line whenever a fetch is requested, we instead keep track of the most current line that have been accessed. Whenever the unit is selected for fetching (and thus replacement), the next line from that line is treated as the oldest line to be replacement. Another way to look at this is that newly fetched lines are considered to be old (in particular, oldest entry in the FIFO), until it is accessed, after which the line (and any line before it) is considered to be new lines.

This scheme has the advantage of limiting the ability of mis-predictions in prefetching to pollute the buffer, since unused lines will be evicted soon if needed. Of course, careful use of LRU can achieve the same effect, but the implementation cost will be higher.

Excessive unnecessary prefetches (overfetching) are the bane to any prefetch schemes. A standard approach to avoid overfetching is to use prefetch filtering. Overfetching is detrimental to performance because it both wastes bandwidth to the lower level and pollutes the buffer by fetching unnecessary stuff. Instead of a table based version, we implemented a cheaper version based on address locality. To each line, we associate a timestamp, which is updated when the line is used. When checking for prefetch

replacements, (unlike on a miss), the lines with a recent timestamp are not considered to be a candidate unless the address to be fetched is close to the tag address of the line. This achieves two effects. Firstly, lines that are currently being used (as in an active stream) will not likely be replaced with prefetch data not in the stream. Secondly, Data in the same stream is allowed to be prefetched. Of course, This scheme is not able to capture the more global prefetch filtering infomation that a table based method can, but it is able to be a quick and cheap filter for the clearly harmful fetches.

# 4 Simulation Results

## 4.1 Baseline Architecture

Our baseline architecture was a PISA (Portable ISA) based, 4-issue super scalar machine simulated using Simple Scalar 3.0. Table 1 show the baseline parameters used in our tests.

For our experiments, we used a L1 data cache size of 8 KB for our reference machine, while our modified simulator used only a 4 KB L1 Data Cache, assuming that our additions will take up at most the chip area or 4 KB of cache, and therefore testing our solution in an isocost environment.

## 4.2 Methodology

We ran portions of SPEC2000 benchmarks with reduced data sets using our modified simulator. We varied first the cache sizes on the baseline architecture from 4K Data L1 Cache to 8K Data L1, so we could see how we compare to an architecture simpler to develop, yet adding more hardware. Then we tested the scheme using the stream buffer and predictor table. The parameters of stream buffer stride threshold (the stride required to use the stream buffer over cache), SEMISTRIDE priority compared to STRIDE (how many cycles to offset the SEMISTRIDE prefetch request by), and SEMISTRIDE threshold (what the tolerable difference in stride is required to allow SEMISTRIDE state).

After choosing an optimal set of parameters, we ran a suit on three benchmarks on our system. We then went on to vary the SEMISTRIDE threshold, in order to classify the actual memory accesses based on strides. We also did limited tests on varying the size of the stream buffer to see impact on performance, if chip area and power became very sensitive issues for an implementation of our scheme. We then tested the baseline system with

| | |
|---|---|
| L1 Data Cache (all caches uses LRU eviction) | 8 (or 4) KB, 2 way set-assoc. |
| L1 Instruction Cache | 16 KB, direct-mapped |
| Unified L2 Cache | 256 KB, 4 way set-assoc. |
| L2 Cache latency | 6 cycles |
| Main Memory Latency | 18/2 |
| Memory bus width | 8 Bytes |
| Fetch, Decode, Issue, and Commit width | 4 |
| Branch mis-prediction latency | 3 |
| Bimodal branch predictor table size | 1048 |
| Return address stack size | 8 |
| 4-way associative, branch target buffer (sets) | 512 |
| Inorder | false |
| Register update unit size | 16 |
| Load Store queue size | 8 |
| Instruction TLB | 16:4096:4:l |
| Data TLB | 32:4096:4:l |
| Integer ALU's | 4 |
| Integer Multiplier/dividers | 1 |
| Memory ports | 2 |
| FP ALU's | 4 |
| FP Multiplier/dividers | 1 |

Table 1: Simple Scalar Configuration Used for Baseline Architecture

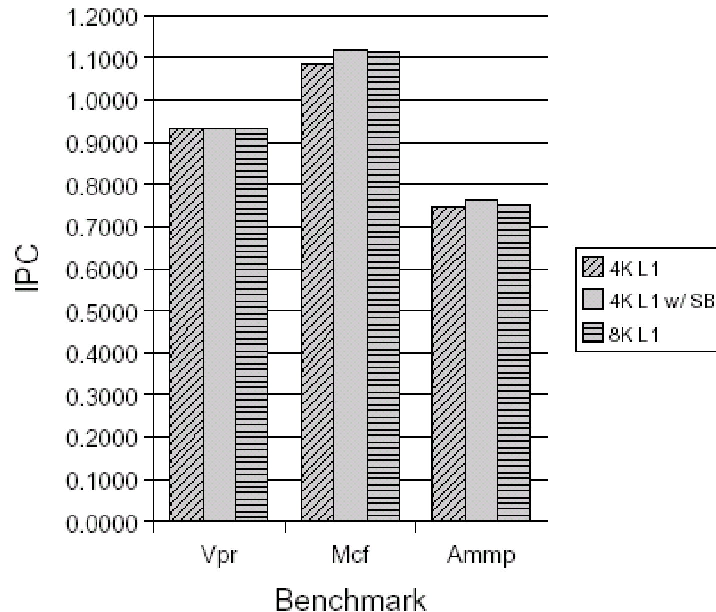


Figure 3: Performance Comparison (IPC)

a direct mapped cache, to compare our results with the effects of increases associativity.

### 4.3 Results

The following section gives the results of our tests, as specified by our methodology. The three benchmarks run were ammp, a program for doing scientific calculations, solving differential equations, mcf a program used for routing vehicles in mass transit systems, and vpr, an FPGA programming tool.

From Figure 3, we can see that we have got modest (between 1% and 5%) improvements, but not as big as we hoped for. Still, the prefetching done gives a comparable increase to doubling the size of the cache, while we only increased 10% of the storage. We will try to analyze the reason of the modest increase, and why certain applications works better then others.

Table 2 gives a distribution of the stride lengthes we have been requesting to prefetch and the corresponding prefetches done. We can see that the stride lengthes have a fairly wide distribution. Also, we are able to see

| BM | stride=8 | | 8<stride<32 | | stride=32 | |
|---|---|---|---|---|---|---|
| | Req. PF | PF Done | Req. PF | PF Done | Req. PF | FP Done |
| vpr | 4,196,590 | 49,124 | 1,956,374 | 142,838 | 4,920,951 | 255,372 |
| mcf | 16,460,868 | 7,530,754 | 9,239,329 | 41,531 | 2,504,918 | 1,756,205 |
| ammp | 9,248,365 | 28,294 | 438,057 | 2,718 | 3,339,122 | 2,589,741 |

Table 2: Prefetch Distribution

| Benchmark | Dec L1 Misses | Stream Hits | SB Prefetches | L1 Prefetches |
|---|---|---|---|---|
| vpr | 39,976 | 190,001 | 398,210 | 49,124 |
| mcf | 546,259 | 535,237 | 3,234,279 | 7,530,754 |
| ammp | 16,584 | 461,067 | 2,615,366 | 28,294 |

Table 3: Cache and Stream Buffer Performance

that the number of useful prefetch requests are not that high for vpr and ammp, whose IPC increase is small. We call the ratio between prefetches done and prefetches requested the predictor efficiency. We can see that for applications whose predictor efficiency is low, there is little chance of getting significient performance increase.

In contrast to the predictor efficiency above, we take a look at the prefetch efficiency in Table 3. Performance from prefetching comes from the decreases of misses on memory accesses. We can see for different applications, the ratio of decreased misses to number of prefetches are reasonablty consistant. But the increase do not all come from the hits into the stream buffer, as the data shown. This shows that the stream buffer is just taking data that would hit in the cache anyway in a lot of the cases. Another thing we can note from the above two tables is that splitting the prefetching load fairly between cache and stream buffer is hard, since a split based on the length of the stride is unlikely to be fair for all applications. This indicates adding data feeding pathes between the stream buffer and cache may be good.

## 5 Conclusion

We have implemented the decoupled stream buffer and predictor in simplescalar and it produced modest results. We have reached the following conclusions, which verify most of our hypothesis, leaving some room for future work:

1. Streams exist in applications, and have widely distributed strides.

2. Prefetching strides is helpful for performance.
3. Decoupled predictors are not very efficient since they issue more redundent prefetches for things already in the cache.
4. More careful choice of items to fetch into stream buffers should be done to filter out fetches friendly to the cache.
5. Prefetching stream increase performance at a reasonable cost.

# References


[1] T. Sherwood, S. Sair and B. Calder Predictor-Directed Stream Buffers. In Proceedings of the 33rd Annual International Symposium on Microarchitecture, December 2000

[2] S. Palacharla and R. Kessler. Evaluating stream buffers as secondary cache replacement. In 21st Annual International Symposium on computer Architecture, April 1994.

[3] N. Jouppi. Imprving direct-mapped cache performance by the addition of a small fully associative cache and prefetch buffers. In Proceedings of the 17th Annual International Symposium on Computer Architecture, May 1990

[4] T. Alexander and G. Kedem. Distributed prefetch-buffer/cache design for high performance memory systems. In Proceedings of the Second International Symposium on High-Performance Computer Architecture, February 1996.

[5] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. Memory-system Design Considerations for Dynamically-scheduled Processors. In Proceedings of International Symposium on Computer Architecture, 1997.

[6] S. P. Wiel and D. J. Lilja. When Caches Aren't Enough: Data Prefetching Techniques. In IEEE Computers, 1997

[7] T. F. Chen and J.-L. Baer. Effective Hardware-Based Data Prefetching for High Performance Processors. IEEE Trans. Computers, May 1995.

[8] N. Oren. A Survey of prefetching techniques. Technical Report TR-Wits-CS-2000-10, July 2000

[9] M. Bekerman et al. Correlated load-address predictors. In 26th Annual International Symposium on Computer Architecture, May 1999

[10] D. Joseph and D. grunwald. Prefetching using markov predictors. In 24th Annual International Symposium on Computer Architecture, Jun 1997

[11] G. P. Jones and N. P. Topham. A comparison of data prefetching on an access decoupled and superscalar machine. In 30th International Symposium on Microarchitecture, December 1997

[12] Y. Sazeides and J. E. Smith. The predictability of data values. In 305h International Symposium on Microarchitecture, December 1997