Predictor-Directed Stream Buffers

Timothy Sherwood Suleyman Sair Brad Calder

Department of Computer Science and Engineering
University of California, San Diego
{sherwood,ssair,calder}@cs.ucsd.edu

Abstract

An effective method for reducing the effect of load latency in modern processors is data prefetching. One form of data prefetching, stream buffers, has been shown to be particularly effective due to its' ability to detect data streams and run ahead of them, prefetching as it goes. Unfortunately, in the past, the applicability of streaming was limited to stride intensive code.

In this paper we propose Predictor-Directed Stream Buffers *(PSB), a scheme in which the stream buffer follows an address prediction stream instead of a fixed stride. In addition, we examine using confidence techniques to guide the allocation and prioritization of stream buffers and their prefetch requests. Our results show for pointer-based applications that PSB provides a 30% speedup on average over no prefetching, and provides an average 10% speedup over using previously proposed stride-based stream buffers for pointer-intensive applications.*

1 Introduction

A great deal of effort has been invested in reducing the impact of cache misses on program performance. As with any other latency, cache miss latency can be tolerated using compile-time techniques such as instruction scheduling, or run-time techniques including out-of-order issue, decoupled execution, or non-blocking loads. It is also possible to reduce the latency of cache misses using multi-level caches, victim caches, and prefetching.

Several approaches have been proposed for prefetching data to reduce or eliminate load latency. These range from inserting compiler-based prefetches to pure hardware-based data prefetching. Compiler-based prefetching annotates load instructions or inserts explicit prefetch instructions to bring data into the cache before it is needed to hide the load latency. They use locality analysis to insert prefetch instructions, showing significant improvements [21]. Hardware-based prefetching can dynamically predict prefetch address streams and predict prefetch addresses that may be hard to find using compiler analysis. Compiler and hardware-based prefetching can be used together, since the compiler can be used to prefetch load instructions for which it can accurately determine locality information, and the hardware prefetcher can be used for those load address patterns not captured. In this paper we focus on a new hardware-based prefetcher.

The focus of our research is improving the performance of data prefetching with stream buffers in the context of a realistic processor design. Stream buffers were originally proposed by Jouppi [19] to prefetch a stream of sequential cache blocks. When a cache miss occurs, the next sequential cache block is allocated into a stream buffer. The stream buffer then prefetches sequential cache blocks from that address, as bandwidth permits, until the buffer is full. As prefetches are used, new data is brought in, keeping the buffer far enough in advance of the data's use so that it can potentially hide the entire latency.

Palacharla and Kessler [22] extended stream buffers by associating a stride with each stream buffer. They examined providing a stride from a table which was indexed by the area of memory being accessed. Farkas et. al. [13] further extended this research by using a PC indexed stride table, which allows for detection of many strides over the same region of memory.

In this paper we propose a new form of stream buffer called the *Predictor-Directed Stream Buffer* (PSB). Instead of associating a fixed stride with each buffer, we use a *predictor* to generate the next address to prefetch. We simulate the use of a hybrid *Stride Filtered Markov* (SFM) predictor to direct stream buffer prefetching and find it is quite adept at finding both complex array access and pointer chasing behavior over a set of pointer intensive benchmarks.

Farkas et. al. [13] show the importance of using allocation filters to prevent the stream buffers from being allocated and deallocated too often and for too many streams, an effect we call *stream thrashing*. We propose a technique based on confidence for eliminating stream thrashing as well as making more effective use of available processor and predictor resources. This is done by using confidence to guide stream buffer allocation and prefetch prioritization.

The rest of the paper is organized as follows. Section 2 describes past address prediction work as it relates to PSBs. In section 3, prior hardware prefetching models are discussed. Section 4 describes our Predictor-Directed Stream Buffer architecture. Simulation methodology and benchmark descriptions can be found in Section 5. Section 6 presents results for our architecture, and our conclusions are summarized in section 7.

2 Address Prediction

To guide hardware-based prefetching, accurate address prediction is needed. In performing this research, we examined using stride-based address prediction, Markov/context address prediction, and correlated address prediction.

2.1 Stride

A *stride* predictor [8, 12] keeps track of not only the last address referenced by a load, but also the difference between the last address of the load and the address before that. This difference is called the stride. The predictor speculates that the new address seen by the load will be the sum of the last value seen and the stride. We chose to use the two-delta stride predictor [12, 28], which only replaces the predicted stride with a new stride if that new stride has been seen twice in a row.

2.2 Context/Markov Predictor

Context [28, 29, 32] and *Markov* [6, 7, 18] predictors are fundamentally similar, in that each predictor bases its prediction on the last values seen. An order k context/Markov predictor uses the k past values to predict the next one. It can only provide a prediction, if the given pattern has been seen and the transition is recorded into a prediction table.

A Markov predictor assumes that the address stream seen in a program can be efficiently modeled by a Markov model. A Markov model is a set of states and transition frequencies where each state has a probability of transition to another. Each transition from address *A* to *B* is assigned a weight representing the fraction of *A*s that are followed by a *B*. The Markov predictor described in [18] is a first order context predictor as it uses only the last address to predict the next one.

Bekerman et. al. [2] propose yet another context-based predictor. For every load, they combine a series of past base addresses (they state that 4 is enough for reasonable accuracy), to generate a history and store it into a first-level table. They use that history as an index into a second level table that stores a predicted *base* address. They then add the load's static offset (which could be stored in the first-level table) with the predicted base address. By using base addresses, a high-level of global correlation is achieved for multiple load instructions accessing different fields in the same object.

In this paper, we only provide results for stride and first order Markov-based prediction. We simulated higher order Markov predictors and the correlation predictor [2], but saw little to no improvement in prediction accuracy and coverage over first order Markov predictor for the programs we examined. This is partially due to the fact that correlated loads lie within the same cache block for the programs we examined. Therefore, correctly predicting the correlated load provides less gains in terms of prefetching, since we perform our predictions and prefetches at the cache block granularity.

3 Hardware Prefetching Models

We classify the prior hardware prefetching research into three models – Fetch Stream Prefetching, Demand-Based Prefetching, and Decoupled Prefetching.

3.1 Fetch Stream Prefetching

The first model follows the branch prediction or fetch stream, predicting and prefetching addresses [9, 16, 10, 4].

Chen and Baer [9] proposed an approach to provide the load prediction early by using a Look-Ahead PC, which can run ahead of the normal instruction fetch engine. The LA-PC is guided by a branch prediction architecture that runs ahead of the fetch engine, and is used to index into an address prediction table to predict data addresses for cache prefetching. Since the LA-PC provided the instruction address stream ahead of the normal fetch engine, they were able to initiate data cache prefetches farther in advanced than if they had used the normal PC, which in turn allowed more of the data cache miss penalty to be masked. The amount of load latency that can be hidden is dependent upon how far the look-ahead PC can get in front of the execution stream.

Reinman et.al. [23] extended the approach of Chen and Baer [9] to instruction prefetching. In their approach, they only have one branch predictor instead of two as in Chen and Baer. This is accomplished by decoupling the branch predictor from the instruction cache with a fetch target queue between them. The queue is used to store fetch block predictions, which are then fed into the instruction cache in a later cycle. The fetch addresses in the queue are used to perform instruction cache prefetching. They recently extended this approach to perform power-efficient instruction prefetching by decoupling the tag component of the instruction cache access from the data component of the cache access [24]. The tag component verifies if an address is in the cache in a separate cycle before the data component access for the instruction lookup. If the fetch address is not found, it is prefetched, while the fetch address is queued up to be consumed by the data component. In this new design, the data component access consumes significantly less power, since only one way of the data component is driven, and the way was determined during the tag access in a prior cycle. They are currently extending this design to fetch stream data cache prefetching.

3.2 Demand-Based Prefetching

The second model can be classified as demand-based prefetching. In this approach an action such as a cache miss

or the use of a cache block has to occur for a prefetch to be generated.

An early example of a demand-based prefetching architecture is *Next Line Prefetching* (NLP) by Smith [31], where each cache block was tagged with a bit indicating when the next block should be prefetched. When a block is prefetched its tag bit is set to zero. When the block is accessed during a fetch and the bit is zero, a prefetch of the next sequential block is triggered and the bit is set to one.

Another demand-based prefetching architecture is Shadow Directory Prefetching (SDP) by Charney and Puzak [6]. In SDP, each L2 cache block has a shadow address associated with it. The shadow address points to the cache block accessed right after the corresponding cache block, providing a simple Markov transition. A hit in the L2 cache with a useful shadow entry triggers a prefetch of the shadow address. Alexander and Kedem [1] examined using a similar Markov table, but distributed over the DRAM modules, which are used to prefetch cache blocks from DRAM array into an SRAM buffer.

The last example we will discuss is the Markov prefetcher used by Joseph and Grunwald [18]. When a cache miss occurred, the miss address would index into their Markov prediction table to provide the next set of possible cache addresses that have followed this miss address before. After these addresses are prefetched, the prefetcher stays idle until the next cache miss. They do not use the predicted addresses to re-index into the table to generate more predictions for prefetching.

In order to minimize the load on the bus, prefetch bandwidth is limited by employing *accuracy based adaptivity* [18]. In this scheme, two-bit saturation counters are added to each prediction address. The idea is to remove prefetches that have exhibited poor behavior in the past. When a prefetch is discarded from the prefetch buffer without being used, the corresponding counter is incremented. If the prefetched block is used, then the counter associated with the entry that made the prediction, is decremented. When the sign bit of the counter is set, the relevant entry in the prediction table is disabled. Prefetch requests from disabled entries are tracked so that they can be enabled when they start making correct predictions.

3.3 Decoupled/Stream Prefetching

In this model the prefetcher is loosely decoupled from the instruction fetch stream and can potentially prefetch down multiple predicted streams independent of what the instruction fetch stream is doing.

3.3.1 Decoupled Models

An access decoupled architecture partitions programs into a prefetching instruction stream and an execution instruction stream [15, 3, 17]. As long as the prefetch stream can run ahead of the execution stream, the memory latency can be masked. Roth et. al. [25, 26] has examined both a software and hardware approach for prefetching recursive data structures using a decoupled model. Yang and Lebeck [33] examined an architecture which uses the compiler to create small prefetch kernels of instructions, which are executed in parallel with the original application in a separate prefetch engine.

3.3.2 Stream Buffers

Jouppi introduced *stream buffers* to improve direct mapped cache performance [19]. The stream buffers follow multiple streams prefetching them in parallel and these streams can be completely decoupled from the instruction stream of the processor. They are designed as FIFO buffers that prefetch consecutive cache blocks, starting with the one that missed in the L1 cache. On subsequent misses, the head of the stream buffer is probed. If the reference hits, that block is transferred to the L1 cache.

Palacharla and Kessler [22] suggested two techniques to enhance the effectiveness of stream buffers : *allocation filters* and a *non-unit stride* detection mechanism. The filter prevents a stream buffer from being allocated until two consecutive misses occur for the same stream. Also presented by Palacharla and Kessler is a *minimum delta* non-unit detection scheme. With this scheme, the dynamic stride is determined by the minimum signed difference between the miss address and the past N miss addresses. If this minimum delta is smaller then the L1 block size, then the stride is set to the cache block size with the sign of the minimum delta. Otherwise, the stride is set to the minimum delta.

To implement the non-unit stride detection an address indexed stride table is used. To find the striding behavior the memory is divided up into chunks, and associated with each chunk is a stride. While this approach is quite effective at finding strides, we found that it was uniformly outperformed by the per-load stride detector of Farkas et. al. [13]. Therefore, we only present comparison results of our approach with the PC-based stride prediction stream buffers.

Farkas et. al. [13] made an important contribution by extending this model to use a *PC-based* stride predictor to provide the stride on stream buffer allocation. The PC-stride predictor determines the stride for a load instruction by using the PC to index into a stride address prediction table. This differs from the minimum-delta scheme, since the minimum-delta uses the global history to calculate the stride for a given load. PC-stride predictor uses an associative buffer to record the last miss address for N load instructions, along with their program counter values. Thus, the stride prediction for a stream buffer is based only on the past memory behavior of the load for which the stream buffer was allocated.

Farkas et. al. [14] further enhanced the stream buffer design of Palacharla and Kessler by enforcing the streams being followed by multiple stream buffers to be non-overlapping. This prevented duplication and saved bus bandwidth. Furthermore, instead of the FIFO structure which had been originally proposed by Jouppi, they proposed the use of a fully-associative stream buffer lookup, which we model.

4 Predictor-Directed Stream Buffers

We will now describe our Predictor-directed Stream Buffer (PSB) architecture. The PSB architecture resides on chip and prefetches data from the L2 cache and main memory into the stream buffers. If a prefetch request is not found in the L2, it will service the request from main memory. We concentrate on stream buffers instead of the many other architectures described in the previous section because of their simple yet effective design, their ability to follow a prefetch stream independent of the fetch stream, and the design fits nicely with an on-chip prefetcher to try to hide L2 and main memory latency.

We present an approach that extends the PC indexed stream buffer design of Farkas et. al. [13]. As described in Section 3, the PC index scheme uses a stream buffer which is guided by a static stride, provided at allocation time by a per-PC stride table as shown in Figure 1. This approach can work well for stride-based applications, but the stream buffers do not follow the correct stream for non-stride based load patterns, such as during the traversal of a recursive data structure.

To address this problem, we propose Predictor-Directed Stream Buffers (PSB) as shown in Figure 2. The general idea of a PSB is to use a predictor to generate an address stream for prefetching. The predictor takes as input some prediction information, such as the last address accessed and history information, and then generates a prediction for a given stream buffer. This prediction is then stored back into the stream buffer, and the prediction information in the stream buffer is updated. In this way we can generate prediction n from prediction $n-1$. The base of the recursion is a cache miss which causes a stream buffer allocation.

There are two major parts to PSBs, a per-stream history which is stored with each stream buffer, and a stateless address predictor which is shared between stream buffers. The per-stream history is used to keep data about a particular stream buffer and may be used for a variety of purposes, such as indexing into the address predictor, confidence information, and local stride. The primary service of the per-stream history is to store a current or speculative state which can be fed to the predictor. The prediction from the address prediction table is then used to update the state information in the stream buffer so that a new speculative prediction can be made. It is a key point that the address prediction table is

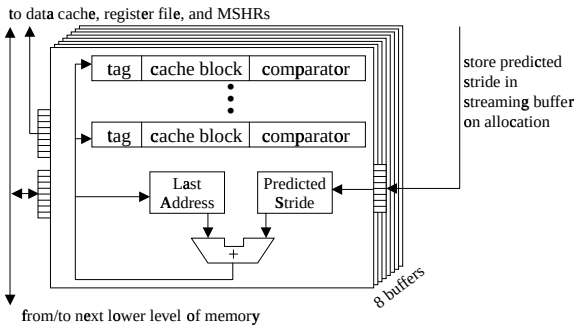

Figure 1: Stride-based Stream Buffer Architecture. Eight stream buffers are shown (overlapping each other). Each stream buffer can hold N cache blocks. When a stream buffer is allocated, it is assigned a predicted stride to use to generate all of its prefetch addresses.

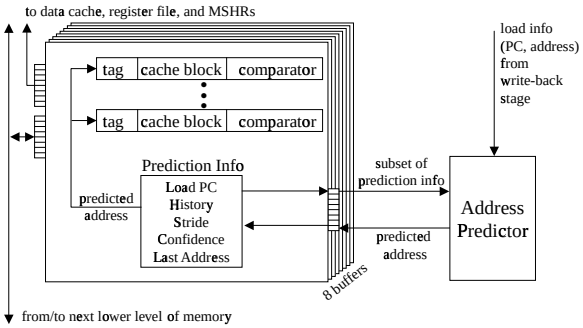

Figure 2: A Predictor-Directed Stream Buffer. We modify the stream buffer so it accesses a separate address prediction table to get its next prefetch address.

not updated when the stream buffer makes a prediction, this step is done separately in the write-back stage when a load has a data cache miss.

This model allows the stream buffer to follow the address prediction stream of any address predictor, whose predictions are more accurate than those of a fixed-stride predictor.

4.1 Predictor-Directed Stream Buffer Implementation

Figure 2 shows the general model of our predictor-directed stream buffer architecture. Each stream buffer holds (1) the PC of the load that caused the stream buffer to be allocated, (2) the last predicted address for the load, and (3) any additional prediction information (e.g., history state or confidence) needed to perform the next address prediction. The stream buffer is on-chip next to the address predictor, which

in our case is a stride-filtered Markov predictor.

There are several stages of execution a stream buffer will go through over the course of a program, starting with the allocation of a stream and ending with it's reallocation. We now show the initialization, steady state operation of, and termination of a stream in a stream buffer.

Allocation A stream buffer is allocated, subject to allocation filters (see section 4.3), when a load executes and it misses both in the data cache and the stream buffer. When a load is given a stream buffer, it copies its PC, current address, and any additional prediction information to the stream buffer from the address predictor. This initialization stage is only done once per allocation, and is directed only from predictor to stream buffer, the state of the address predictor is not modified. This copied state will later be used for indexing into the prediction table.

Prediction Each cycle, one stream buffer is chosen to make a prediction using the address predictor, according to priority heuristics described in section 4.4. The information stored in the stream buffer is used to index into the address predictor, returning the next predicted address, and potentially updating the stream buffer's history information. We properly model allowing only a single prediction per cycle to be generated from the predictor. Due to the fact that only one request (miss or prefetch) can be processed by the bus from the L1 to the L2 cache at a time, the predictor was not a bottleneck even with the one prediction per cycle limitation.

Once a stream buffer has been allocated, the stream buffer's history information is updated after each prediction. The address prediction table, as was mentioned earlier, remains unchanged while generating a prediction for a stream buffer. For example, a design such as a context predictor which uses a history of the last N addresses to index into the address predictor would store the history of its last N predictions in the stream buffer, and use this as an index into the address predictor each cycle. The history of the last N addresses stored in the stream buffer is updated after a prediction, *not* the state in the address prediction table. Therefore, the stream buffer maintains its own prediction history information.

Before inserting the prediction into the stream buffer, the stream buffers are searched in parallel for the cache block of the predicted address. This was used by Farkas et. al., [13] to prevent stream buffers from prefetching down overlapping paths. If the prediction is found to be already resident in a buffer entry then the prediction is ignored, no useful prediction is made that cycle, and the stream buffer prediction history information is updated. If prediction is not found in the stream buffer, the prediction is stored in the stream buffer's least recently used entry, and that entry is marked as ready for prefetching. Once all entries have been predicted for a stream buffer, no further entries will be predicted until (1) an entry is cleared during a lookup (it is a hit), or (2) the stream buffer is reallocated.

Prefetching Once an entry has a valid prediction associated with it, it is ready to be prefetched. We only allow prefetches to occur if the L1-L2 bus is free at the start of any given cycle. When the bus is free, a stream buffer with an entry containing a valid un-prefetched prediction is chosen using the priority scheduling algorithms described in section 4.4. The prefetch is then sent to the lower levels of memory and the entry is marked as prefetched and waiting.

Lookup When a load performs a lookup in the L1 data cache, it searches all of the stream buffer entries in parallel for a hit. For our results, we assume the data cache lookup latency is the same as the stream buffer lookup latency. If there is a hit in the stream buffer, and the data is not yet ready in the data cache, the cache block stored in the stream buffer is moved into the data cache. If there is a tag hit in the stream buffer, but the block is not ready in the stream buffer, the tag is moved into a data cache MSHR, and the data cache handles the block when it comes back from memory. For a stream buffer hit, the corresponding stream buffer entry is freed for a new prediction and prefetch.

We will now describe our design using a Stride-Filtered Markov (SFM) address predictor, although any address predictor [2, 18, 28, 29, 32] can be used to guide the predictor-directed stream buffer. We examined several types of predictors (including stride with correlated [2]), but only provide results for a SFM table, as it performed uniformly better.

4.2 Stride-Filtered Markov Predictor

Charney and Reeves [7] and also Joseph and Grunwald [18] introduced *Markov* prefetching, and provided results for a "stride and Markov in series" predictor. We use this predictor to guide our predictor-directed stream buffer, and make a few minor improvements which are described below.

To provide address prediction for the stream buffers we use a *Stride-Filtered Markov* (SFM) predictor. The predictor has a two-delta stride table in front of a Markov prediction table, as shown in Figure 3. In the write-back stage, the load instruction is checked to see if it hit or missed in the L1 data cache. The prediction table is only updated on a miss (i.e. we are predicting the miss stream). In addition, our implementation does not store loads that receive their value forwarded from stores in the prediction table, since we found little benefit from prefetching these loads.

In the write-back stage, the load-PC (for a missed load) is used to index into the stride table. The stride table stores (1) the last address for the load, (2) the last stride for the load, (3) the 2-delta stride, and (4) some confidence information. If the stride calculated by (current miss address - last address) does not match the last stride or 2-delta stride, then the Markov table is updated noting the transition from last address to current address. The last address is stored as

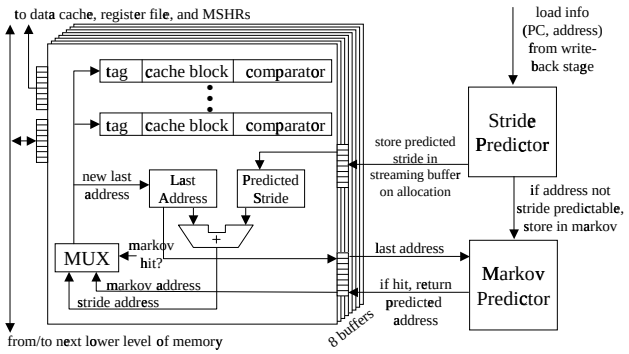

Figure 3: Stride-Filtered Markov Predictor-Directed Stream Buffer Architecture. When a stream buffer is allocated it is assigned a fixed stride from the stride-pc table. To generate the next prefetch address the last address is (1) looked up in the Markov table, and (2) used to calculate a next stride address. If the Markov table hits, then the Markov address is used, otherwise the next stride address is used for the prefetch.

the tag, and the current address as the data entry. Accordingly, when that same last address is seen again, it will get a hit in the Markov table, predicting the next miss address not captured by the stride predictor.

For the SFM predictor examined in this paper, we do not use any history to index into the Markov part of the table, in other words we present results from a first order Markov predictor. We examined using higher order Markov predictors as in [18], but found that it provided little improvement, confirming their results. The only additional information we copy into the stream buffer from the predictor is some confidence information, to guide priority scheduling described below.

In order to reduce the size of the Markov predictor table we store into the table only the difference between consecutive cache miss addresses, rather than the absolute address as is done in prior work. Of course this number can be further reduced by storing this difference as the number of cache blocks rather than at a byte granularity. To calculate the address to prefetch, a stream buffer adds its last missing address to the signed offset contained in the table. The table is still indexed by the last miss as in the standard Markov table. Figure 4 shows how many bits are needed to represent the address difference for all of the miss transitions found in the Markov table. The results show that having 16 bits captures almost all of the transitions. This number could perhaps be further reduced by smart heap memory allocation which could place objects with high temporal locality close to one another. In this paper we use a Markov table with 2K entries, which uses a total of 4Kbytes for the data storage. In addition, the tag size can also be reduced by storing only partial address tags.

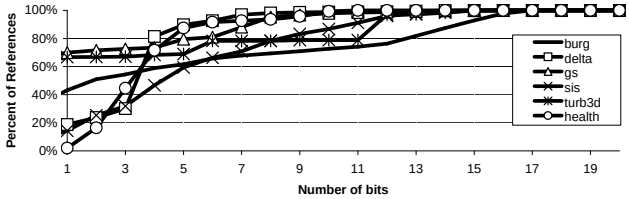

Figure 4: The number of bits to accurately predict cache misses using the Markov Difference Predictor. The y-axis shows the percent of L1 cache misses that could be correctly predicted given the number of bits used for each entry of the markov table shown on the x-axis. The cache miss address is predicted by adding together the address used to index the Markov table with the value stored in the Markov table.

4.3 Allocation Filtering

Stream buffer allocation is one of the most important parts of a stream buffer architecture. Since there are only a small number of stream buffers, there is high contention, as every data cache miss could potentially allocate a stream buffer.

Farkas et. al. [13] showed that using *two miss stride filtering* provided good results for a PC-based stream buffer. Two miss filtering only allocates a stream buffer for a load when it misses 2 times in a row, and the last two strides are identical. For our predictor-directed stream buffers we examine two methods for filtering allocation – a general form of two miss filtering, and using our new prediction confidence to guide allocation.

When updating the SFM predictor for a load that misses in the cache, both the PC-based stride table and the address based Markov table are indexed, and potentially updated. Our two-miss allocation filter allows a load to allocate a stream buffer when the load has two cache misses in a row, and both times the load would have been correctly predicted using the stride predictor or the Markov predictor. If this occurs, then it allocates a stream buffer. This modified scheme is our two-miss allocation filter.

The second heuristic we examine uses address prediction confidence to guide stream buffer allocation. Each entry in the PC-based table stores an *accuracy counter*, which is incremented every time the load's update address matches the prediction of the stride or Markov table, and decremented when it does not match. The saturating counter reflects the ability of the predictor being able to predict the load's misses. By separating the confidence counters from the stream buffer we can gauge how well a particular load is performing before we allocate a stream buffer to it. In this way we can avoid stream thrashing. When a stream buffer is allocated it copies the accuracy confidence counter into a *priority counter* in the stream buffer. Maintaining the priority counter is described in more detail in the next section.

On a cache miss, the accuracy confidence counter in the

prediction table is used to guide stream buffer allocation. If the address prediction confidence level of the load is above an allocation threshold, it is allowed to contend for a stream buffer. Our results suggest that a threshold value of 1 is appropriate for our benchmark suite. In addition, a load is only allocated a stream buffer if there is at least one stream buffer whose *priority* confidence counter is less or equal to the *accuracy* confidence counter of the load. If the load's accuracy confidence is lower than all of the stream buffers priority confidence, then a stream buffer will not be allocated for it.

4.4 Stream Buffer Priority

The predictor and bus create a resource constraint, since there are potentially several stream buffers which have empty entries, or have predicted addresses waiting to be prefetched. We examine two approaches for determining which stream buffer should get access to the predictor and L1-L2 bus each cycle.

The first heuristic is *Round-Robin* giving each buffer an equal chance at performing a prediction or prefetch. A pointer is kept to the last stream buffer to perform a prediction and another pointer for the last entry to issue a prefetch. The stream buffers are then sequentially examined in round-robin order, looking for a buffer with an entry in need of prediction or a predicted entry ready to be prefetched.

The second heuristic uses *Priority Counters* to guide which stream buffer gets to perform the next prediction or prefetch. Every time there is a lookup and the stream buffer gets a hit, the priority counter is incremented by a constant value (2 in our implementation). To enable the reuse of stream buffers that had high confidence but outlived their usefulness, after several allocation requests (i.e. data cache misses that also miss in stream buffers) we decrement each stream buffer's priority counter by a value of 1. We found using 10 L1 data cache misses as our aging period provided decent results. When determining which stream buffer gets to use the predictor or perform a prefetch, the stream buffers are examined in the order from highest priority to lowest. If there are several stream buffers that are at the same confidence level, we use an LRU policy to choose the winner. As described in section 4.3 when a stream buffer is allocated, the accuracy confidence is copied into the stream buffer's priority counter. This cuts down the contention time of load that has proven to be predictable.

In addition, as also described in the prior section, the priority counter is used to guide stream buffer allocation when using accuracy confidence to guide allocation. A stream buffer will only be re-allocated for a data cache miss if the load's prediction accuracy confidence is greater than or equal to a stream buffer's priority counter. Therefore, stream buffers that are performing useful prefetches will stay allocated and have a longer lifetime.

Program	Description
health	A hierarchical health-care system simulator taken from the Olden Benchmark suite (input: 3 500).
burg	A program that generates a fast tree parser using BURS technology. It is commonly used to construct optimal instruction selectors for use in compiler code generation. The input used was a grammar that scribes the VAX instruction architecture.
deltablue	A constraint solution system which is implemented in C++, with an abundance of short lived heap objects.
gs	Ghostscript is an implementation of Adobe Systems' PostScript (tm) language. The input run converts a PostScript file into a jpeg.
sis	Synthesis of synchronous and asynchronous circuits (input: simplify). It includes a number of capabilities such as state minimization and optimization. The program has approximately 172,000 lines of source code and a good deal of pointer arithmetic.
turb3d	Simulates isotropic, homogeneous turbulence in a cube with periodic boundary conditions in x,y,z coordinate directions (input: ref).

Table 1: Description of benchmarks used.

4.5 TLB Translation and Prefetching

As we store the virtual effective address of a load in our predictor, we need to translate this to a physical address before we access memory. On a prefetch, we access the data TLB for the translation and perform a replacement if necessary. In essence, this amounts to TLB prefetching [27]. However, we did not observe any benefits or performance losses caused by this approach, as the benchmarks we have used had only a small number TLB misses. The TLB translations could potentially be stored with each stream buffer when the stream buffer is allocated. Then a TLB lookup would only need to be performed when the next virtual prefetch address goes outside the current page boundary.

5 Methodology

The simulator used in this study was derived from the SimpleScalar/Alpha 3.0 tool set [5], a suite of functional and timing simulation tools for the Alpha AXP ISA. The timing simulator executes only user-level instructions, performing a detailed timing simulation of an aggressive 8-way dynamically scheduled microprocessor with two levels of instruction and data cache memory. Simulation is execution-driven, including execution down any speculative path until the detection of a fault, TLB miss, or branch mis-prediction.

To perform our evaluation, we collected results for the programs shown in Table 1. The programs were compiled on a DEC Alpha AXP-21164 processor using the DEC FORTRAN, C and C++ compilers under OSF/1 V4.0 operating system using full compiler optimization (`-O4 -ifo`). Table 2 shows the number of instructions simulated, L1 data cache miss rate, percent of executed in-

program	#inst (Mill)	%L1 MR	%lds	%sts	IPC	L1-L2 %bus	L2-M %bus
health	33	26.5	36.0	14.2	0.62	38.5	0.5
burg	300	6.5	19.1	18.7	1.91	19.5	4.8
deltablue	96	16.7	28.9	9.9	1.22	39.3	4.1
gs	300	2.0	19.2	6.8	3.5	6.8	0.9
sis	300	3.7	28.7	12.8	1.94	12.2	0.9
turb3d	300	6.5	23.3	16.2	2.54	26.2	13.7

Table 2: Baseline results showing the number of instructions simulated, L1 data cache miss rate, percent of executed instructions that were loads and stores, the IPC for each program, and the percent of cycles the bus was busy from the L1 to L2, and the bus from the L2 to main memory were busy.

structions that were loads and stores, the IPC for each program, and the percent of cycles the bus from the L1 to L2, and the bus from the L2 to main memory were busy (occupied). `Turb3d` was fast forwarded 1.3 billion instructions [30] before gathering statistics.

5.1 Baseline Architecture

Our baseline simulation configuration models a next generation out-of-order processor microarchitecture. We've selected the parameters to capture underlying trends in microarchitecture design. The processor has a large window of execution; it can fetch up to 8 instructions per cycle. It has a 128 entry re-order buffer with a 64 entry load/store buffer. To compensate for the added complexity of disambiguating loads and stores in a large execution window, we increased the store forward latency to 2 cycles.

To make sure that the prefetching speedups we report are from actual prefetching benefit and not from compensating for a conservative memory disambiguation policy, we implemented perfect store sets [11]. Perfect store sets cause loads to only be dependent on stores which write to the same memory, i.e when they are actually dependent instructions. In this way loads will not be held up by false dependencies making the prefetcher look better. The performance difference between the two schemes is explored in section 6.

In the baseline architecture, there is an 8 cycle minimum branch mis-prediction penalty. The processor has 8 integer ALU units, 4-load/store units, 2-FP adders, 2-integer MULT/DIV, and 2-FP MULT/DIV. The latencies are: ALU 1 cycle, MULT 3 cycles, Integer DIV 12 cycles, FP Adder 2 cycles, FP Mult 4 cycles, and FP DIV 12 cycles. All functional units, except the divide units, are fully pipelined allowing a new instruction to initiate execution each cycle. We use a McFarling gshare predictor [20] to drive our fetch unit. Two predictions can be made per cycle with up to 8 instructions fetched.

We rewrote the memory hierarchy in SimpleScalar to better model bus occupancy, bandwidth, and pipelining of the second level cache and main memory. For the majority of our results, the L1 instruction cache is a 32K 2-way associative cache with 32-byte lines. The baseline results are run with a 32k 4-way associative data cache with 32-byte lines. A 1 Megabyte unified L2 cache is simulated with 64-byte lines. The L2 cache has a latency of 12 cycles, and is pipelined three accesses deep. The main memory has an access time of 120 cycles. The L1 to L2 bus can support up to 8 bytes per processor cycle whereas the L2 to memory bus can support 4 bytes per cycle.

6 Prefetching Performance

This section compares predictor-directed stream buffers to the best performing prior stream buffer approach. This is the pc-based stride stream buffers of Farkas et. al. [13], which was described in section 3. We call their approach *PC-Stride*, where data cache missed loads are kept track of in a 256 entry 4-way associative stride address prediction table. On a miss, the predicted stride is copied into the stream buffer to guide the predictions. We examined using PC stride tables larger than 256 entry, but they provided little to no improvement.

For our PSB architecture, we also use a 256 entry 4-way stride address prediction table to filter stride predictions out of a 2K entry Markov table. We use a differential Markov table as described in section 4.2, where each entry in the Markov table is only 16-bits (total table size of 4Kbytes). The advantage of PSB over PC-Stride is that we can accurately follow non-stride based miss patterns. Results are shown for PSB for all four combinations of the allocation filter and priority scheduler. These are (1) two miss allocation filter with round-robing scheduling (2Miss-RR), (2) two miss allocation filter with priority confidence scheduling (2Miss-Priority), (3) confidence allocation with round-robin scheduling (ConfAlloc-RR), and (4) confidence allocation with priority scheduling (ConfAlloc-Priority). For the accuracy confidence counter stored in our stride table, we used a saturating value of 7, and for the priority confidence counter in the stream buffers we used a saturating confidence value of 12. See Section 4 for the other values used for the accuracy and priority counters.

For both the PC-Stride and the PSB architectures we used 8 stream buffers, each with 4 entries. All stream buffers are checked in parallel on a lookup. In addition, when a stream buffer generates a prediction, all stream buffers are checked to guarantee that the stream buffers do not follow overlapping streams.

For the address predictors we use, we predict the virtual address stream instead of the physical address stream, and we perform TLB translations on those addresses when performing the prefetch. Since we only insert loads into the stride PC-table on a cache miss, we only require a small

256 entry stride PC-table to capture all the critical loads that miss. Finally, we only store and use cache block addresses *not* the full address for both the stride and Markov tables.

Figure 5 shows the speedup over the baseline architecture IPC shown in Table 2 for PC-Stride and our predictor-directed stream buffer configurations. Results are shown for five pointer-based applications, and one stride-based FORTRAN program. We ran several FORTRAN programs, and they all had similar performance to the results shown for `turb3d`. Since these programs are mostly stride-based, our PSB architectures achieves basically the same performance as the PC-stride architecture, getting benefit only from the addition of confidence and scheduling. For pointer based applications, our results show that predictor-directed stream buffers can achieve significant speedups (17% speedup for `deltablue` and 18% speedup for `burg`) over using PC-stride guided stream buffers.

It can be seen in Figure 5 that confidence allocation is very important for `burg` and `sis`. The reason why performance degrades for `sis` when using 2Miss filter allocation for our approach is due to stream thrashing. Using the confidence counters to guide allocation, allows stream buffer allocation to concentrate on highly predictable loads, and avoids replacing stream buffers that are receiving a lot of hits. Stream thrashing is a serious problem for programs with large amounts of missing loads as is the case in both large programs and tight inner loops which are highly software pipelined. Performing loop unrolling and software pipelining increases the number of load instructions in the program, which can degrade the performance of stream buffers. If an architecture has stream buffers, a loop with a hardware predictable reference stream may achieve better performance performing no loop unrolling, and instead use the stream buffers to hide the load latency.

Figure 6 shows the prefetching accuracy for the different configurations examined, where prefetching accuracy is the percentage of all prefetches that were used by the processor. Allowing the stream buffer to follow non-stride predictions can increase the prefetching accuracy by almost a factor of 2 for `deltablue` when using confidence allocation.

Figure 7 shows the cache miss rates for the baseline and prefetching architectures. We define a cache miss as an access to a cache block which is not currently resident in the cache, i.e. accesses to in-flight data count as cache misses. We have found this tracking of cache misses to be more representative of the system behavior than simply checking the cache tags and MSHRs for a hit.

The total impact of the system can be seen in figure 8 which shows the average load latency for the different benchmarks and techniques. For `deltablue`, we remove 4 full cycles from the average latency, and 3 cycles for `burg`. Even a moderate reduction in average latency can produce a significant performance impact and this is reflected in the speedups obtained.

Figure 9 shows the percent of bus utilization for both the bus from the L1 to L2, and the bus from the L2 to main memory. This reveals an interesting characteristic of `sis`. When confidence is not employed, the prefetcher spends the majority of its time issuing useless prefetch requests to the L2 cache due to stream buffer thrashing. The bus utilization rises by a factor of four and the accuracy drops significantly.

By far the largest consumers of L1 to L2 bandwidth are `deltablue`, and `health`, and it is for these programs which stream buffer prioritization scheduling performed the best. The scheduling of prefetches allows the stream buffers that are most likely to hit to use the bandwidth first, allowing these high confidence prefetches to cover more latency. Stream buffer priority scheduling provided an additional speedup of 11% for `deltablue` when confidence was used in conjunction.

The speedup that we are achieving is due to the hiding of latency associated with capacity problems in the L1 cache. This is shown by figure 10, where we look at the performance for 16K 4-way, 32K 2-way, and 32K 4-way cache. The results show the speedup obtained for the different prefetching techniques over a baseline architecture with the same cache configuration. It can be seen that the speedup obtained is independent of cache size over a reasonable set of configurations.

6.1 Perfect Disambiguation Results

As mentioned earlier, we simulated the effects of perfect load-store disambiguation. The IPC results with and without perfect memory disambiguation for the baseline architecture and our proposed scheme are presented in Figure 11. For no disambiguation (NoDis), a load waits to issue until all prior stores have issued. Perfect store sets [11] provides a decent speedup for the baseline architecture for `deltablue` and `sis`. However it yields little improvement in conjunction with prefetching for all programs, except for `sis`.

7 Summary

We chose to focus on stream buffers because of their ability to follow address streams independent of what the fetch stream is doing. Prior stream buffer architectures were limited to following down a stream using a fixed stride [13], which limits their benefit for commercial pointer-based applications. To go beyond this limit we presented a new stream buffer architecture (Predictor-Directed Stream Buffer) to follow non-stride streams. In addition, we presented a new stream buffer allocation and priority scheduling technique based on confidence.

It should be noted that any address predictor can be used to guide the predicted prefetch stream for our predictor-directed stream buffer. Due to space constraints, we only

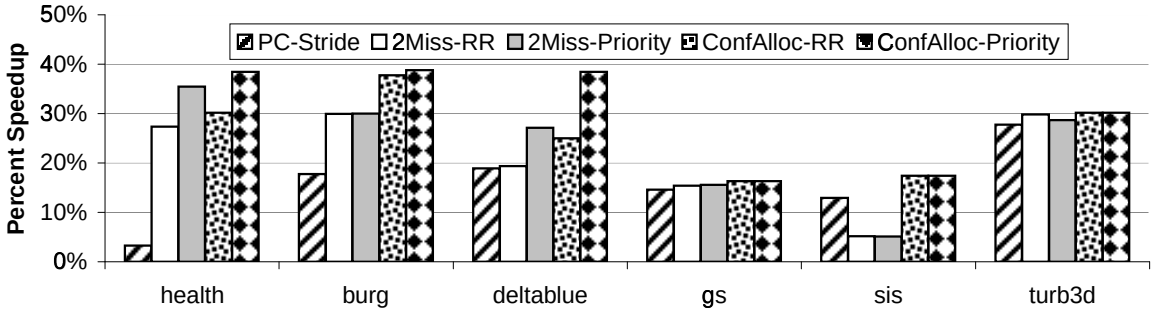

Figure 5: Percent speedup over base using prior PC-stride prefetching and our Predictor-Directed Stream Buffers.

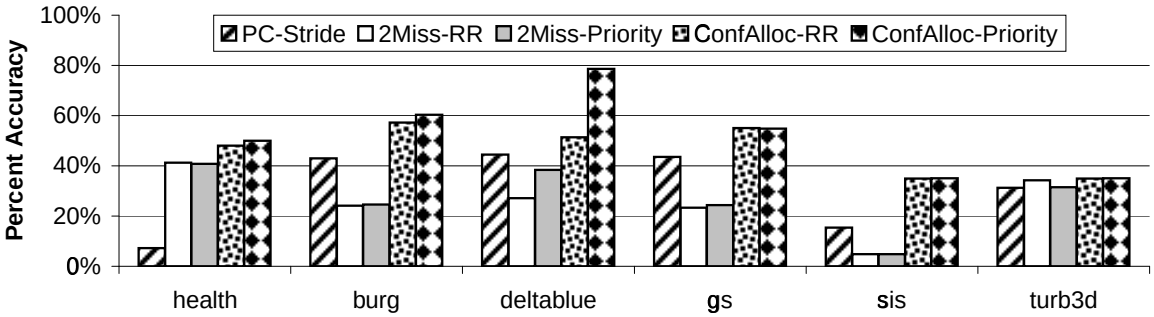

Figure 6: Prefetch accuracy. This is the number of prefetches used divided by the number of prefetches made.

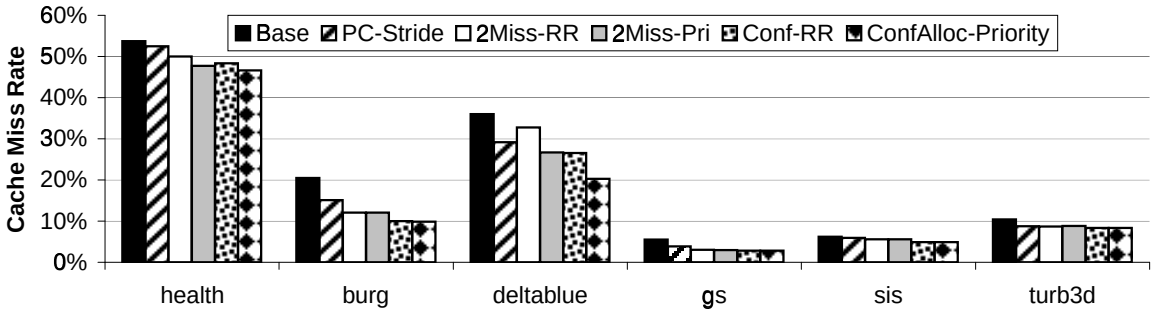

Figure 7: Data cache miss rates (where in-flight cache blocks count as a miss).

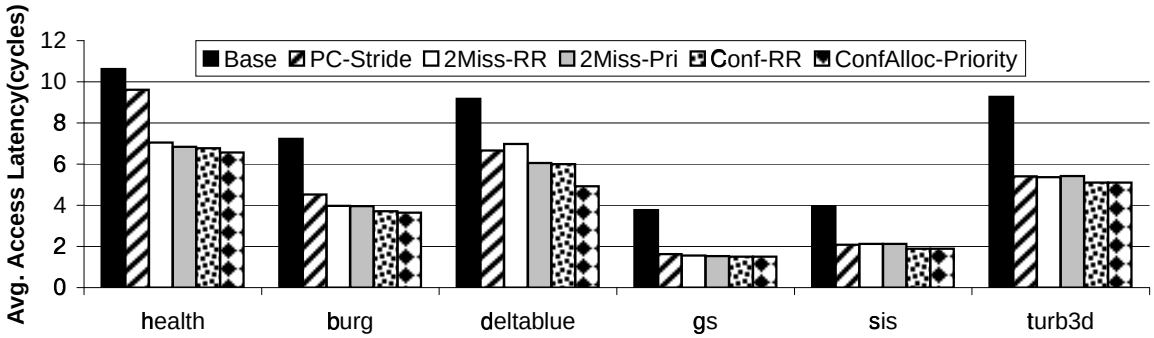

Figure 8: Average latency of a load in cycles for the different architectures.

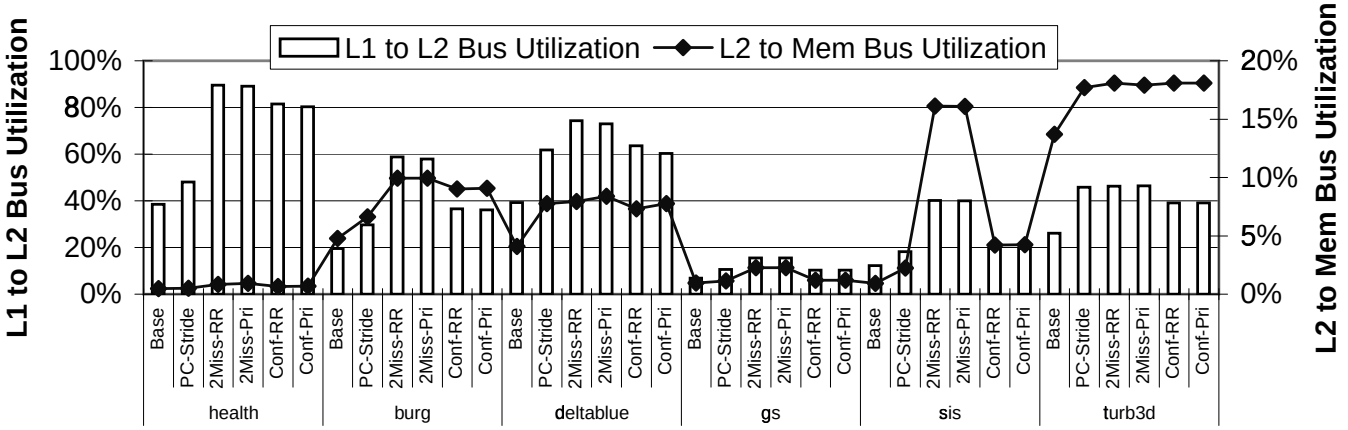

Figure 9: The percent of cycles the bus was busy. The L1-L2 bus utilization is shown with bars using the left axis, and the L2 to Main Memory bus utilization uses the right axis.

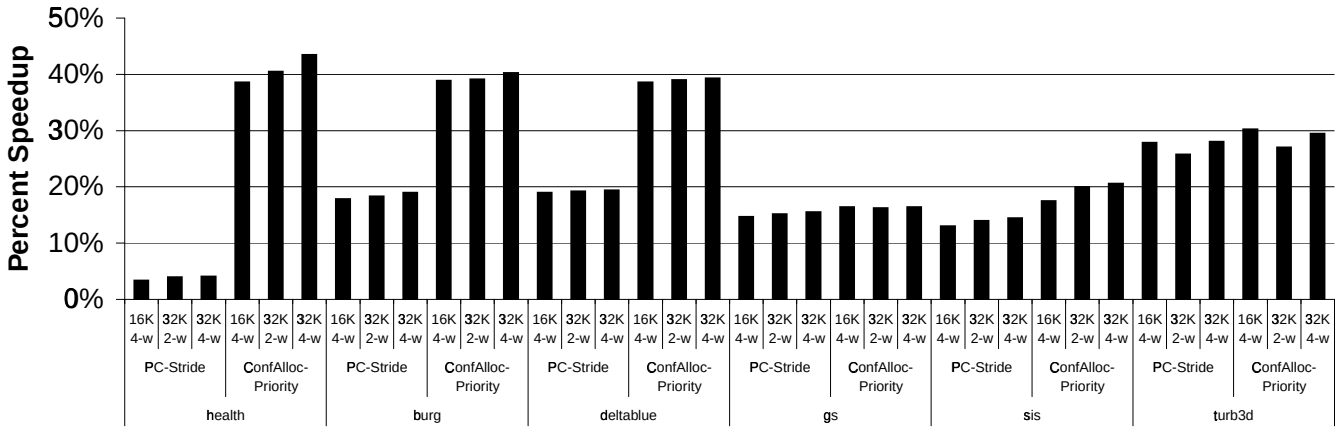

Figure 10: Percent Speedup Varying the Cache Size and Associativity.

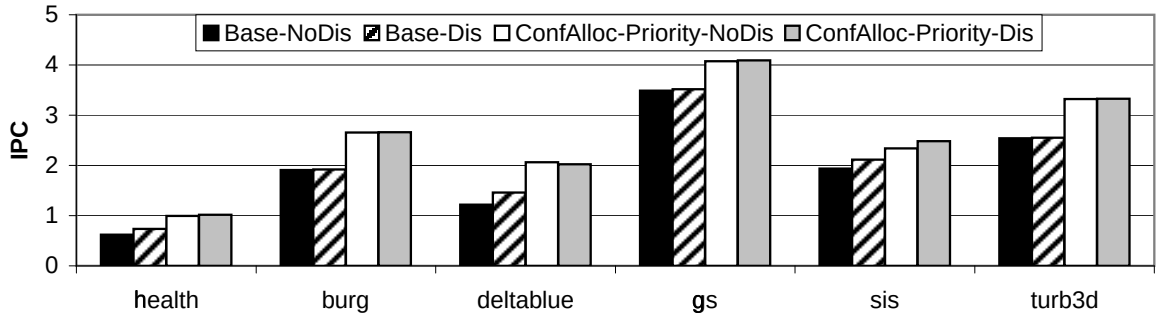

Figure 11: Performance results with (Dis) and without (NoDis) perfect store sets (perfect disambiguation).

presented results for using a stride-filtered Markov address predictor to guide stream buffer prefetching. The Markov predictor was a differential Markov predictor whose data size was only 4Kbytes. We found this predictor to perform

better than other recently proposed context [28] and correlated predictors [2] for data prefetching.

For stride-based applications (e.g., FORTRAN programs), predictor-directed stream buffers provided similar performance to stride-based stream buffers. For the 5 pointer-based applications we examined, predictor-directed stream buffers provide a 30% speedup on average over no prefetching, and 10% average speedup over using the best performing prior stream buffer architecture.

Acknowledgments

We would like to thank the anonymous reviewers for providing useful comments on this paper. This work was funded in part by NSF CAREER grant No. CCR-9733278, by DARPA/ITO under contract number DABT63-98-C-0045, and a grant from Compaq Computer Corporation.

References

[1] 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.

[2] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappoport, A. Yoaz, and U Weiser. Correlated load-address predictors. In *26th Annual International Symposium on Computer Architecture*, May 1999.

[3] A. Berrached, L. Coraor, and P. Hulina. A decoupled access/execute architecture for efficient accesss of structured data. In *In the Hawaii International Conference on System Services*, January 1993.

[4] B. Black, B. Mueller, S. Postal, R. Rakvie, N. Utamaphethai, and J. P. Shen. Load execution latency reduction. In *12th International Conference on Supercomputing*, June 1998.

[5] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[6] M.J. Charney and T.R. Puzak. Prefetching and memory system behavior of the spec95 benchmark suite. *IBM Journal of Research and Development*, 41(3), May 1997.

[7] M.J. Charney and A.P. Reeves. Generalized correlation based hardware prefetching. Technical Report EE-CEG-95-1, Cornell University, February 1995.

[8] T-F. Chen and J-L. Baer. Effective hardware-based data prefetching for high performance processors. *IEEE Transactions on Computers*, 5(44):609–623, May 1995.

[9] T.F. Chen and J.L. Baer. Reducing memory latency via non-blocking and prefetching caches. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 51–61, October 1992.

[10] C. Chi and C. Cheung. Hardware-driven prefetching for pointer data references. In *In the ACM International Conference on Supercomputing*, pages 377–384, June 1998.

[11] G. Chrysos and J. Emer. Memory dependence prediction using store sets. In *25th Annual International Symposium on Computer Architecture*, June 1998.

[12] R. J. Eickemeyer and S. Vassiliadis. A load instruction unit for pipelined processors. *IBM Journal of Research and Development*, 37:547–564, July 1993.

[13] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. Memory-system design considerations for dynamically-scheduled processors. In *24th Annual International Symposium on Computer Architecture*, June 1997.

[14] K. Farkas and N. Jouppi. Complexity/performance tradeoffs with non-blocking loads. In *21st Annual International Symposium on Computer Architecture*, pages 211–222, April 1994.

[15] M. Farrens and A.Pleszkun. Implementation of the pipe processor. *IEEE Computer*, January 1991.

[16] J. Gonzalez and A. Gonzalez. Speculative execution via address prediction and data prefetching. In *11th International Conference on Supercomputing*, pages 196–203, July 1997.

[17] 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.

[18] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *24th Annual International Symposium on Computer Architecture*, June 1997.

[19] N. Jouppi. Improving 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.

[20] S. McFarling. Combining branch predictors. Technical Report TN-36, Digital Equipment Corporation, Western Research Lab, June 1993.

[21] T.C. Mowry, M.S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, October 1992.

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

[23] G. Reinman, B. Calder, and T. Austin. Fetch-directed instruction prefetching. In *32nd International Symposium on Microarchitecture*, November 1999.

[24] G. Reinman, B. Calder, and T. Austin. A power efficient speculative fetch architecture. Technical Report UCSD-CS2000-0657, University of Califonia, San Diego, June 2000.

[25] A. Roth, A. Moshovos, and G. Sohi. Dependence based prefetching for linked data structures. In *Eigth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[26] A. Roth and G. Sohi. Effective jump-pointer prefetching for linked data structures. In *26th Annual International Symposium on Computer Architecture*, May 1999.

[27] A. Saulsbury, F. Dahlgren, and P. Stenstrom. Recency-based tbl preloading. In *27th Annual International Symposium on Computer Architecture*, June 2000.

[28] Y. Sazeides and J. E. Smith. The predictability of data values. In *30th International Symposium on Microarchitecture*, pages 248–258, December 1997.

[29] Y. Sazeides and J. E. Smith. Modeling program predictability. In *25th Annual International Symposium on Computer Architecture*, June 1998.

[30] T. Sherwood and B. Calder. Time varying behavior of programs. Technical Report UCSD-CS99-630, University of Califonia, San Diego, August 1999.

[31] J. E. Smith and W.-C. Hsu. Prefetching in supercomputer instruction caches. In *Proceedings of Supercomputing*, November 1992.

[32] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *30th Annual International Symposium on Microarchitecture*, December 1997.

[33] C. Yang and A. Lebeck. Push vs. pull: Data movement for linked data structures. In *In the ACM International Conference on Supercomputing*, June 2000.