A Decoupled Predictor-Directed Stream Prefetching Architecture

Suleyman Sair Timothy Sherwood Brad Calder

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

Index Terms: Data Prefetching, Stream Buffers, and Address Prediction

Abstract

An effective method for reducing the effect of load latency in modern processors is data prefetching. One form of hardware-based 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*), which allows the stream buffer to follow a general address prediction stream instead of a fixed stride. A general address prediction stream complicates the allocation of both stream buffer and memory resources, because the predictions generated will not be as reliable as prior sequential next-line and stride-based stream buffer implementations. To address this, we examine using confidence-based techniques to guide the allocation and prioritization of stream buffers and their prefetch requests. Our results show, when using PSB on a benchmark suite heavy in pointer-based applications, that PSB provides a 23% speedup on average over the best previous stream buffer implementation, and an improvement of 75% over using no prefetching at all.*

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. These techniques use locality analysis to determine where to insert prefetch instructions and show significant improvements [27]. 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 on improving the performance of data prefetching with stream buffers. Stream buffers were originally proposed by Jouppi [22] 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. Along with introducing allocation filters, Palacharla and Kessler [28] 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. [16] 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. Then that prefetch address can be used to generate the next predicted address to prefetch. After allocation, a stream buffer is decoupled from the execution stream. In other words, it can independently prefetch down a predicted address stream. 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. [16] 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 PSB 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 [10, 15] 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 address value and the stride. We chose to use the two-delta stride predictor [15, 38], 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 [38, 39, 46] and *Markov* [7, 8, 21] predictors are fundamentally similar, in that each predictor bases its prediction on a set of past 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 As that are followed by a B. The Markov predictor described in [21] is a first order context predictor as it uses only the last address to predict the next one.

Bekerman et. al. [3] 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 [3], 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 [5, 9, 11, 19].

Chen and Baer [9] proposed an approach to provide the load prediction early by using a Look-Ahead PC(LA-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 advance 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. [29, 30, 31] 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. These same fetch addresses can also be used to guide data prefetching, similar to what was proposed in [9].

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 each prefetch generated.

An early example of a demand-based prefetching architecture is *Next Line Prefetching* (NLP) by Smith [42], 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 by Charney and Puzak [7]. In this approach, 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 valid 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.

A recently proposed demand-based prefetcher identifies/predicts when an L1 data cache block becomes "dead" (i.e. evictable). It then uses an address predictor to prefetch a cache block to be potentially used in the future in the place of that "dead" block [23]. The result of the address predictor is similar to shadow prefetching [7], predicting a cache block that was previously mapped to the same block frame. It will potentially record the address that occurred right after the dead cache block was evicted, and then predict this same address when that cache block is predicted to be "dead" in the future.

The last example we will discuss is the Markov prefetcher used by Joseph and Grunwald [21]. 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.

3.3 Decoupled/Stream Prefetching

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

3.3.1 General Decoupled Models

An access decoupled architecture partitions programs into a prefetching instruction stream and an execution instruction stream [4, 18, 20]. As long as the prefetch stream can run ahead of the execution stream, the memory latency can be masked. Roth et. al. [32, 33] have examined both a software and hardware approach for prefetching recursive data structures using a decoupled model. Yang and Lebeck [47] examined an architecture which uses the compiler to create small prefetch kernels of instructions, which are executed in parallel with the original applica-

tion on a separate prefetch engine. A similar technique was recently proposed by Solihin et. al [43]. They present a scheme where context-based prefetching is performed by a User-Level Memory Thread running on a simple general-purpose processor in memory. A software handler records L2 miss addresses in a correlation table which resides in memory. When a stripped down version of the target application executes on the processor in memory, the software handler accesses the correlation table to prefetch several blocks into the L2 cache. To provide high coverage and timely prefetches, their correlation table stores several levels of successor misses per entry.

3.3.2 Stream Buffers

Jouppi introduced *stream buffers* to improve direct mapped cache performance [22]. The stream buffers follow multiple streams prefetching them in parallel and these streams can be 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 [28] suggested two techniques to enhance the effectiveness of stream buffers: *allocation filters* and a *minimum delta non-unit stride* detection mechanism. The filter prevents a stream buffer from being allocated until two consecutive misses occur for the same stream. With the non-unit stride 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.

Farkas et. al. [16] made an important contribution by extending Palacharla and Kessler's 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. We compared the global minimum-delta approach to the PC-stride predictor and found that it was uniformly outperformed by the per-load stride detector. Therefore, we only present comparison results of our approach with PC-based stride prediction stream buffers.

Farkas and Jouppi [17] 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 in [22], they used a fully-associative stream buffer lookup, which we model.

3.3.3 Speculative Pre-Execution

There has recently been a plethora of research into using speculative pre-execution on idle hardware threads to hide memory latencies [2, 13, 14, 24, 26, 34, 35, 44, 48]. The goal of this area of research is to (1) identify loads that are problematic, (2) construct a minimal sequence of instructions (kernel) needed to produce the address stream for these loads, and then (3) execute this minimal kernel speculatively on an idle thread in a Simultaneous Multithreading processor [45]. These architectures attempt to prefetch the address stream ahead of the execution stream, via the minimal kernel running on a spare multi-threaded hardware context.

Most of these techniques rely on a profiler and a compiler or the user to determine the problematic loads and construct minimal kernels for them. Others rely on a profiler to determine delinquent loads, but construct the kernels in hardware [2, 13].

This decoupled form of prefetching has been shown to achieve speedups similar to the decoupled prediction-directed architecture we present in this paper. The advantage of the speculative pre-computation approach is that it follows an actual instruction stream allowing it to more effectively determine the pointer traversals to follow, since it pre-executes branch instructions. A potential disadvantage of speculative pre-execution is that the minimal kernels can sometimes have a hard time running far enough ahead of the miss stream to hide the miss latency due to dependencies in the pointer chain.

In comparison, our predictor-directed approach can easily run ahead when traversing the pointer-chain since it is based on predicting the address. This of course assumes that a compressed version of the pointer traversal over the miss stream can be captured in our predictor. The disadvantage is that our PSB approach can have lower prefetch accuracy when it comes to objects with multiple possibilities for next pointer traversals guarded by branches, since our prefetcher currently does not incorporate branch prediction information to help guide the stream buffer down its prediction stream.

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 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. [16]. 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). 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. An implementation of this is shown in Figure 2.

There are two major parts to PSBs, a per-stream history which is stored with each stream buffer, and an address predictor which is shared between stream buffers. The per-stream history is used to keep data about a particular stream buffer including confidence information, and local stride. This data may be used for a variety of purposes, such as indexing into the address predictor. The primary service of the per-stream history is to store the current speculative state which can be fed to the predictor to generate the next address prediction for that stream buffer. 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 *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 type of address predictor.

4.1 Predictor-Directed Stream Buffer Implementation

Figure 2 shows a specific implementation 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 describe the initialization and steady

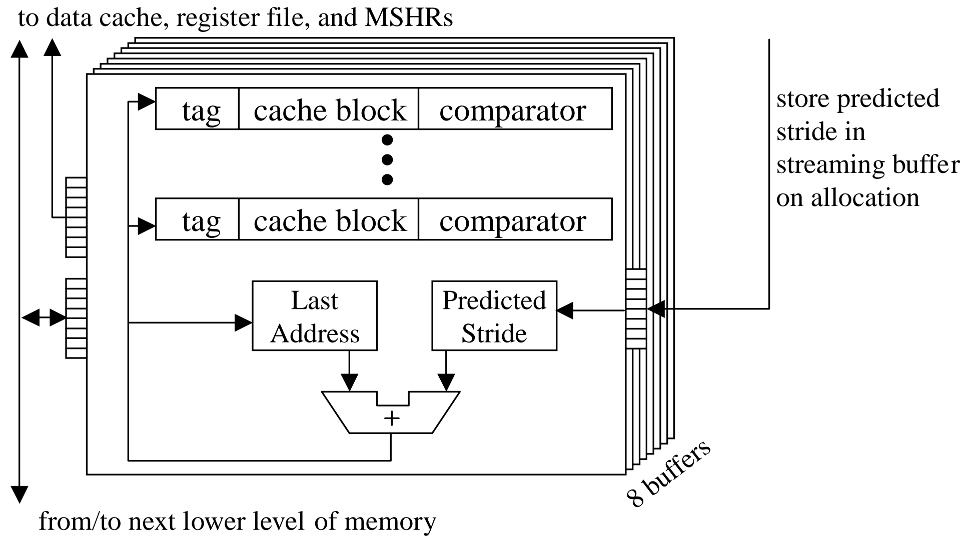

Figure 1: Stride-based Stream Buffer Architecture. Eight stream buffers are shown (overlapping each other). Each stream buffer can hold N cache blocks. When a stream buffer is allocated, it is assigned a predicted stride to use to generate all of its prefetch addresses.

state operation of 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. If a load miss is assigned to a stream buffer, the load PC, current address, and any additional prediction information are copied 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. This would

then be used 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., [16] to prevent stream buffers from prefetching down overlapping paths. To alleviate the pressure on the stream buffer ports and tags, we replicated the tags and added a dedicated port for this check. Note that this does not introduce a significant overhead as there are only 8 stream buffers and they each have only 4 entries. If the prediction was found to be already resident in a buffer entry, then it is ignored, no useful prediction is made that cycle, and the stream buffer prediction history information is updated. If prediction is not found in a 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 prefetching 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 3 cycles whereas the stream buffer lookup latency is a single cycle. If there is a hit in the stream buffer, and the data is not 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 yet, 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 [3, 21, 38, 39, 46] can be used to guide the predictor-directed stream buffer. We examined several types of predictors (including stride with correlated [3]), but only provide results for a SFM table, as it performed uniformly better.

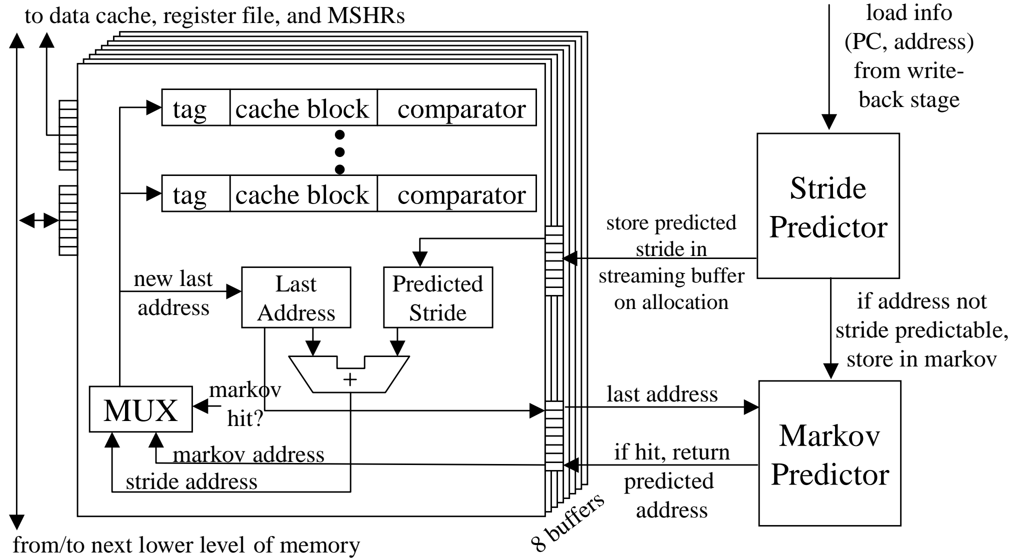

Figure 2: Stride-Filtered Markov Predictor-Directed Stream Buffer Architecture. When a stream buffer is allocated, it is assigned a fixed stride from the stride table. To generate the next prefetch address the last address is (1) looked up in the Markov table and (2) used to calculate a next stride address.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

4.2 Stride-Filtered Markov Predictor

Charney and Reeves [8] and also Joseph and Grunwald [21] 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 2. 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 update the predictor with loads that receive their value forwarded from stores, 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 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.

When a stream buffer is allocated, the 2-delta stride table is accessed, and a stride is assigned to the stream

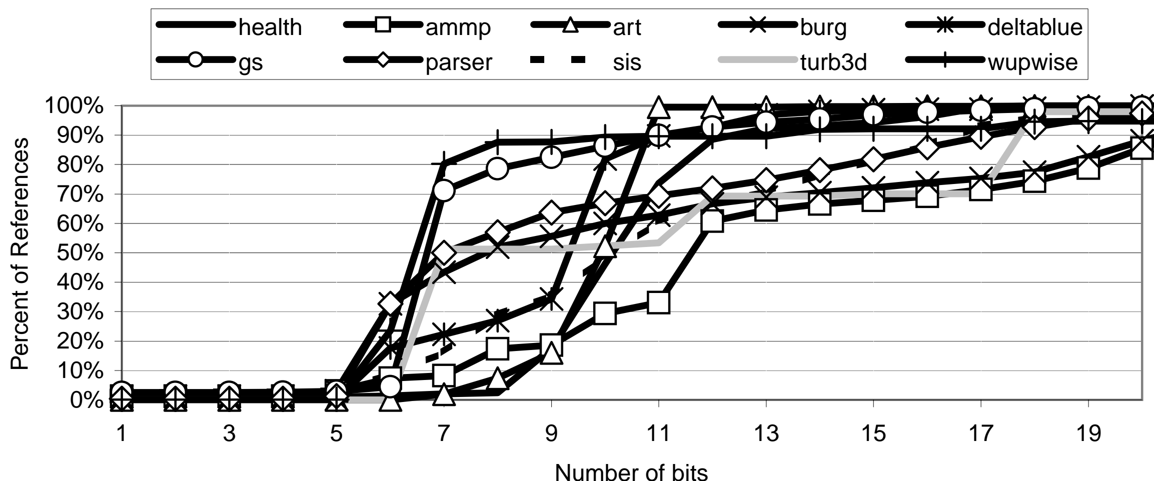

Figure 3: The number of bits to accurately predict cache misses using the Markov Difference Predictor. The y-axis shows the percent of L1 cache misses that could be correctly predicted given the number of bits used for each entry of the markov table shown on the x-axis. The cache miss address is predicted by adding together the address used to index the Markov table with the value stored in the Markov table.

buffer. If there is no 2-delta stride in the table, then a next line stride is assigned by default. The stride assigned to the stream buffer is used to generate the predicted address if the Markov table misses during the prediction generation. Otherwise if there is a Markov table hit, then the Markov address is used for the prediction.

For the SFM predictor examined in this paper, we only use the current address 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 [21], 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 as described below.

In order to reduce the size of the Markov predictor table we store into the table *only the difference (at the cache block granularity) between consecutive cache miss addresses*, rather than the full address as is done in prior work. 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 3 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 20 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 addition, the tag size can also be reduced by storing only partial address tags. In this paper we use a Markov table with 2K entries, which uses a total of 5Kbytes for the data storage.

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. [16] showed that using *two miss stride filtering* provides 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 either 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's miss stream can be predicted before we allocate a stream buffer to it. This is used to 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 guides stream buffer allocation. If the address prediction confidence level (accuracy counter) 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. Intuitively, this policy tries to deallocate stream buffers that do not incur frequent hits. If a stream buffer can eliminate 5% or more of the misses then the priority counter value will not be affected by aging because on average it will get a hit during a 20 miss period. 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 based on their priority counter. If there are several stream buffers that are at the same confidence level, we use an LRU policy to choose the winner.

As described earlier in section 4.3, the priority counter is also used to guide stream buffer allocation along with the accuracy counters. 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. When a stream buffer is allocated, the accuracy confidence is copied into the stream buffer's priority counter.

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. For this study, we assumed a multi-ported TLB that can support two demand accesses and one prefetch access per cycle. On a prefetch, we access the data TLB for the translation and perform a replacement if necessary. In essence, this amounts to TLB prefetching [37]. 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 of TLB misses (except for `ammp`). As an optimization, 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.

Program	Description
health	A hierarchical health-care system simulator taken from the Olden Benchmark suite (input: 3 500).
ammp	This benchmark solves the ODE defined by Newton's equations for the motions of the atoms in the system on a protein-inhibitor complex which is embedded in water (input: ref).
art	The Adaptive Resonance Theory 2 neural network is used to recognize objects in a thermal image. The objects are a helicopter and an airplane. The neural network is first trained on the objects. After training is complete, the learned images are found in the scanfield image (input: ref).
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 (input: long).
gs	Ghostscript is an implementation of Adobe Systems' PostScript (tm) language. The input run converts a PostScript file into a jpeg.
parser	The Link Grammar Parser is a syntactic parser of English, based on link grammar, an original theory of English syntax. Given a sentence, the system assigns to it a syntactic structure, which consists of set of labeled links connecting pairs of words (input : ref).
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 (input: markex).
turb3d	Simulates isotropic, homogeneous turbulence in a cube with periodic boundary conditions in x,y,z coordinate directions (input: ref).
wupwise	Wupwise is an acronym for Wuppertal Wilson Fermion Solver, a program in the area of lattice gauge theory involving quantum chromodynamics (input: ref).

Table 1: Description of benchmarks used.

5 Methodology

The simulator used in this study was derived from the SimpleScalar/Alpha 3.0 tool set [6], 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. 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 instructions 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 [40] before gathering statistics. `Ammp, art,`

program	#inst (Million)	%L1 Miss Rate	%loads	%stores	IPC	L1-L2 %bus utilization	L2-Mem %bus utilization
health	11	50.72	36	14.2	0.76	37.4	0.5
ammp	300	26.69	27.6	4.5	0.19	3.6	11.4
art	300	57.27	30.2	7.6	0.41	23.9	60.3
burg	300	18.60	19.1	18.7	1.97	17.4	4.9
deltablue	96	35.10	28.9	9.9	1.47	38.3	4.1
gs	300	3.42	19.2	6.8	3.43	4.2	0.9
parser	300	5.63	27.7	8.8	1.48	7	4.1
sis	300	3.99	28.7	36.8	10.5	4.8	0.6
turb3d	300	9.23	23.3	16.2	2.58	24.6	13.9
wupwise	300	4.26	22.2	7.5	2.42	5.7	9.9

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 L2 to main memory was busy.

`parser` and `wupwise` were also fast forwarded by 2, 2.9, 10.7 and 2.5 billion instructions respectively. These fast forward numbers were derived using the SimPoint tool [41]. Results for `health` are provided for comparison purposes as this is a popular benchmark in prefetching studies. It is not intended to reflect the true merits of our prefetching architecture. More detailed analysis of the pointer behavior (pointer variability and fan-out) for these programs can be found in [36].

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 [12]. Perfect store sets cause loads to only be dependent on stores that they are actually dependent upon. In this way loads will not be held up by false dependencies making the prefetcher look better. All the architectures simulated in this study utilize perfect store sets.

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. We use a McFarling gshare predictor [25]

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 4-way associativity and 64-byte lines. The L1 cache has a 3 cycle hit latency while stream buffer hits incur a 1 cycle latency. 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. [16], which was described in Section 3. We call their approach *PC-Stride*. Loads that miss in the data cache 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 entries, but they provided little to no improvement.

For our PSB architecture, we also use a 256 entry 4-way PC-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 20-bits (total table size of 5 Kbytes). The advantage of PSB over PC-Stride is that we can accurately follow non-stride based miss patterns. For the counter stored in our stride table, which we use to estimate accuracy in the allocation policy, we used a saturating value of 7. For the priority counters in the stream buffers, we used a saturating value of 12. Table 3 summarizes the different simulation parameters and their respective values.

For both the PC-Stride and the PSB architectures we used 8 stream buffers, each with 4 entries unless otherwise noted. 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.

There are two possible address types we could choose to predict, virtual or physical. If virtual addresses are used, these will have to pass through the TLB before being prefetched from memory. On the other hand, if physical addresses are used, the effectiveness of the predictors will be greatly diminished when address traces cross page boundaries. We chose to predict virtual addresses and translate the prefetches with the TLB. In this study we assumed an additional port on the TLB to handle the prefetch requests, and there can be at most one prefetch per cycle, unless otherwise noted. If the predictions are accurate, these prefetch requests will actually

parameter	value
L1 data cache	32K, 4-way, 32B blocks
L1 instruction cache	32K, 2-way, 32B blocks
Unified L2 cache	1M, 4-way, 64B blocks
L1-L2 bus bandwidth	8 bytes/cycle
L2-Memory bus bandwidth	4 bytes/cycle
L1 hit latency	3 cycles
Load-store forward latency	2 cycles
L2 hit latency	12 cycles
Memory access latency	120 cycles
TLB miss latency	30 cycles
Number of stream buffers	8
Number of stream buffer entries	4
Stream buffer hit latency	1 cycle
Stride predictor size	256 entries, 4-way set associative
Markov predictor size	2048 entries
Markov predictor entry size	20 bits
Predictor access latency	1 cycle
Accuracy counter saturation	7
Priority counter saturation	12
Allocation threshold	1
Accuracy counter increment	1
Accuracy counter decrement	1
Priority counter increment	2
Priority counter decrement	1
Priority counter aging period	10 L1 misses

Table 3: Summary of simulation parameters. The top parameters summarize the baseline architecture. The next group lists the additional parameters for the PC-Stride stream buffer architecture. The final group describes the additional parameters on top of that needed for the Predictor-directed Stream Buffer prefetching architecture.

generate the equivalent of a TLB prefetch [37]. Therefore, using effective cache prefetching performs the function of a dedicated TLB prefetcher. However, the benchmarks we examine do not have high enough TLB miss rates to see much benefit from the TLB prefetching.

To reduce the size of the tables used in the prediction phase of prefetch, we employ two small optimizations. First, in order to reduce the size of the stride PC-table, we only perform inserts when there has been a cache miss. In this way we can reduce the PC-table size down to 256 entries while still capturing all of the critical loads that we may come across. In addition to that, to reduce the size of the stride and the Markov tables, the tables *only store cache block address*, instead of the full address. Just using the cache block aligned address instead of the full byte address, reduces each entry by 5 bits while still being sufficient to perform any prefetch. Finally, to reduce the size of the Markov predictor table we only store *the difference* (at the cache block granularity) between consecutive cache miss addresses, rather than the full address. For the programs we looked at, our results indicate 20-bits are sufficient to cover almost all of the miss transitions in the Markov predictor.

6.1 Results

To analyze the benefits of our approach, we now show the overall effect that the stream buffers have on the performance of the system. Figure 4 shows performance results for the baseline architecture, the best prior approach, PC-Stride, and our new Predictor-Directed Stream Buffers with and without confidence based allocation and priority. PSB results are shown 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-Pri), (3) confidence allocation with round-robin scheduling (Conf-RR), and (4) confidence allocation with priority scheduling (Conf-Pri). The results were generated for several pointer-based applications, and two stride-based FORTRAN programs. We ran several FORTRAN programs, and they all had similar performance to the results shown for `turb3d` and `wupwise`.

The most notable performance improvements can be seen on `health`, `ammp`, `burg`, and `deltablue`. All of these programs benefit significantly from the addition of a Markov predictor. The speedups over the best previously known stream buffer technique, PC-Stride, range from under 1% for `gs` which has very little pointer behavior, to over 100% for `ammp` which has many misses, most of which go to main memory. `Wupwise` exhibits one shortcoming of the confidence-based allocation scheme. For this program, the confidence-based configurations allocate stream buffers to references that are L1 misses but are often found in the L2 cache. The configurations based on two miss filters on the other hand allocate buffers for references that are L2 misses as well. Since the two miss filter approach is able to hide longer latencies, even though it hides fewer misses in the big picture, it achieves higher speedup than confidence-based allocation. One remedy for this problem can be to assign higher priority/confidence to stream buffers that prefetch streams that service the slower levels of the memory hierarchy. For most of the programs, confidence based allocation works better. Using confidence based allocation makes sure that stream buffers are only deallocated when they are no longer providing useful prefetches.

6.1.1 Accuracy and Coverage

In understanding how the performance of the system will be affected by stream buffers it is important to verify that the optimizations do not detrimentally affect prediction accuracy and prefetch coverage.

To quantify the benefit of using the Markov predictor with our PSB architecture, we measured the accuracy of the address predictors at predicting cache misses. To measure accuracy, we count the number of times the predictor (either PC-Stride or Stride-Filtered Markov) was able to correctly predict the miss address, for all cache misses. When a new miss occurs for the SFM predictor, we generate a prediction from the stride PC and Markov

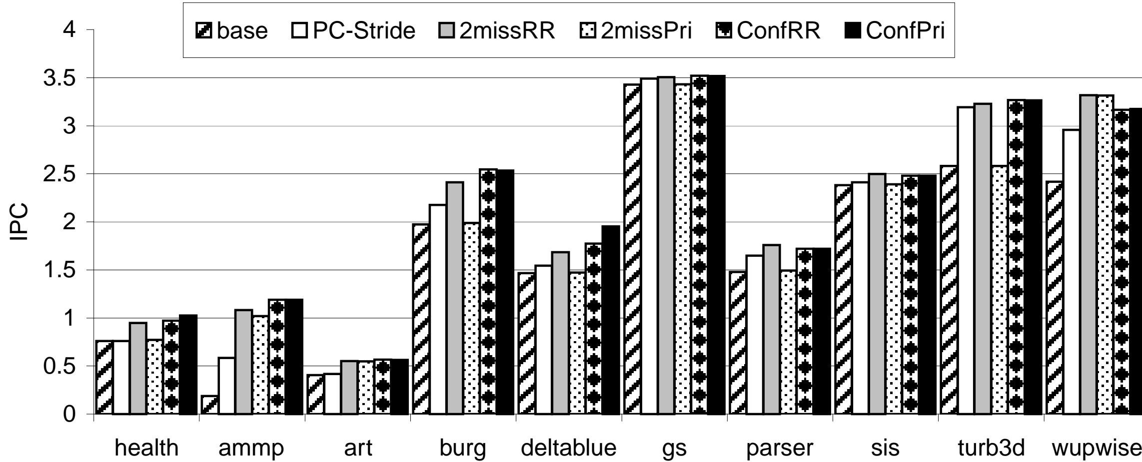

Figure 4: IPC performance results of base case, prior PC-Stride prefetching, and our Predictor-Directed Stream Buffers.

tables, and see if the miss would have been correctly predicted by the Markov prediction if there was a Markov hit, otherwise by the stride predictor. Note, this prediction accuracy is not the accuracy of the stream buffers. It is the accuracy of the address predictors for predicting every cache miss on the non-speculative path of execution. The accuracy of the two-delta stride predictor used for PC-Stride prefetching, and the Stride-Filtered Markov predictor used in our PSB configurations are shown in Figure 5. The results show that all but two programs experience a significant increase in miss stream prediction when using the SFM predictor instead of only a PC-Stride predictor.

We now consider the coverage of the prefetches, and how it relates to predictor accuracy. Coverage is defined as the percentage of base case misses that have their latency *reduced* by either completely hiding the memory latency or hiding some part of it via stream buffer prefetching. The bars in Figure 6 show coverage results for PC-Stride and the Confidence Priority PSB when using 8 stream buffers, where at most one prefetch can be initiated per cycle. The results show that the Predictor-Directed Stream Buffers increase the useful prefetches by augmenting the PC-Stride scheme with prefetches it could not have captured before.

When comparing predictor accuracy in Figure 5 with prefetching coverage Figure 6, we can see that the coverage is not as high as it could be. Note that Figure 5 is an upper bound for the coverage PSB could achieve, since the accuracy results are for updating the predictor after each prediction on the non-speculative path. There are two main reasons for the low coverage. The first reason is due to stream buffer contention amongst these high confidence miss streams preventing the coverage of some of the predictable streams. In Figure 6, we show an “X” for a special configuration of Confidence Priority PSB with 128 stream buffers, each with 4 entries, where we can perform up to 4 predictions and 4 prefetches per cycle. The IPC performance results for this configuration are discussed in Section 6.2. When removing the stream buffer contention and increasing prefetch

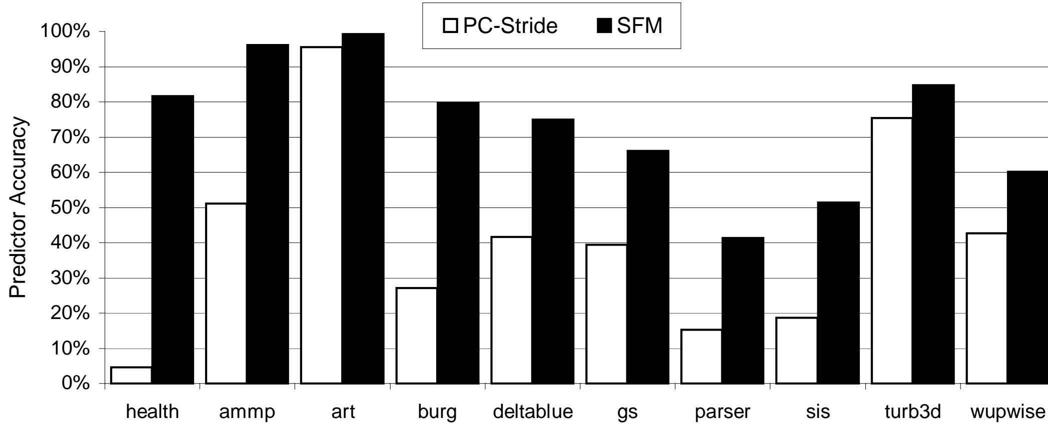

Figure 5: Predictor accuracy. This is the number of potential correct address predictions for all of the data cache misses for the PC-Stride predictor and the Stride-Filtered Markov predictor.

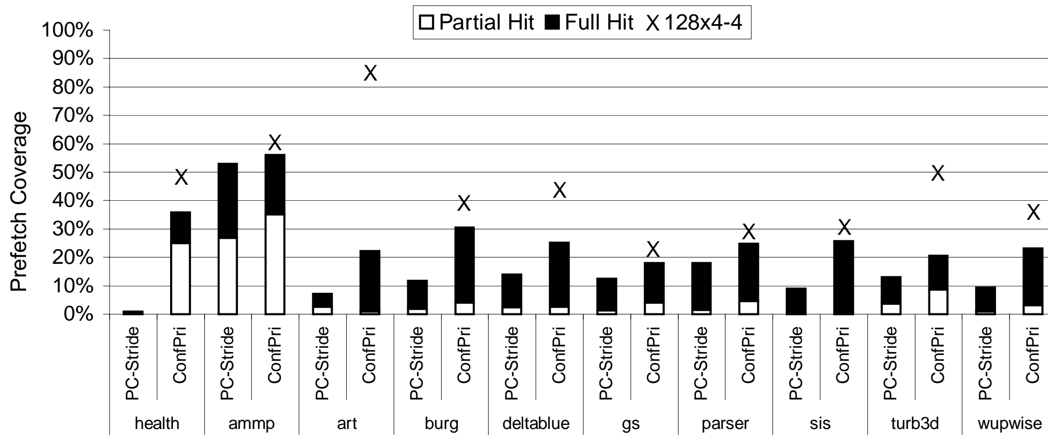

Figure 6: Prefetch coverage breakdown. This is the number of stream buffer hits divided by the number of base case L1 misses. It represents the number of cache misses we are able to cover/remove with stream buffers compared to an architecture with no prefetching. The PC-Stride and Confidence Priority results are for the baseline prefetching architecture where we have 8 stream buffers and only one prefetch can be initiated per cycle. For these configurations, we show the breakdown of the stream buffer hits that completely hide the memory latency and those that only partially hide the cache miss latency. The "X" results show the prefetch coverage for the Confidence Priority scheme with 128 stream buffers (each with 4 entries), where up to 4 predictions and prefetches can be initiated per cycle.

bandwidth, coverage increases significantly for programs like `health`, `art`, `deltablue` and `turb3d`. In fact, the coverage for `art` increases to 85% of all cache misses. The second reason for the lower than expected coverage is that a stream buffer follows only an address prediction stream, and it is decoupled from the execution stream. Once a stream buffer is allocated, it builds each prediction upon a prior prediction, so the probability of each subsequent prediction being correct will decrease. In addition, we've seen that pointer transitions guarded by conditional branches end up choosing the incorrect Markov predictor entry for some predictions, because no

branch information is available to guide the speculative prediction stream. Therefore, incorporating branch history information into the predictor may be able to improve the stream buffer prefetch coverage.

6.1.2 Timeliness of Prefetches

The third criteria for a prefetch architecture is that it provide its results in a timely manner. Figure 6, along with Figures 7, 8, and 9 quantify the timeliness of the prefetches.

Because loads are scheduled optimistically assuming an L1 hit latency, it is important to measure the number of loads that now have 1 or more stall cycles (beyond a full L1 hit or full stream buffer hit), which is shown in Figure 7. This figure shows the percent of loads that stall due to either a complete miss in the cache and stream buffer or stall due to a partial hit. The percentage of loads that encounter a stall is significantly reduced with PSB for all programs except `ammp`. For `ammp` the total number of stalling loads is increased by around 30% over PC-Stride. Even so, the performance for `ammp` is better for PSB than PC-Stride.

`Ammp` has two important loads dependent upon each other in a loop, which our Confidence Priority scheduling scheme accurately focuses on. The round-robin scheduling used by PC-Stride causes delay in initiating the prefetches for these important loads. For one of these loads, the prefetch always occurs late resulting in the execution stream waiting on a high latency stall (comparable to memory access latency) for that partial hit. This allows the second load to be prefetched with its prefetch latency being completely hidden, since it is overlapped with the prior prefetch. This pattern repeats during this loop's execution resulting in PC-Stride having fewer partial hits, but its partial hits have a long latency almost as high as a main memory access. The Confidence Priority architecture on the other hand, focuses the prefetch resources on these two loads. This results in shorter stalls for both loads. This allows the execution stream to keep up with the prefetch stream of every load, making it hard to completely hide the prefetch latency for these two loads. This results in more partial hits for Confidence Priority as Figure 7 shows, but the latency of each partial hit is significantly smaller than PC-Stride as seen in Figure 9. This results in overall better performance for Confidence Priority over PC-Stride.

Figure 8 shows the average load latency for the different benchmarks and techniques, showing where the time is being spent for the percent of loads that stall from Figure 7. The L1 cache hit latency is 3, anything above this is a cycle spent waiting by a load for its result. Since some L1 hits are only partial hits (i.e. the tag hits but data is not ready), average L1 hit latency can actually be higher than 3 cycles. We break down the latency by where it is waiting for memory from L1 (L1-hit), L2 (L2-hit), Memory (L2-Miss), or a Stream Buffer (SB hit). Cycles for partial hits are charged to the structure that the reference is due to arrive at. If the prefetches we made were not timely, the cycles spent waiting for each load would show up as waiting for stream buffers. The results show that after applying the optimizations there is only one program, `ammp`, that spends a significant number of cycles

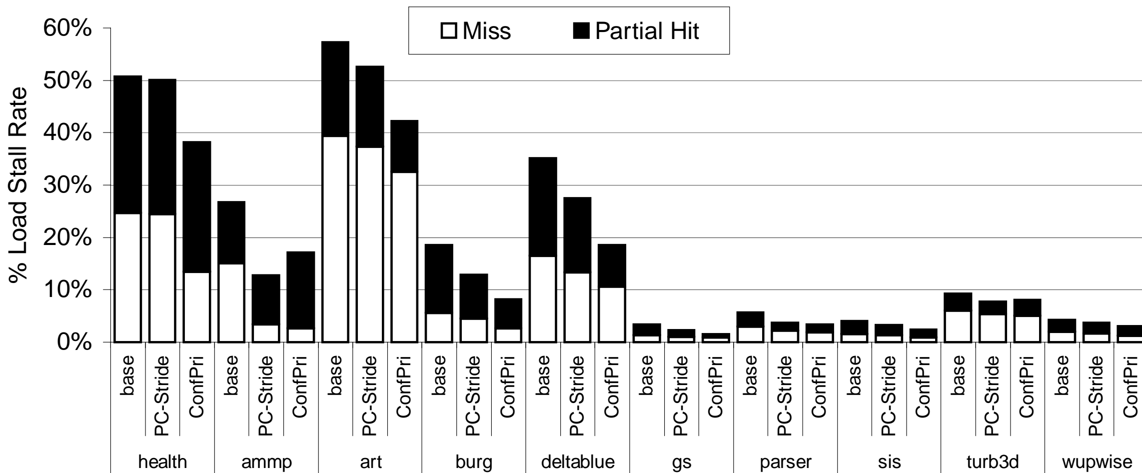

Figure 7: Load Stall Rate. This is the percent of all dynamic loads that stall for one or more cycles when considering the different stream buffer architectures. These are references that are not full cache hits and are not full stream buffer hits. The percent of load stalls are broken down into those that are complete misses and those that are partial hits.

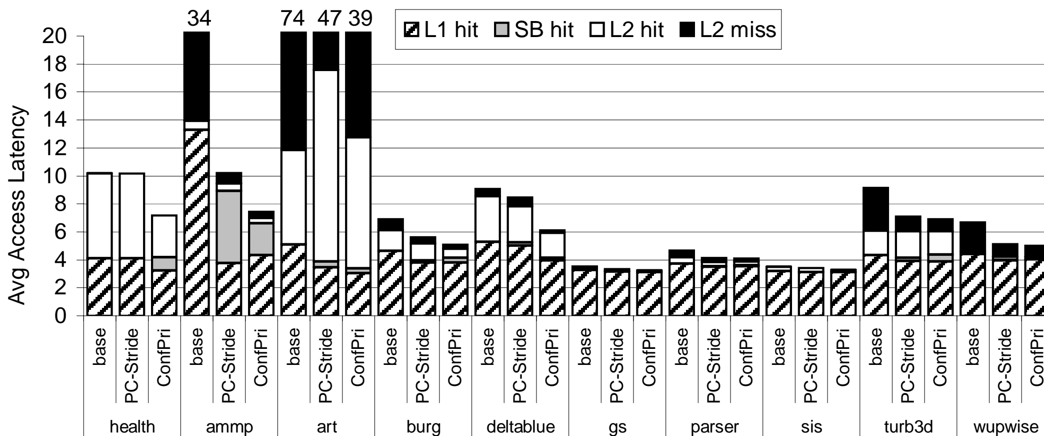

Figure 8: Average latency of a load in cycles for the different architectures broken down by where the time is spent. The L1 access time is 3 cycles. Partial hits add to the access time of the level that they hit in.

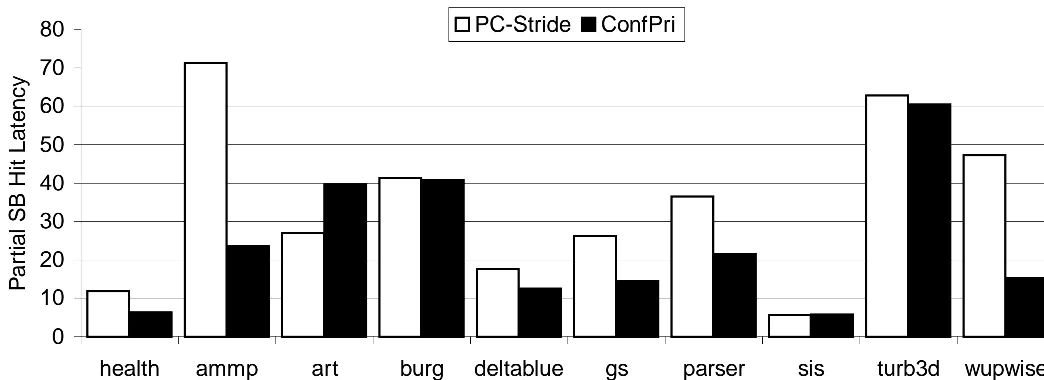

Figure 9: Average partial stream buffer hit latency. This is average latency of stream buffer tag hits that still have the prefetch outstanding. These hits therefore do not completely hide the memory latency.

waiting on the stream buffer partial-hits. Ammp has a significant number of prefetch hits that need to go all the way out to main memory to retrieve their blocks, and as described above the prefetches are not occurring early enough to cover the full latency.

Provided earlier in Figure 6 is the percent of all stream buffer prefetches that successfully cover all of the latency associated with that load (full hits), versus the fraction that only cover some of the latency (partial hits). According to Figure 6, health and ammp incurred a significant number of partial stream buffer hits. To discover the effects this has on performance, we analyzed average partial stream buffer hit latencies in Figure 9. We found that with health there is very little residual latency after prefetching. Even though the prefetcher cannot cover the full memory latency, it is able to cover most of it. Hence the partial hit latency has very little impact on the performance. As mentioned above, the only benchmark with significant partial hit latencies is ammp. Still, in comparison to PC-Stride, the prefetches are significantly more timely for ammp due to the priority scheduling.

6.1.3 Memory Controller and Bus Utilization

Figure 10 shows the percent of bus utilization for both the bus from the L1 to the L2, and the bus from the L2 to main memory. The total amount of L1 bus utilization is only increased significantly for two programs with our technique, health and ammp. However, we saw in Figure 4, these bus utilization increases are more than justified for the amount of speedup achieved. In fact, for three of the programs, art, turb3d, and wupwise, our combination of techniques simultaneously achieves *both* speedup and reduces bus utilization over PC-Stride techniques.

Figure 11 shows the percent of useless prefetches for the different configurations examined. These are the prefetches that never end up being used by the processor. This graph shows that in some cases the accuracy of the new prefetcher more than doubles that of PC-Stride. This can be attributed to using the confidence counters to guide allocation. This 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. We found 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 with limited loop unrolling, and instead use the stream buffers to hide the load latency. As expected, in cases where the Confidence Priority scheme is significantly better than the PC-Stride scheme in terms of wasted prefetches, such as art, deltablue, turb3d and wupwise, the bus is utilized at the same or a higher level of efficiency.

In order to gain more insight into the pressure placed on the memory subsystem (i.e. bus, caches, MSHRs

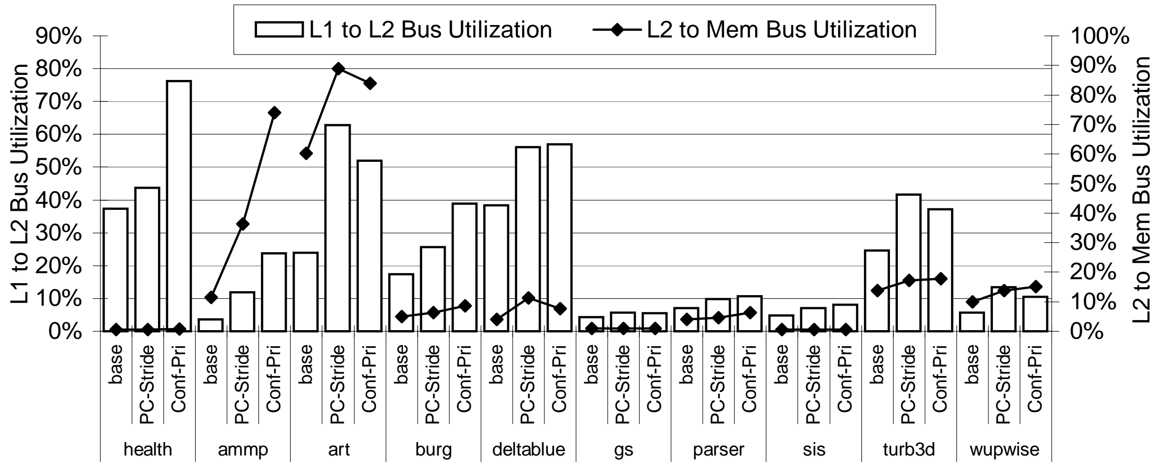

Figure 10: The percent of cycles the L1-L2 bus and the L2-Mem bus were busy.

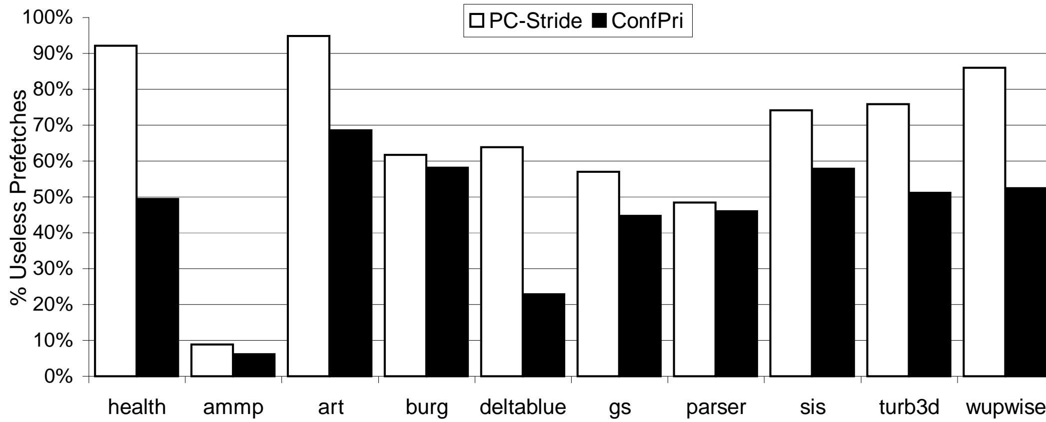

Figure 11: Wasted prefetches. This is the number of prefetches not used by the processor divided by the number of prefetches made.

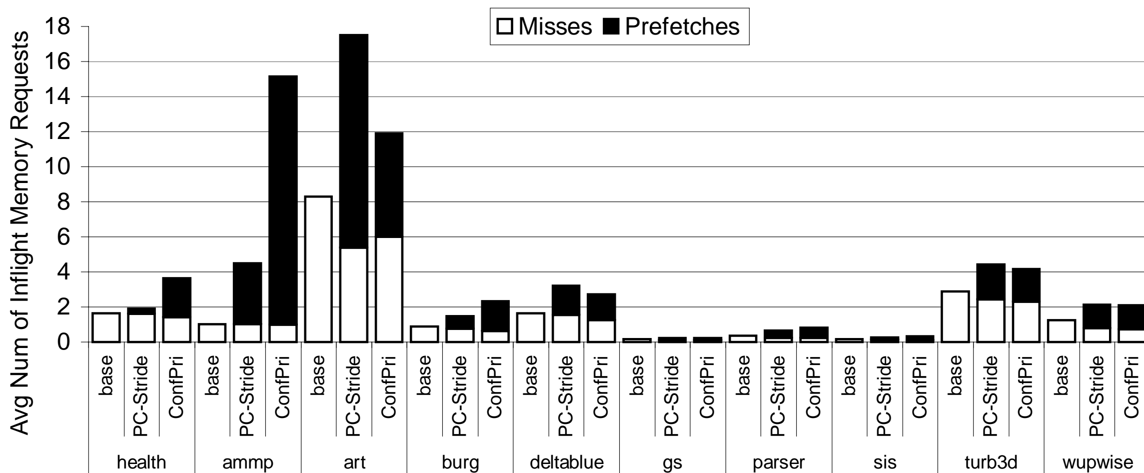

Figure 12: Average number of in-flight memory requests per cycle.

etc.), we looked at the number of in-flight memory requests at each clock cycle. Figure 12 shows the average number of memory requests divided into demand cache misses and prefetches. The maximum number of in-flight L1 misses range from 6 for `sis` to 24 for `art`. The maximum number of prefetches for PC-Stride vary between 13 for `sis` and 32 for `art`. For our Confidence Priority configuration, these numbers fall between 14 and 32 for the same programs. For PC-Stride, at any given cycle the maximum combined number of outstanding prefetch and demand misses deviate between 16 for `sis` and 60 for `art`. For the Confidence Priority, the maximum number of combined in-flight memory requests is 14 for `sis` at one end of the spectrum and at 68 for `turb3d` at the other end. `Art` exhibits an interesting behavior where the average number of L1 misses increases with the Confidence Priority scheme as opposed to the PC-Stride technique. This can be attributed to the significant speedup (40%) achieved over the PC-Stride technique. The shorter execution time causes the misses to cluster together and increases the average number of L1 misses. On average, except for `art`, `deltablue` and `turb3d`, programs have more in-flight prefetches with Confidence Priority over PC-Stride due to the clustering effect of shorter execution time and higher prefetch accuracy. As the stream buffer entries are used, they become available for more prefetches. Consequently, higher stream buffer hit rates result in higher in-flight prefetch requests.

6.2 Sensitivity of Results

We evaluated the performance of our best performing technique across a range of stream buffer configurations. These results are summarized in Figure 13. The first bar in each group corresponds to the baseline Confidence Priority prefetching architecture with 8 stream buffers each having 4 entries (sb8x4). This architecture can issue up to 1 prediction and 1 prefetch per cycle. Results are shown varying the number of stream buffers (8, 32 and 128). The last two bars show increasing the maximum number of potential predictions and prefeches allowed per cycle to 2 and then 4 for the 128 stream buffer configuration. For these last two bars, the L1-L2 and L2-Memory bus bandwidths are adjusted by 2 or 4 times with the prefetch capabilities to expose more available performance. Half of the programs show considerable speedups with increased prefetch bandwidth. As mentioned in Section 6.1.1, programs with highly predictable streams gain the most benefit from increased prefetch bandwidth.

We also examined varying the number of stream buffer entries from 2 to 16 for the same configurations shown in Figure 13, but do not graph their results. We found that only two programs (`turb3d` and `wupwise`) significantly benefit from the lengthening of stream buffers over the default length of 4. These two programs have access patterns that are easy to predict far in advance (Figure 11), combined with still uncovered latency (Figure 8), and available L1 bandwidth (Figure 10).

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 14, where we look at the performance when increasing the cache size to 64K 4-

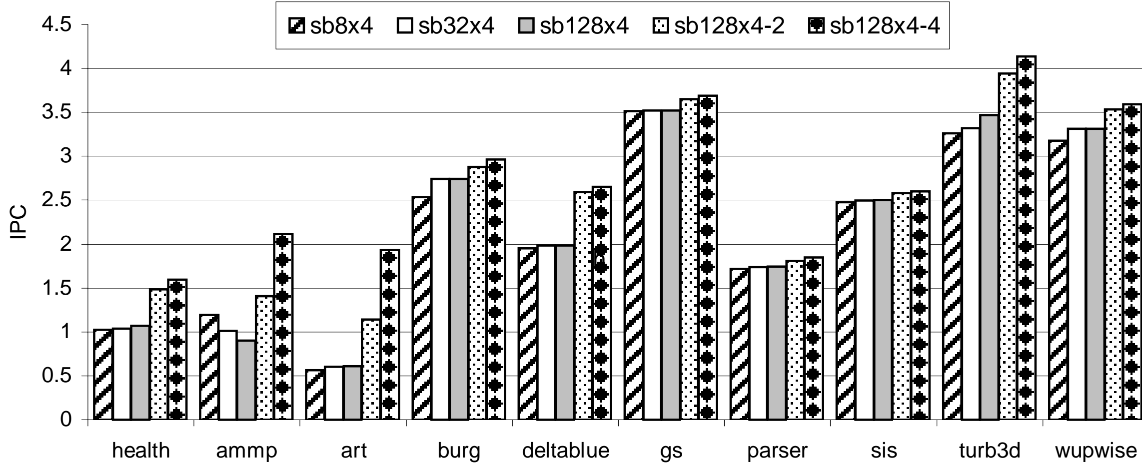

Figure 13: Performance results showing different stream buffer configurations using our confidence allocation and priority scheduling technique. The first bar in each group is the baseline prefetching architecture with 8 stream buffers each having 4 entries, and it can issue up to 1 prediction and 1 prefetch per cycle. For each bar titled sbMxN[-P], M indicates the number of stream buffers and N denotes the number of entries in each stream buffer. P implies the maximum number of predictions and prefetches allowed per cycle. If not shown, P is assumed to be 1.

way and 128K 4-way cache. It can be seen that the speedup obtained over PC-Stride is still seen over a reasonable set of cache configurations for the programs we examined. We also investigated the performance obtained by simply using a larger L1 data cache for the baseline architecture without prefetching to determine if the larger cache can better capture the working set, or if it is better to use the hardware resources for prefetching. Results in Figure 15 indicate that in most cases the performance of our PSB architecture with a 32K 4-way data cache is comparable to 512K 4-way cache results. For some of the programs, the working set does not even fit into a 512K cache.

Finally, we performed simulations to measure the sensitivity of our architecture to various predictor access latencies ranging from 1 cycle to 3 cycles. All the results shown in the figures were simulated with a single cycle predictor access. Although not shown here, results for varying predictor access latency indicate that the predictor can take 3 cycles to access without affecting performance, assuming the predictor is pipelined.

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. Previous stream buffer architectures were limited to streaming only address patterns with a fixed stride [16], which limits their benefit for commercial pointer-based applications. To go beyond this limit

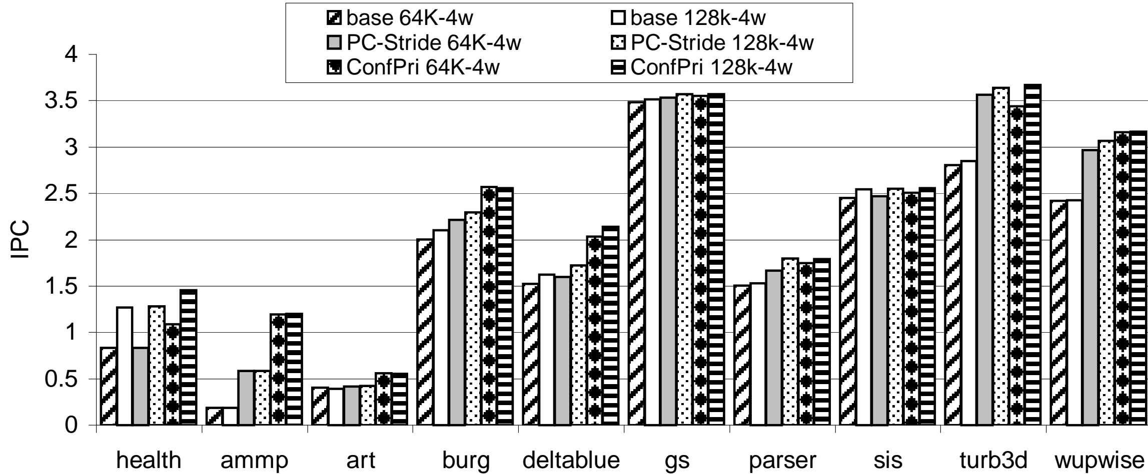

Figure 14: Performance results for increased cache size.

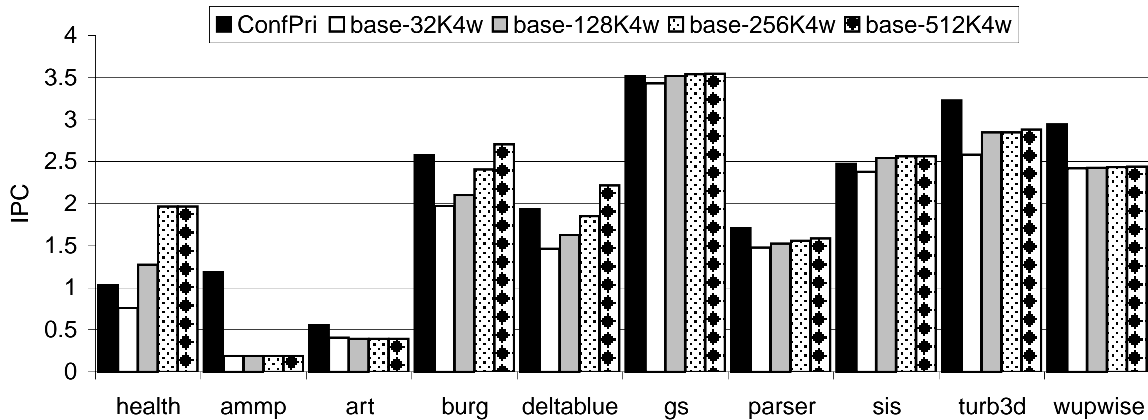

Figure 15: Performance results for default Confidence Priority configuration using a 32K 4-way cache in comparison to the baseline (no prefetching) architecture with a 32,128,256, and 512 4-way caches. The results show the trade off between utilizing area for stream buffers versus increasing the size of the L1 data cache.

we presented a new stream buffer architecture, Predictor-Directed Stream Buffers, that allow stream prefetching of any sort of predictable address patterns.

While Predictor-Directed Stream Buffers can be used in conjunction with any address prediction scheme, we examine them using a Stride-Filtered Markov predictor. The SFM predictor captures complex sequential, stride, and pointer behaviors as well as tightly integrated combinations of those. We proposed a modified version of the Markov part of this predictor, which we call a Differential Markov predictor, whose table data size was only 5 Kbytes for the results we presented in this paper.

We show two problems that arise when allocating resources for Predictor-Directed Stream Buffers, allocation of stream buffers and sharing of memory bandwidth. We present two confidence-based techniques, allocation

filtering and priority scheduling, that overcome these problems and allow the stream buffers to perform efficiently. For the applications we examined, Predictor-Directed Stream Buffers provided a 75% speedup on average over no prefetching, and 23% average speedup over the best performing prior stream buffer architecture.

The stride-filtered markov predictor we use has the potential to achieve a 74% prediction accuracy when predicting the L1 misses. But, the baseline prefetching architecture (8 stream buffers) only covers (prefetches) 28% of the L1 base case misses over the set of benchmarks studied. We showed that if we can increase both the numbers of streams (128 stream buffers) that can be prefetched and the bandwidth (up to 4 prefetches per cycle) then the coverage can be increased to 45% of the cache misses. This aggressive design achieved a 49% speedup over our 8 stream buffer PSB architecture, and resulted in a 178% speedup over the baseline no prefetching architecture. The other reason for the lower coverage is that the stream buffers are using the predictor speculatively building prediction upon prediction, which will result in lower accuracy. Related to this, we've seen that pointer transitions guarded by conditional branches can end up choosing the incorrect Markov predictor entry for some predictions, because there is no branch information guiding the speculative prediction stream. Therefore, we are examining the incorporation of branch history information into the predictor in order to improve the stream buffer prefetch coverage.

Considering the fact that 4GHz processors are currently being manufactured and tested, memory latencies are expected to be in the several hundred to a thousand cycles range. Our PSB architecture can successfully hide the memory latency for data that is stored in on-chip caches (L2 and L3), where the miss stream can compactly be represented in a prediction buffer. This fits in well with recently proposed architectures such as Solihin et. al's [43], where PSB would focus on eliminating latency due to data that exhibits temporal locality within the on-chip caches, and their processor in memory approach would target misses that go off-chip.

Acknowledgments

We would like to thank the anonymous reviewers for providing very useful comments. 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. Annavaram, J. Patel, and E. Davidson. Data prefetching by dependence graph precomputation. In *28th Annual International Symposium on Computer Architecture*, June 2001.

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

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

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

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

[13] J. Collins, D. Tullsen, H. Wang, and John P. Shen. Dynamic speculative precomputation. In *34th International Symposium on Microarchitecture*, December 2001.

[14] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y. Lee, D. Lavery, and John P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *28th Annual International Symposium on Computer Architecture*, June 2001.

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

[16] 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.

[17] K. Farkas and N. Jouppi. How useful are non-blocking loads, stream buffers and speculative execution in multiple issue processors? In *Proceedings of the First International Symposium on High-Performance Computer Architecture*, pages 78–89, January 1995.

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

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

[20] 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.

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

[22] 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.

[23] A. Lai, C. Fide, and B. Falsafi. Dead-block prediction and dead-block correlating prefetchers. In *28th Annual International Symposium on Computer Architecture*, June 2001.

[24] C.K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *28th Annual International Symposium on Computer Architecture*, June 2001.

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

[26] A. Moshovos, D. Pnevmatikatos, and A. Baniasadi. Slice processors: An implementation of operation-based prediction. In *International Conference on Supercomputing*, June 2001.

[27] 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.

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

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

[30] G. Reinman, B. Calder, and T. Austin. Optimizations enabled by a decoupled front-end architecture. *IEEE Transactions on Computers*, 50(4), April 2001.

[31] G. Reinman, B. Calder, and T. Austin. High performance and energy efficient serial prefetch architecture. In *4th International Symposium on High Performance Computing*, May 2002.

[32] 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.

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

[34] A. Roth and G. Sohi. Speculative data-driven multithreading. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, January 2001.

[35] A. Roth, C. B. Zilles, and G. S. Sohi. Micro-architectural miss/execute decoupling. In *International Workshop on Memory access Decoupled Architectures and Related Issues*, October 2000.

[36] S. Sair, T. Sherwood, and B. Calder. Quantifying load stream behavior. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture*, February 2002.

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

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

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

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

[41] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

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

[43] Y. Solihin, J. Lee, and J. Torrellas. Using a user-level memory thread for correlation prefetching. In *29th Annual International Symposium on Computer Architecture*, May 2002.

[44] Y. Song and M. Dubois. Assisted execution. Technical Report CENG 98-25, University of Southern California, October 1988.

[45] D.M. Tullsen, S.J. Eggers, and H.M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, June 1995.

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

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

[48] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *28th Annual International Symposium on Computer Architecture*, June 2001.