In Proceedings of the 8th International Symposium on High Performance Computer Architecture (HPCA), February 2002.

Quantifying Load Stream Behavior

Suleyman Sair Timothy Sherwood Brad Calder

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

Abstract

The increasing performance gap between processors and memory will force future architectures to devote significant resources towards removing and hiding memory latency. The two major architectural features used to address this growing gap are caches and prefetching.

In this paper we perform a detailed quantification of the cache miss patterns for the Olden benchmarks, SPEC 2000 benchmarks, and a collection of pointer based applications. We classify misses into one of four categories corresponding to the type of access pattern. These are next-line, stride, same-object (additional misses that occur to a recently accessed object), or pointer-based transitions. We then propose and evaluate a hardware profiling architecture to correctly identify which type of access pattern is being seen. This access pattern identification could be used to help guide and allocate prefetching resources, and provide information to feedback-directed optimizations.

A second goal of this paper is to identify a suite of challenging pointer-based benchmarks that can be used to focus the development of new software and hardware prefetching algorithms, and identify the challenges in performing prefetching for these applications using new metrics.

1 Introduction

One of the most important impediments to current and future processor performance is the memory bottleneck. Processor clock speeds are increasing at an exponential rate, much faster than DRAM access time improvements [21]. Cache memory hierarchies and data prefetching are the two primary techniques used in current processors to try and hide or eliminate memory latency. Memory latency can be removed if data is found to be in the cache. For data not in the cache, data prefetching is used to hide the latency by beginning the fetch before it is needed.

Several models 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 approaches use locality analysis to insert prefetch instructions, showing significant improvements [14]. Hardware-based prefetching approaches are able to dynamically predict address streams and prefetch down them in ways that may be hard to do using compiler analysis. In [18], we presented a prefetching architecture that uses a predictor-directed stream buffer to prefetch down data miss streams independent of the instruction stream being executed. Other hardware schemes attempt to pre-compute the computation kernel on a separate thread or co-processor to reduce the memory latency [1, 6, 12].

We first show how to classify load miss streams into different classes based on their miss access patterns. We show for a large number of programs what types of accesses are causing misses so that they may be targeted for future research. We further show how this classification can be done efficiently in hardware with a high degree of accuracy, so that architectural structures such as the caches or prefetching engines can be made access pattern aware. We classify these loads into four types of access patterns or streams – (1) next-line, (2) stride, (3) same-object (additional misses to a recently referenced heap object), or (4) pointer-based misses.

Out of these four types of cache miss streams, pointer-based streams can be the most difficult to eliminate using existing hardware and software prefetching algorithms. To better understand the behavior of these loads and their applications we examine two new metrics. The first metric, *Object Fan Out*, is used to quantify the number of pointers in an object that are transitioned and frequently miss in the cache. The second metric, *Pointer Variability*, quantifies how many pointer transitions are stable versus how many are frequently changing. A pointer transition is a load that loads a pointer. Pointer variability shows how many times a pointer transition for a given *address* loads a pointer different from the last pointer that was loaded from that address. Programs with low object fan out and pointer variability will be much easier to prefetch, in comparison to programs that have high object fan out and variability, and we show that a large percentage of misses in real programs fall into this second category.

An additional goal of this paper is to compare the behavior of Olden, SPEC 2000, and set of additional pointer-based benchmarks. This is to identify a suite of challenging pointer-based benchmarks that can be used to focus the development of new software and hardware prefetch algorithms.

Program	Description
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 set architecture.
deltablue	A constraint solution system implemented in C++. It has an abundance of short lived heap objects.
dot	Dot is taken from the AT&T's GraphViz suite. It is a tool for automatically making hierarchical layouts of directed graphs. Automatic generation of graph drawings has important applications in key technologies such as database design, software engineering, VLSI and network design and visual interfaces in other domains.
equake	Equake is from the SPEC 2000 benchmark suite. The program simulates the propagation of elastic waves in large, highly heterogeneous valleys, such as California's San Fernando Valley, or the Greater Los Angeles Basin. The goal is to recover the time history of the ground motion everywhere within the valley due to a specific seismic event. Computations are performed on an unstructured mesh that locally resolves wavelengths, using a finite element method.
mcf	Mcf is from the SPEC 2000 benchmark suite. It is a combinatorial optimization algorithm solving a minimum cost network flow problem.
sis	Synthesis of synchronous and asynchronous circuits. It includes a number of capabilities such as state minimization and optimization. The program has approximately 172,000 lines of source code and performs a lot of pointer arithmetic.
vis	VIS (Verification Interacting with Synthesis) is a tool that integrates the verification, simulation, and synthesis of finite-state hardware systems. It uses a Verilog front end and supports fair CTL model checking, language emptiness checking, combinational and sequential equivalence checking, cycle-based simulation, and hierarchical synthesis.

Table 1: Description of pointer-based benchmarks used.

2 Methodology

We make use of both profiling and detailed cycle accurate simulation in this study. When performing profiling we use Compaq's ATOM [20] tool, to gather miss rates and perform base line classifications.

The simulator used in this study was derived from the SimpleScalar/Alpha 3.0 tool set [2], 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 from the complete SPEC 2000 integer benchmark suite, selected SPEC 2000 floating point benchmarks, the popular programs from the Olden benchmark suite, and a set of other pointer intensive programs. The pointer intensive programs we will examine in detail are described in Table 1. All programs were compiled on a DEC Alpha AXP-21264 processor using the DEC FORTRAN, C or C++ compilers under OSF/1 V4.0 operating system using full compiler optimization (`-O4 -ifo`).

2.1 Baseline Architecture

Our baseline simulation configuration models a next generation out-of-order processor microarchitecture. We have 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 load classification speedups we report are from eliminating those load memory latencies and not from compensating for a conservative memory disambiguation policy, we implemented perfect store sets [5]. Perfect store sets cause loads to only be dependent on stores that 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.

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, and 2-FP MULT/DIV. The latencies are: ALU 1 cycle, MULT 3 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 [13] 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. The L1 instruction cache is a 32K 2-way associative cache with 32-byte lines. The baseline results are run with a 32K 2-way associative data cache with 32-byte lines. A 1 Megabyte unified 4-way L2 cache is simulated with 64-byte lines. The L2 cache has a latency of 12 cycles. 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.

3 Prefetching Focused at the Different Load Stream Classifications

In this section we categorize and describe prior software and hardware prefetching research into the classes of next-line, stride, same-object, and pointer traversals. This classification corresponds to an increasing implementation complexity of hardware prefetching techniques.

3.1 Next-Line

The simplest form of prefetching is to prefetch the next cache block that occurs after a given load. This form of prefetch is very accurate, since programs have a lot of spatial locality.

Next-Line Prefetching (NLP) was proposed by Smith [19], where each cache block is 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.

Jouppi introduced *stream buffers*, as a high latency hiding form of a next-line prefetching architecture [9]. The stream buffers follow multiple streams prefetching them in parallel and these streams can run ahead independent of 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.

3.2 Stride-based Prefetching

A logical extension of next-line prefetching is *stride-based prefetching*. This scheme allows the prefetcher to eliminate miss patterns that follow a regular pattern but access non-sequential cache blocks. This type of access frequently occurs in scientific programs using multidimensional arrays.

Palacharla and Kessler [15] suggested a *non-unit stride* detection mechanism to enhance the effectiveness of stream buffers. This technique uses a *minimum delta* non-unit detection scheme. With this scheme, the dynamic stride is determined by the minimum signed difference between 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.

Farkas et. al. [7] 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. A 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.

3.3 Same Object Prefetching

Programs make use of different types of data structures to accomplish their final goal. Often times, logically related data are grouped together into an *object* to enhance semantics. The amount of data located inside an object does not always fit into a single cache block, and accesses to various parts of the same object can cause multiple cache misses. To eliminate these incidental misses, one could trigger the prefetch of the whole object once a miss occurs to data within the object. This could require the prefetching algorithm to know/predict the size of an object.

Zhang and Torrellas [22] recognized the benefit of grouping together fields or objects that are used together, and prefetching these all together as a prefetch group of blocks. They examined using user added grouping instructions that allowed the user to group together fields/objects that should be prefetched together. These groupings are then stored in a hardware buffer, and as soon as one of them is referenced and misses, all the cache blocks in the group are prefetched.

3.4 Pointer-Based Prefetching

As logically related data is collected into an object, objects that are related are also connected to each other via pointers. *Pointer-based prefetching*, either predicts or accesses these pointer values to prefetch the next object that is likely to be visited after the current one.

The inherent dependency between neighbor objects limits the amount of latency that can be hidden by the prefetching algorithm. This is known as the *pointer-chasing problem* [10]. The imposed serialization of object accesses constrain the prefetcher from running enough ahead of the execution stream to hide the full memory latency.

Luk and Mowry [10] examined prefetching for Recursive Data Structures (RDS). They examined the phenomenon of using pointer chaining for prefetching to hide latency. Greedy Prefetching was used to prefetch down all the pointers in a given heap object. They also examine adding *jump-pointers* to hook up a heap object X to another heap object Y that occurs earlier in the pointer chain, by adding an explicit jump-pointer from $YtoX$. This approach can hide more latency than their demand based greedy algorithm, but comes at a cost of adding jump-pointers into their structures. In addition, this could potentially perform badly if the structure of the RDS changes radically between traversals over the structure.

Roth et. al. [16] propose analyzing the producer-consumer relationship among loads to alleviate the effects of the pointer-chasing problem. In this scheme, load instructions that produce object addresses are linked together to facilitate a prediction chain. Prefetches read address values from memory and initiate another prefetch using the value just prefetched as an address. They examine prefetching one iteration ahead of the current execution to reduce the number of useless prefetches. Furthermore, Roth and Sohi [17] extend the jump-pointer prefetching technique by providing hardware, software and cooperative schemes to facilitate linking objects together. These different techniques provide a variety of trade-off points between prefetch accuracy and prefetching overhead.

Markov prefetching has been proposed as an effective technique for correctly predicting pointer-based loads [8]. When a cache miss occurs, the miss address would index into a Markov prediction table that provides the next set of possible cache addresses that have followed that miss address in the past. After these addresses are prefetched, the prefetcher stays idle until the next cache miss.

Recently we proposed a decoupled architecture for prefetching pointer-based miss streams [18]. We extended the stream buffer architecture proposed by Farkas et. al. [7] to follow prediction streams instead of a fixed stride. Our *Predictor-Directed Stream Buffer* (PSB) architecture uses a Stride-Filtered Markov predictor to generate the next addresses to prefetch. Predictor-directed stream buffers are able to achieve timely prefetches, since the stream buffers can run independently ahead of the execution stream, filling up the stream buffer with useful prefetches. Different predictors can be used to direct this architecture making it quite adept at finding both complex array access and pointer chasing behavior over a variety of applications. The predictor-directed stream buffer achieved a 30% speedup on pointer-based applications using only a 4 Kilobyte Markov table along with a 256 entry stride prediction table.

4 Load Miss Stream Classification

In section 3 we described the types of misses we wish to classify and how they behave in relationship to prior prefetching research. In this section we start by defining the different load miss models and show how the misses for many different types of programs are classified into these models. We then present a hardware technique for quickly and efficiently classifying cache misses for the purpose of guiding dynamic prefetching.

4.1 Miss Classes Defined

As described in section 3, there are four major types of memory access behavior prevalent in most programs that can be captured by hardware. Listed in order of increasing complexity they are: next-line, stride, access within an object, and dereferencing of pointers.

Next-line accesses are the simplest to capture with hardware, a simple stream buffer is very efficient at capturing this type of behavior. The stream buffer can identify accesses to sequential cache blocks and use this information to fetch sequentially down the stream. While this type of access is very simple, it is also very common in a multitude of applications. We classify a cache miss as being a next-line access if it is an access to a cache block that is adjacent to a cache block that was recently fetched.

Stride accesses are the next easiest to capture in hardware and many different prefetching architectures exist to capture this type of behavior. Farkas et. al. [7] showed that the most efficient way to capture this sort of behavior is by examining access patterns on a per static load basis. We use this observation to help us define stride access behavior. We define a cache miss to be stride miss if the same stride has been seen twice in a row for the static load that performed the access.

So far we have concerned ourselves with regular access patterns, the type that may be commonly found in programs dominated by large multidimensional arrays. The next two classes may not fall into this category. The class of misses, which we call Same-Object, are non-sequential, non-striding accesses going within a single object. We define a *Same-Object* cache miss as a miss to an object that has already had a miss recently. These misses may possibly be prevented if whenever we access an object we fetch the whole object, or at least those cache blocks of the object that will soon be referenced. These misses can also be targeted by field reordering [4].

The final hardware classification that we make is the Pointer class. The *Pointer* class represents misses to objects that are accessed via the dereference of a pointer. Pointer misses can constitute a large portion of total cache misses due to the accesses having very little conventional spatial locality. This has led to many recent hardware and software schemes that attempt to capture their behavior for the purpose of prefetching. These misses can be targeted by the software techniques of object reordering [11], smart object placement [3], software prefetching [10], or hardware schemes such as jump pointer prefetching [16, 17] and predictor-directed stream buffers [18].

We classify all cache misses into one and only one of these categories. If there is a cache miss that can fit into more than one category, such as loads with a stride of 1, we classify them into the simplest category possible, which in this case would be next-line.

In the description of the classifications please note the use of the term “recently”. In order to capture the recent behavior of the loads we use a profiling technique called *Windowing*. We build a window of the last cache misses and loads, and use only this information when classifying subsequent loads. Using windowing prevents all loads being classified as “next-line”, since only a fixed amount of history is kept track of. The window models the recent working set of load misses that could potentially trigger a prefetch of the load that missed. The window size is limited, because the prefetched block could only reside in a prefetch buffer or cache without being used for a window of time before being evicted. For the results presented in this section we capture the last 200 cache misses and the last 500 loads for pointer tracking.

4.2 Miss Classification Results

The first thing to look at before we begin discussing the classification of loads is the cache miss rates for the various programs. Tables 2, 3, and 4 show the data input used to run each program, the L1 and L2 cache miss behavior, and the percent of loads executed by each program. The L1 and L2 cache misses are both in terms of the average number of cache misses that occurred per 1K of executed instructions.

The programs consist of the full set of SPEC 2000 integer programs, a subset of the SPEC 2000 floating point programs that have been used in recent prefetching and precomputation papers, the Olden benchmark suite, and a set of pointer intensive programs. We chose the Olden benchmarks that have shown performance improvements from prior prefetching papers.

We now show the results of applying the classification technique described above over several suites of programs. Figure 1 shows the classification of the loads that miss in the L1 cache for the SPEC 2000 benchmarks. The four programs of interest from the SPEC 2000 suite that have a significant amount of cache misses include `art`, `ammp`, `equake` and `mcf`, and all of these have 80% or more of their misses classified, with `mcf` having the largest pointer behavior of these applications.

The classification of the programs from the Olden suite can be seen in figure 2. `Health` has by far the largest miss rate of all the programs, and is also the most dominated by the pointer behavior. The other programs have less significant miss rates and are more balanced in the types of misses that they exhibit. The one counter example to this is `treeadd` which is dominated by next-line prefetchable structures. All `treeadd` does is allocate a tree in depth-first order, and then traverse the tree in depth-first order. This results in the tree being created where every left child is allocated in memory right after its parent node, and therefore almost all of the cache misses would be covered by next-line prefetching. For that reason, `treeadd` has almost no pointer misses.

Figure 3 shows the same classification technique applied to our own set of pointer intensive benchmarks. As the name suggests, the suite of applications that we have assembled are dominated by pointer behavior. However this pointer behavior is not the only form of misses. There is a mix of access patterns seen, from next-line to pointer behavior, stride and same-object. The benchmark with the highest miss rate, `dot`, is also the program most dominated by pointer behavior. The programs `sis` and `vis` also show very strong pointer behavior.

The ability to do this classification in software is useful to quantify applications allowing researchers to find applications exhibiting certain behavior or to guide profile-directed

benchmark	input	L1 MissPer1kI	L2 MissPer1kI	% Loads
ammp	ref	14.14	8.42	8.29%
applu	ref	28.42	13.57	25.89%
apsi	ref	13.70	3.34	21.34%
art	110	15.04	0.01	4.84%
bzip2	graphic	4.49	0.90	17.48%
crafty	ref	6.08	0.03	22.32%
eon	cook	0.09	0.00	10.06%
equake	ref	44.76	16.29	31.91%
galgel	ref	24.14	2.39	21.64%
gap	ref	1.90	0.88	14.70%
gcc	200	10.30	0.81	21.65%
gzip	graphic	7.47	0.11	17.47%
lucas	ref	0.92	0.47	4.25%
mcf	ref	130.16	91.54	28.34%
mgrid	ref	21.81	6.13	29.34%
parser	ref	13.00	1.92	19.66%
perlbmk	diffmail	4.62	0.08	23.00%
swim	ref	45.31	17.23	19.73%
twolf	ref	19.81	2.45	20.26%
vortex	two	2.38	0.29	21.00%
vpr	place	12.77	1.21	20.81%
wupwise	ref	6.34	2.94	16.56%

Table 2: SPEC 2000 cache miss behavior. The L1 data cache is a 32K 2-way associative cache with 32 byte lines. The L2 is a unified 1 Meg 4-way associative cache with 64 byte lines.

benchmark	input	L1 MissPer1kI	L2 MissPer1kI	% Loads
burg	rrh-vax	15.82	0.89	17.63%
deltablue	long	56.12	1.11	27.93%
sis	markex	4.58	0.04	36.32%
vis	clma	7.67	2.17	19.91%
dot	small	90.03	68.85	32.91%

Table 3: Pointer-based programs cache miss behavior. The L1 data cache is a 32K 2-way associative cache with 32 byte lines. The L2 is a unified 1 Meg 4-way associative cache with 64 byte lines.

benchmark	input	L1 MissPer1kI	L2 MissPer1kI	% Loads
health	5 500 1 1	122.93	0.36	34.86%
mst	1024 1	9.63	5.53	18.19%
perimeter	12 1	13.58	9.42	17.60%
treeadd	20 1	9.20	4.56	21.30%
tsp	100000 1	1.53	0.65	6.94%

Table 4: Olden cache miss behavior. The L1 data cache is a 32K 2-way associative cache with 32 byte lines. The L2 is a unified 1 Meg 4-way associative cache with 64 byte lines.

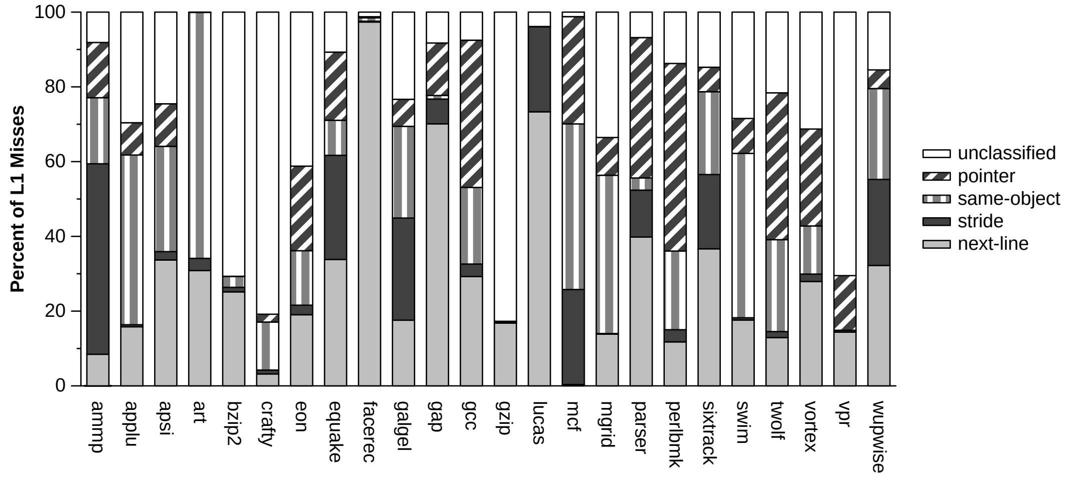

Figure 1: Classification of SPEC 2000 program load misses into the four prefetching models of next-line, stride, same-object, and pointer.

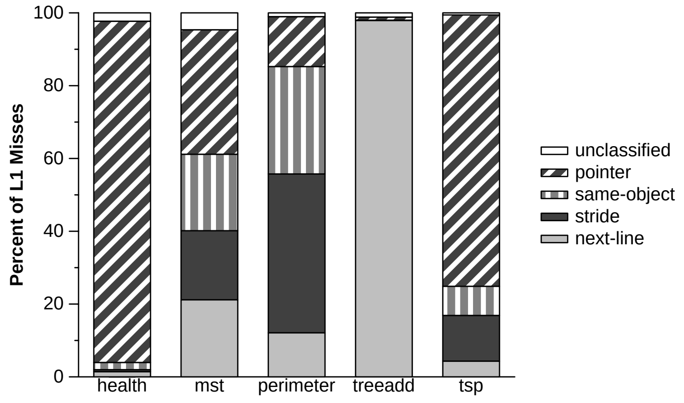

Figure 2: Classification of Olden program load misses into the four prefetching models of next-line, stride, same-object, and pointer.

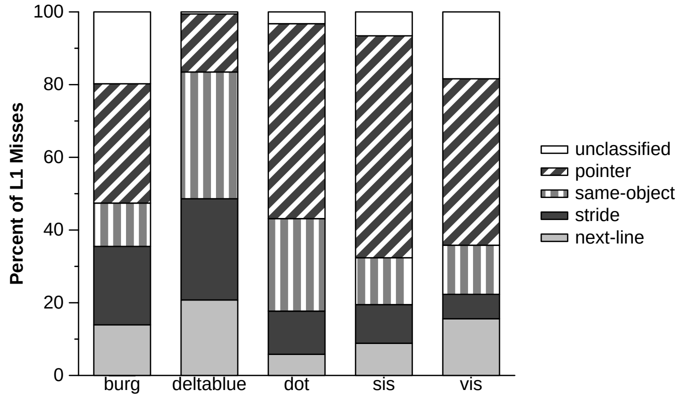

Figure 3: Classification of Pointer program load misses into the four prefetching models of next-line, stride, same-object, and pointer.

optimizations. However, there are still some questions to be answered about those loads that are left unclassified.

4.3 Unclassified Loads

As can be seen in figure 1, some programs have a fair number of cache misses that do not fit into the categories of next-line, stride, same-object or pointer transition. These cache misses have arithmetic operations (not captured by stride) used to calculate their effective addresses. These classifications are not easily captured by existing hardware prefetchers. For these cache misses, we use three additional types of cache miss classifications. To determine the classification we search back over the dependency chain used to calculate the effective address.

We classify cache misses as *Recurrent* if there is an instruction that is inside of a loop that has the same logical register definition as one of its operands, and the operand register being defined has an address stored in it. A load whose effective address is calculated in this manner is producing its effective address each iteration of the loop off of the prior loop's address calculation. Cache misses that are classified as recurrent perform arithmetic operations to produce the effective address not captured by stride prefetching.

A cache miss is labelled as *Base Address* if there is an instruction in the load's dependency chain that uses the same address over and over again in a calculation to produce the load's effective address. This occurs when the calculation for the effective address is performed off of the same base address every loop iteration.

Finally, a load is labelled as *Complex* if it is not recurrent nor base-address, and the load's effective address is calculated from a prior load in the dependency chain, and that prior load's value was an address. The address from that prior pointer load is used in an equation to produce the ef-

benchmark	% Unclassified Recurrent	% Unclassified Base Address	% Unclassified Complex
ammp	6.24%	0.94%	0.95%
applu	14.05%	1.00%	14.54%
apsi	7.91%	3.06%	13.56%
art	0.00%	0.06%	0.02%
bzip2	70.65%	0.03%	0.00%
crafty	79.45%	1.37%	0.01%
eon	23.75%	5.61%	11.85%
equake	10.60%	0.08%	0.02%
fma3d	0.19%	0.03%	1.01%
galgel	9.26%	0.19%	13.89%
gap	8.05%	0.21%	0.00%
gcc	6.52%	0.93%	0.07%
gzip	82.58%	0.13%	0.00%
lucas	1.88%	0.90%	1.08%
mcf	1.19%	0.01%	0.01%
mgrid	17.59%	0.20%	15.76%
parser	6.53%	0.21%	0.05%
perlbmk	12.24%	1.41%	0.09%
swim	13.85%	0.79%	0.11%
twolf	26.76%	0.67%	1.04%
vortex	20.00%	0.78%	0.82%
vpr	29.61%	1.63%	0.03%
vpr	57.53%	5.41%	7.56%
wupwise	1.51%	0.18%	13.77%
burg	19.45%	0.31%	0.00%
deltablue	0.49%	0.07%	0.00%
dot	3.23%	0.02%	0.00%
sis	5.69%	0.85%	0.03%
vis	17.16%	0.38%	0.86%
health	1.58%	0.64%	0.07%
mst	4.55%	0.08%	0.02%
perimeter	1.03%	0.00%	0.00%
treeadd	1.14%	0.01%	0.00%
tsp	0.31%	0.00%	0.25%

Table 5: Detailed classification of unclassified L1 misses.

fective address that missed in the cache.

Table 5 presents a detailed look into load misses that go unclassified as described above. Most of the unclassified misses are recurrent pointer misses, potentially indicating that the loop induction variable is updated in a non-linear fashion. This makes these misses unclassifiable to next-line or stride predictors.

In the remainder of the paper we concentrate completely on the next-line, stride, same-object and pointer classifications, since these are captured by hardware prefetching techniques, and in the next section we will describe an approach for performing these four classifications in hardware.

4.4 Hardware Classification

In order to take advantage of classification we need to provide a way for it to be done efficiently in hardware at run time. To accomplish this we make use of the windowing technique described in section 4.1, along with a very small fully associative buffer. The classification hardware keeps information in the buffer for the last N cache misses and then performs a lookup on its tables during a cache miss. Different types of matches mean different classifications for that load.

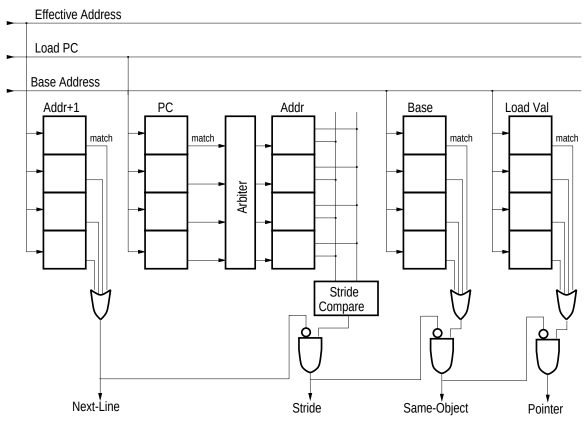

Figure 4: Load Miss Classification Hardware.

Figure 4 shows the proposed classification architecture. The basic structures in the architecture are small CAMs each with an update pointer. Every time there is a cache miss, the CAM is checked for hits for the four different models. This hit information is used to calculate the classification of the cache miss. The structure is then updated at the update pointer and the update pointer is incremented to the next entry. The structure is therefore accessed in two ways, a parallel lookup and a rotating register style update using the update pointer.

The structure to find next-line misses is a small CAM with a list of the past addresses that have missed with one block size added to them. Figure 4 is drawn showing a CAM with 4 entries, while each actual CAM has 32 entries in the architecture we modeled. Whenever a cache miss occurs, we simply check for a match in the CAM. A hit in any of the elements of the CAM indicates that the load was of the next-line type. Then after this information is computed we update the CAM with the most recent address information.

To find stride misses we add a structure that performs a parallel lookup as in the CAM before, but this time we attempt to match the PC of the load instruction rather than the address that is being loaded. If there is a hit in the CAM, the address seen in the slot is output and the two most recent addresses for that load are output and subtracted. The output of this subtraction is then added to the first address and a check for a match is done, with a match indicating a successful classification. This circuit is very similar in behavior to the arbiter used in the issue stage of an out of order processor, but much smaller. It is further simplified by the fact that the operation can be multicycle and pipelined.

Misses to the same heap object can be easily detected using the same type of structure used for next-line detection, but this time we store the base address rather than the possible next-line address. The base address from a cache miss is looked up in the CAM, and a hit indicates that this miss is classified as an access to the same object. On update, each cache miss stores its base address into this CAM.

To detect pointer based loads we add one last small structure, which is another CAM. This CAM is updated with the result of every load using the update pointer associated with that CAM. Because the result of all recent loads are stored in the CAM, loads to pointers are captured. When there is a cache miss we check the CAM to see if we can find the base address of the missing load. If there is a hit then we know that a prior load recently loaded the base pointer for the object that just missed and hence it is a pointer miss.

In using this classification scheme, the first miss to an object for a pointer-based application most likely will be classified as a pointer miss, and all subsequent misses to that same object would be classified as same object misses.

To test this the hardware scheme we compare it against the real classifications that were presented in section 4.2. We use a very small hardware window size, of N=32. This means that we only need 896 Bytes of storage to implement this architecture, and this could be further reduced to around 256 Bytes if partial tags are used with a small hash function.

To evaluate the classification hardware, we would like to predict what the *next* classification will be for a given load. To accomplish this we keep a small direct mapped table, which is indexed by the address of the load. In this table we store the last known classification of the load, as generated by the classification hardware. We then compare this value stored in this table to the true classification of the load. Figures 5 and 6 show the accuracy of this classification prediction mechanism over the pointer and Olden benchmark suites if a prediction table of size 128 is used.

The prediction accuracies show that by using the presented architecture we can correctly classify the majority of cache misses for the applications examined. The programs that the predictive classification had the most trouble with were `burg` and `sis`. For `burg` the predictor is caught jumping between stride and pointer classifications, this is because the application happened to allocate some of its' objects at a fixed stride from each other. This causes stride to be predicted when a stride is done, but pointer is the still the true access type. This is not a problem because either answer is really valid, but it could be fixed with the addition of a small amount of hysteresis. The classification for `Sis` is around 80% because the program performs pointer arithmetic to load some of its data, and these are not accurately classified.

4.5 Performance Results

In order to measure the potential benefit of applying our classification scheme, we ran detailed performance simulations using the SimpleScalar model described in section 2. The goal of these simulation results are to show the potential IPC performance if all of the cache misses are eliminated related to one or more of the prior four types of load miss classifications using the classification hardware presented in the prior section. Figures 7, 8, 9, and 10 show IPC results when we assume perfect load latency for these loads. The first bar shows IPC results for the baseline architecture. The next bar (NL) shows results when loads classified as next-line using our hardware classification architecture are given perfect L1 latency (they do not miss in the cache). The remaining bars show the same optimization applied to loads classified as stride (ST), same-object access (SO), pointer accesses (Pointer), and combinations of different classes. The bar (All) shows the IPC where all loads classified (i.e. next-line, stride, same-object and pointer loads) are assumed to hit in the cache. The last bar (PerfL1) shows the IPC when there are no memory stalls. During simulation, each L1 cache miss

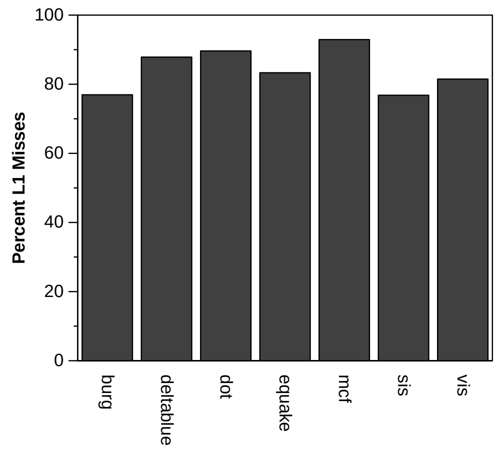

Figure 5: Classification prediction accuracy for the suite of pointer intensive applications.

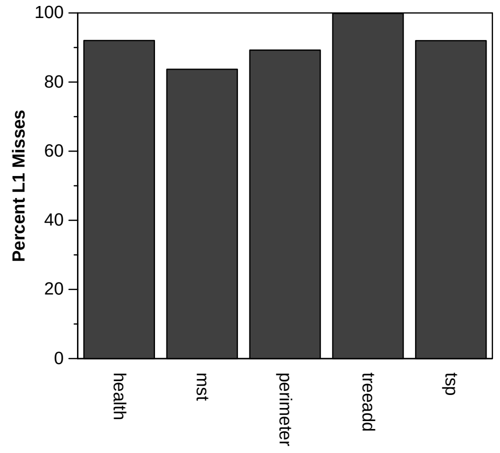

Figure 6: Classification prediction accuracy for the Olden benchmark suite.

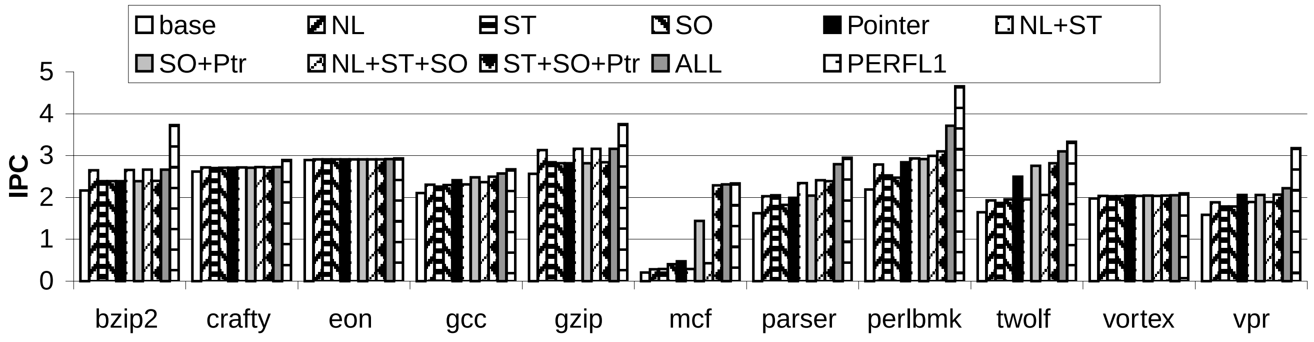

Figure 7: SPEC'CINT00 integer benchmark suite performance results when assigning perfect load latency for loads that match the different classifications.

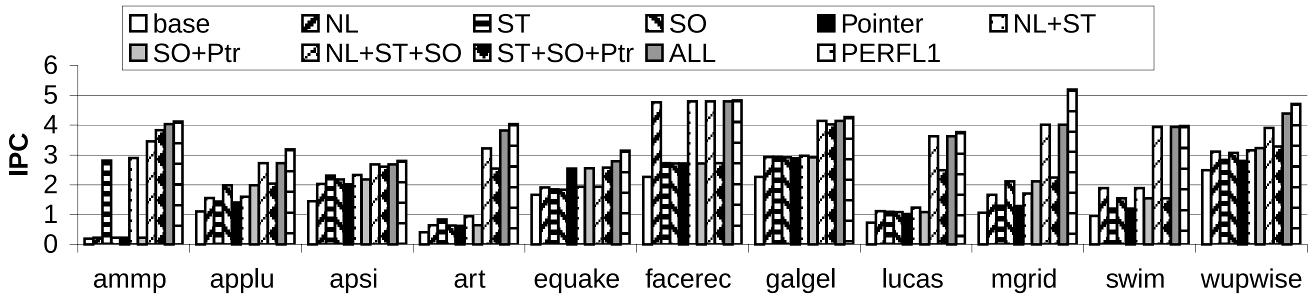

Figure 8: SPEC'CFP00 floating point benchmark suite performance results when assigning perfect load latency for loads that match the different classifications.

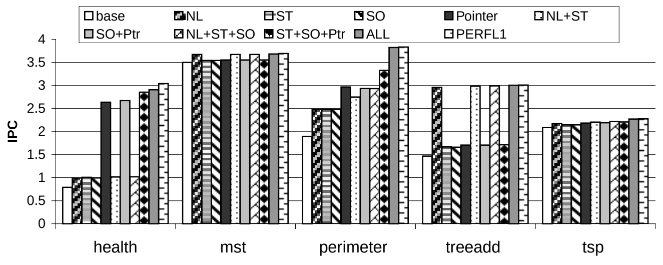

Figure 9: Olden benchmark suite performance results when assigning perfect load latency for loads that match the different classifications.

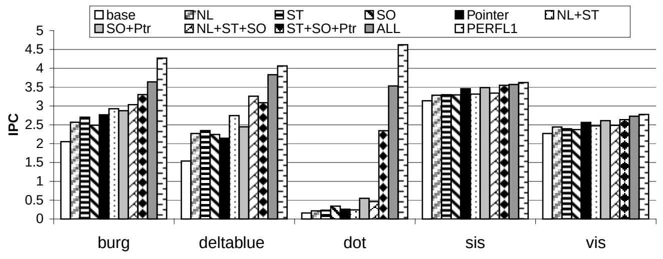

Figure 10: Pointer benchmark suite performance results when assigning perfect load latency for loads that match the different classifications.

is passed to the classification hardware. If the cache miss is classified as one of the types of misses we are eliminating, then the cache access is performed with no latency.

For all of the Olden benchmarks, a single load classification dominates the misses for `health` and `treeadd`. Applying a single optimization targeting specific loads achieves very good results. All of `Health's` important load misses are pointer misses. All of the important misses in `treeadd` are captured by the next-line classification.

Figure 10 presents the results for a suite of pointer-intensive applications. No one particular load class dominates the accesses. In this regard, a prefetching algorithm needs to be able to handle all four of these different classes of loads at the same time in order to provide significant speedups. This is shown in `burg`, `deltablue`, `dot`, `mcf`, and `vis`. This is in stark contrast to most of the Olden benchmarks, in which handling just one of the classifications provides complete speedups.

5 Quantifying Object and Pointer Behavior

One of the most challenging types of access patterns to capture are pointer transition patterns. These are often also the most critical to capture because of the high degree of dependence typically seen between pointer loads, and the poor spatial locality exhibited by this type of access.

There are two major factors that make capturing pointer behavior difficult. Pointer structures often have a high degree of fan out making the path to be traversed more difficult to choose. In addition, pointer transitions can be dynamic by

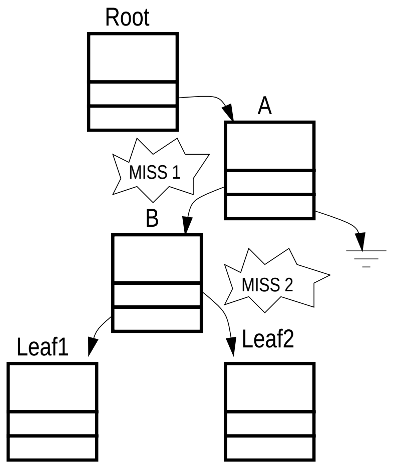

Figure 11: Fan Out Example

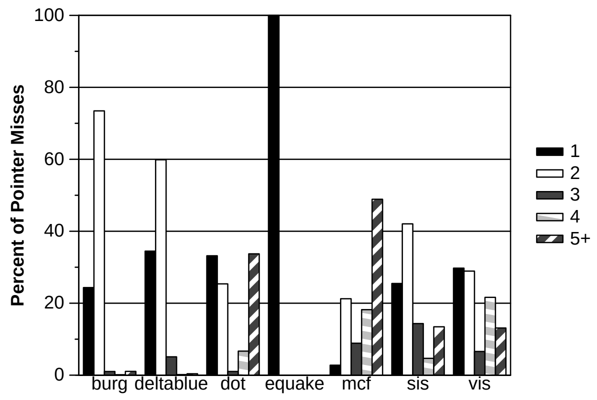

Figure 12: Object fan-out of the pointer-based programs. A histogram of object fan-out is shown for L1 cache misses classified as pointer transitions in section 4.

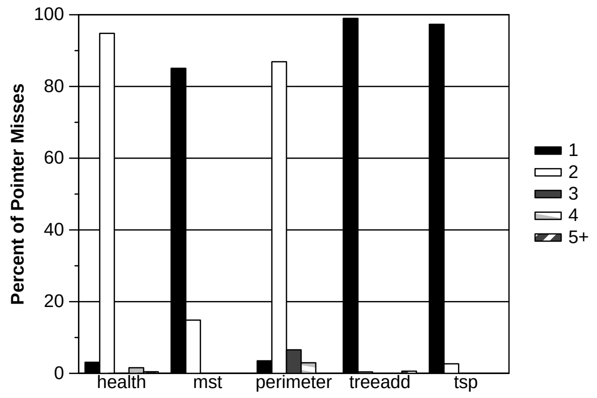

Figure 13: Object fan-out of the Olden benchmark suite. A histogram of object fan-out is shown for L1 cache misses classified as pointer transitions in section 4.

their very nature and can change dramatically through the execution of the program via insertions or deletions to the data structure. Applications that have a high degree of fan-out and pointer transition variability will potentially be harder to accurately prefetch. In this section we provide an analysis of these two factors over our set of pointer based programs and the Olden benchmark suite.

5.1 Object Fan-Out

A given heap object that contains a set of n pointers to other objects, is said to have a *fan-out* of n. For example, a binary tree with a right and left child is said to have a fan-out of 2, while a tree with three child pointers is said to have a fan-out of 3. When calculating fan-out for an object, we only count a pointer to another object as part of the fan-out if it is actually traversed at least once during execution.

Since we are concerned with memory performance, we are interested in pointer transitions from object A to object B that result in a cache miss. In this example, we are concerned with the object fan-out of A, because the fan-out (number of pointer transitions) out of A will influence the ability of the hardware to prefetch the cache miss transition to B. Figure 11 shows an example of this. Suppose that we have the small tree, where A has one `NULL` child and one child transition to B, and node B has two real children. Now suppose that there is a cache miss when the program attempts to transition to node B, noted as $Miss1$ in the diagram. This cache miss will be classified as having a fan-out of 1, because it was the dereference of a pointer from an object (A) with a fan-out of 1. Cache $Miss2$ on the other hand will be noted as having a fan-out of 2, because it comes from the dereference of node B, which has a fan-out of 2.

Now that we have this measure of fan-out, we wish to see how the misses are distributed across objects with different fan-outs. Figures 12 and 13 show the histogram of fan-out misses for both the pointer-based programs we have chosen and the Olden benchmarks. The fan-out results are shown for the L1 cache misses that are classified as pointer transitions in figures 1, 2 and 3. Looking at the graphs in figure 12, the fan-out for *equake* stands out. `Equake` has all of its misses coming from objects with a fan-out of 1, such as a simple linked list. The programs `deltablue` and `burg` are split between objects that have a fan-out 1 or 2 that transition to a miss, while `dot`, `mcf`, `sis`, and `vis` have misses from objects with many transitions to chose from. The dominate fan-out for `dot` and `mcf` is at 5 or greater.

This is in stark contrast to the behavior of the Olden benchmarks seen in Figure 13, where all of the programs are dominated by a single fan-out of either 1 or 2. This shows that the behavior of the Olden benchmarks is dominated by a single simple homogeneous data structure which is not representative of the complexity inherent in the other pointer based applications.

5.2 Variability

Another factor that makes prefetching of pointer structures difficult is the fact that the pointer transitions to other objects changes over the lifetime of the application. In order to understand how the pointer structures change over time we add a new metric called variability.

The *variability* of a pointer in a program is the number of different values (addresses) it has over the life time of the program. In order for a data structure to change, the pointers within the structure must point to different objects. Every time we see one of these changes, we record that it changed. After the program has completed running, we put all of the cache misses associated with a pointer address into a bucket based upon the number of different addresses stored in that pointer (the variability) during execution. From this we make a histogram, which can be seen in figures 14 and 15. This shows the percent of pointer classification L1 misses that had the different degrees of variability.

In analyzing figure 14, `equake` again shows that the pointer transitions do not change during the execution after the initial data structure has been set up. This correlates to the fan-out results in figure 12, which showed each object has only one outgoing edge creating misses and that transition retains its' value throughout the program. `Mcf`, `sis` and `vis` also are interesting to look at as most of the misses are caused by objects with high variability. These programs are difficult to accurately prefetch as the data stream is constantly changing and hence, is highly unpredictable.

Most of the Olden benchmarks data structures remain fairly static with little variability, making them suitable for software based prefetching techniques [10]. The one program that is, at least at the surface, counter to this characterization is `treeadd`. The reason why `treeadd` is shown to have a high degree of variability is that only 3% of all of its cache misses were classified as pointer misses as shown in Figure 2. These 3% of the misses came from a load in malloc which temporarily stores the address of newly allocated memory, and the pointer is overwritten repeatedly.

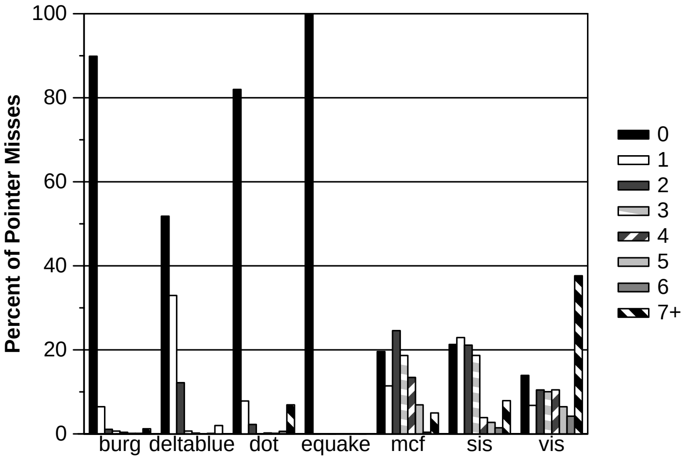

Figure 14: Pointer variability for the pointer-based applications. A histogram of pointer variability is shown for L1 cache misses classified as pointer transitions in section 4.

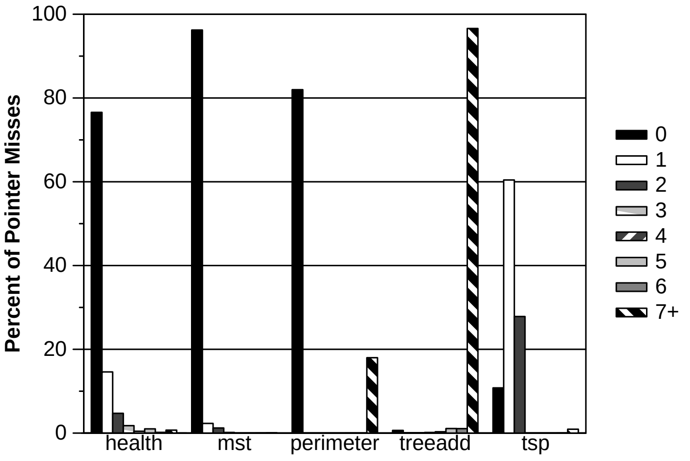

Figure 15: Pointer variability for the Olden benchmark suite. A histogram of pointer variability is shown for L1 cache misses classified as pointer transitions in section 4.

6 Summary

That gap between processor performance and memory latency continues to grow at an astonishing rate, and because of this the memory hierarchy continues to be the target of a great deal of architectural research. In this paper, we present both an analysis of cache miss behavior, to help guide researchers in future cache and prefetching research, and a dynamic hardware technique to perform classification on the fly, which will enable architectural structures to be access pattern aware.

We classify load access patterns into one of four types, next-line, stride, same-object (additional misses that occur to a recently accessed object), and pointer-based transitions. These four access patterns account for more than 90% of all cache misses in the programs we examined. We then show a hardware technique that can detect this behavior using very little on-chip area. The dynamic classification technique presented can accurately predict more the 77% of cache misses as being of the correct type for all programs. On average across all programs, the technique correctly classifies 85% of all misses.

We also evaluate the potential benefit of correctly identifying load classes and the effect of removing their memory latency. This in effect simulates a perfect prefetcher for each class of loads. Our results show that a multi-pronged attack is needed to hide the majority of the memory latency. Prefetching only a single class of load does not provide noticeable benefits for the pointer-based collection of applications we examined. In contrast, removing the latency for only a single load stream classification achieved perfect results for a few of the Olden benchmarks.

In addition to the hardware classification technique presented, we further study those misses identified as pointer-based. Pointer-based misses have become the subject of a great deal of research in recent years and for future research it is important to understand their behavior.

To quantify the behavior of pointer loads, we examined two metrics each weighted by the number of cache misses for pointer-based loads. We use the fan-out metric of objects to quantify the branching factor that data structures have. For a set of programs that are actually used to solve real problems, the fan-out tends to be both large and non-uniform. In contrast, the Olden benchmark suite shows both a very regular and a very small fan-out.

In addition to fan-out, we also examine how often a pointer transition changes over the life time of the application. To track this we keep a list of every pointer in the program and note how many times the pointer's value changes over the execution. We found that while about half the programs simply build and then destroy a large data structure, the other half change their data structures around quite often. This can make prefetching techniques that are based on learning the access pattern much more difficult to implement. The Olden benchmarks showed very little variability in their pointer transitions.

Acknowledgments

We would like to thank Yuanfang Hu for her help in gathering the recurrent, base address, and complex pointer load classification results. In addition, we would like to thank the anonymous reviewers for providing useful comments on this paper. This work was funded in part by DARPA/ITO under contract number DABT63-98-C-0045, a grant from Compaq Computer Corporation, and an equipment grant from Intel.

References

[1] M Annavaram, J. Patel, and E. Davidson. Data prefetching by dependence graph precomputation. In *28th Annual International Symposium on Computer Architecture*, June 2001.

[2] 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.

[3] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, October 1998.

[4] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, May 1999.

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

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

[7] 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.

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

[9] 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.

[10] C.-K. Luk and T. C. Mowry. Compiler based prefetching for recursive data structures. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[11] Chi-Keung Luk and Todd C. Mowry. Memory forwarding: Enabling aggressive layout optimizations by guaranteeing the safety of data relocation. In *ISCA99*, pages 88–99, May 1999.

[12] 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.

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

[14] 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.

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

[16] 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.

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

[18] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. In *33rd International Symposium on Microarchitecture*, December 2000.

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

[20] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 196–205. ACM, 1994.

[21] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. Technical Report CS-94-48, 1, 1994.

[22] Z. Zhang and J. Torrellas. Speeding up irregular applications in shared-memory multiprocessors: Memory binding and group prefetching. In *22nd Annual International Symposium on Computer Architecture*, June 1995.