Neighborhood Prefetching on Multiprocessors Using Instruction History[*]

David M. Koppelman

Department of Electrical & Computer Engineering, Louisiana State University

koppel@ee.lsu.edu

Abstract

A multiprocessor prefetch scheme is described in which a miss is followed by a prefetch of a group of lines, a neighborhood, *surrounding the demand-fetched line. The neighborhood is based on the data address and the past behavior of the instruction that missed the cache. A neighborhood for an instruction is constructed by recording the offsets of addresses that subsequently miss. This* neighborhood prefetching *can exploit sequential access as can sequential prefetch and can to some extent exploit stride access, as can stride prefetch. Unlike stride and sequential prefetch it can support irregular access patterns. Neighborhood prefetching was compared to adaptive sequential prefetching using execution-driven simulation. Results show more useful prefetches and lower execution time for neighborhood prefetching for six of eight SPLASH-2 benchmarks. On eight SPLASH-2 benchmarks the average normalized execution time is less than* 0.9*, for three benchmarks, less than* 0.8.

1. Introduction

One solution to the growing disparity between CPU and memory speed is to *prefetch*, move lines into a cache that might soon be accessed. In hardware prefetch schemes added hardware determines which lines to prefetch and then moves them to the cache (or to a special prefetch buffer). If a load or store instruction subsequently accesses such lines, and they are still present, a cache miss is avoided. By hiding substantial miss delays hardware prefetching has vast potential to improve the performance of systems running code that has not been specially prepared. This is especially true for multiprocessors where miss delays can be hundreds of cycles or more.

Existing schemes do a good job of prefetching lines that will be accessed soon. In many schemes the selected line is the one that would be accessed if an observed sequential or stride memory reference pattern would continue [5,6,25]. However such schemes have no mechanism for detecting the end of a sequence, so some prefetched lines are accessed much later, if at all. The problem is particularly bad when the address reference stream consists of mostly short sequences of sequential or stride accesses.

[*]A shorter version of this report was presented at PACT 2000 [17].

The prefetching of these unused (or not soon used) lines can slow other activity by consuming resources such as busses and cache ports. The unused lines may replace other lines that will be accessed, and, in multiprocessors, may contribute to false sharing.

A prefetch scheme is described here in which the lines prefetched more closely match the lines that are actually used. On a miss prefetches are issued for a set of lines surrounding the line that was missed, the *neighborhood*. The choice of lines to prefetch is based on the identity of the instruction that missed and the effective address it used.

This *neighborhood prefetch* provides the benefits of sequential prefetching, speculative exclusive access, and to a limited extent, stride prefetching. Execution-driven simulations were performed of multiprocessors using neighborhood prefetch, a sequential prefetch scheme, and no prefetching, running programs from the SPLASH-2 benchmark suite [28]. The simulations show that this is not merely a sequential prefetch mechanism with an on-off switch but that prefetches are issued for non-contiguous lines. The history information that specifies prefetch candidates also detects contended areas of memory, avoiding harmful prefetches.

The remainder of this paper is organized as follows. Neighborhood and other prefetch schemes are described in Section 2. Prefetch implementations are described in detail in Section 3. Simulator details appear in Section 4 and Section 5. Simulation results appear in Section 6. Related work is described in Section 7 and conclusions follow in Section 8.

2. Prefetching

2.1. Prefetch Mechanisms

Hardware and software prefetch mechanisms speculatively bring data to a processor that is likely to be accessed soon. Hardware prefetch is typically initiated after a *demand fetch* (the execution of an ordinary CPU load or store instruction); in response *prefetch requests* are issued if suitable *prefetch addresses* are found.

The data returned by prefetch requests may be placed in a special *prefetch buffer* or in an existing cache. The systems simulated here use the existing cache, this simplifies the hardware and makes more efficient use of memory.

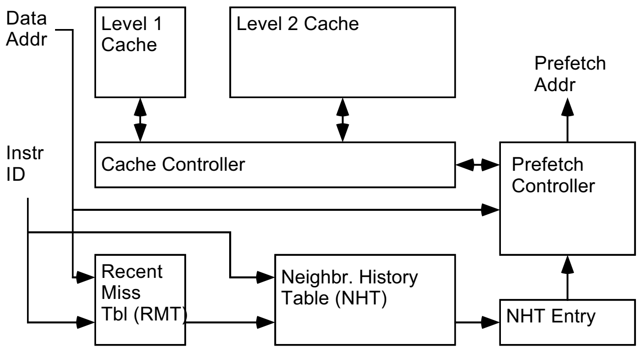

Figure 1. Cache and neighborhood prefetch hardware. The RMT collects the offset of addresses from a nearby initiator address that recently missed the level-two cache. The NHT, indexed by instruction id, holds the offset data collected by the RMT. When an NHT entry for a missing instruction is found the offsets are used to construct prefetch addresses.

Prefetch mechanisms base their decisions on memory reference streams, they differ in which stream they observe: level-one, level-two, etc. The most aggressive observe all accesses to the level-one cache, the added hardware can possibly add to hit latency. Prefetch mechanisms that observe the level-two stream (misses to the level-one cache) may leave the level-one cache unchanged, and so do not add to level-one hit latency. The systems simulated here (with a minor exception) observe the level-two miss stream, minimizing the impact on hit latency and relaxing speed constraints on the prefetch hardware.

2.2. Address Determination

Two common prefetch methods are *sequential* and *stride*. Sequential prefetch is designed to exploit reference (memory address) sequences of the form $a \cdots a+1 \cdots a+2 \cdots$. The prefetch mechanism may react to the use of an address a by fetching just $a+1$ or, for systems where the fetch latency is high, by fetching $a+1, a+2, \ldots, a+d$, where $d > 0$ is the prefetch *degree*. Such sequences are of course quite common; generated, for example, by sequential access to an array.

In *adaptive sequential prefetch* [6] (ASP) the degree is adjusted based on the success of past prefetches. The hardware keeps track of the number of prefetched lines that have been used. After some number of prefetches (15 was used in [6]) the number of used prefetches is checked and the degree adjusted.

Stride prefetch is designed to exploit reference sequences of the form $a \cdots a+s, \cdots a+2s \cdots$, where s is the *stride* of the prefetch. Such sequences might be generated by accessing an array at some stride (every s'th element) or by sequential access to an array of large elements, with only a part of each element being accessed.

Let a denote a memory address scaled to a *line*, the unit of cache storage, so that a and $a+1$ would be cached on different lines and there are no other lines in $[a, a+1]$. Let $a \cdots b \cdots c \cdots$ denote a memory reference sequence generated at a processor that includes addresses a, b, and c and possibly additional addresses interspersed.

Sequential prefetch is designed to exploit reference sequences of the form $a \cdots a+1 \cdots a+2 \cdots$. The prefetch mechanism may react to the use of an address a by fetching just $a+1$ or, for systems where the fetch latency is high, by fetching $a+1, a+2, \ldots, a+d$, where $d > 0$ is the prefetch *degree*. Such sequences are of course quite common; generated, for example, by sequential access to an array.

In a multiprocessor where fetch latencies can be very high and where there is the potential for false sharing a high degree can be both very helpful and harmful. In *adaptive sequential prefetch* [6] (ASP) the degree is adjusted based on the success of past prefetches. The hardware keeps track of the number of prefetched lines that have been used. After some number of prefetches (15 was used in [6]) the number of used prefetches is checked and the degree adjusted. A single degree is used for each processor. ASP initiates prefetch on a level-two miss, and so the degree (and prefetching) only applies to new, large, or moving working sets which generate level-two misses.

Stride prefetch is designed to exploit reference sequences of the form $a \cdots a+s, \cdots a+2s \cdots$, where s is the *stride* of the prefetch. Such sequences might be generated by accessing an array at some stride (every s'th element) or by sequential access to an array of large elements.

Stride prefetch hardware must determine what stride to use. Two methods have been described: associating a stride with the effective address of a demand fetch [9] and associating a stride with the address of the load or store instruction making a demand fetch [5,23]. In the former technique addresses which are nearby in time and space are used to form possible strides and to predict future addresses. If the predicted address appears the stride is used, associated with a next predicted address. In the latter scheme effective addresses are stored, indexed by the address (program counter value) of the instruction that issued them. On subsequent appearances of the instructions tentative strides are formed or a previous tentative stride is confirmed.

Neighborhood prefetch (NP) exploits sequences of the form $a \cdots a+o_1 \cdots a+o_2 \cdots \cdots \cdots a+o_d$, where $o_1, o_2, \ldots o_d$ are small-magnitude integers individually called *offsets* and collectively called a *neighborhood*. Such sequences might be generated when accessing a few parts of a large structure. Consider:

```
for( i=0; i<1000; i++ )
{ a = str[i].member_a;  s = str[i].member_s;
  z = str[i].member_z; }
```

Three members of a structure are read; a cache miss on an access to the first, `member_a`, might trigger a prefetch to the other two members. Using the notation above, if a is the address of the first member read, `&str[i].member_a`, then $o_1 =$ `&str[i].member_s - &str[i].member_a` and $o_2 =$`&str[i].member_z -`

Table 1. Neighborhood History Table Fields

Name, Bits	Description
Instr. tag, 2	Part of the instruction address; used to check for an NHT lookup miss.
Match count, 3	Number of times tag matched. (Initialized to 1, incremented on hit, decremented on miss, saturates at zero and 4.) An entry is not replaced when field is positive.
Neighb., 160	Field divided into subfields, called *offset counts*, for each possible offset (based on neighborhood size) for both loads and stores. The count is related to the number of times that an effective address was encountered at the corresponding offset plus adjustments for hits to, evictions of, and invalidations of prefetched lines. In the base configuration, the subfield size is five bits and the neighborhood size is 16 offsets, including zero.

`&str[i].member_a`. If the maximum offset were large enough then o_3 =`&str[i+1].member_a - &str[i].member_a`, and so data for the next iteration could be prefetched.

Like sequential and stride prefetching, neighborhood prefetching can exploit sequential and stride access patterns. Let $\{o_1, o_2, \ldots, o_d\}$ specify the offsets in a neighborhood. Neighborhood $\{1, 2, \ldots\}$ describes sequential access and $\{s, 2s, \ldots\}$ describes stride access.

In neighborhood prefetching a *neighborhood* is found for load and store instructions that miss the cache, identified by their program counter value. Details appear in the next section. Ignoring finite neighborhood size, the patterns exploitable by neighborhood prefetch are a superset of stride prefetch, which are in turn a superset of sequential prefetch. (Because prefetch mechanisms do not precisely detect these access patterns, the relative effectiveness might be different than this ordering.)

An advantage of sequential and adaptive sequential prefetch is that they can issue prefetches for effective addresses and instructions which have not been seen before. In contrast, stride and neighborhood prefetch must first detect and store the access pattern.

Dahlgren and Stenström compared stride prefetch and sequential prefetch on multiprocessors under conditions similar to the ones considered here [7]. They find that sequential prefetch is at least as good as stride prefetch, indicating that the benefit of detecting stride patterns is outweighed by learning time and inability to prefetch small sequential sequences. Learning time can potentially hobble neighborhood prefetch too, the results of this study indicate that its effectiveness overcomes missed prefetch opportunities due to learning time.

3. NP Implementation

A multiprocessor implementation of neighborhood prefetching will be described. (A multiprocessor is a cached shared-memory parallel machine. For background see [4,20,26].) Neighborhood prefetching can also be applied to serial systems, though the impact on execution time would be smaller due to lower miss latency. In addition to a greater potential for performance improvement, a multiprocessor implementation is more interesting since the prefetch mechanism must minimize accesses that cause or aggravate contention for shared data and because data can be prefetched in either the shared or exclusive states.

The description that follows is of the base system used to obtain the results described in Section 6. The base machine is a multiprocessor in which each CPU has two levels of cache. Each processor is paired with a memory module, which implements part of the shared address space. The mechanism handling protocol messages and controlling the cache and network interface will be called the *cache controller*. (For brevity a single name is used for what may be implemented as several controllers.) A full map directory coherence protocol is used [4].

Prefetching is initiated by a level-two cache miss. The cache controller will dispatch a request to the appropriate memory module for this demand fetch (as in a conventional system) and follow it with prefetch requests. Depending on the prefetch scheme used, addresses for these may be based on the demand fetch effective address, the *demand address* for short, the past behavior of the instruction missing the cache, called the *triggering instruction* here, and the presence of some other cache line. To avoid wasting resources by prefetching a line already present, the prefetch mechanism checks whether a line to be prefetched is already present using a tag store port shared with the processor and cache controller. Prefetched data is placed in the level-two cache.

The neighborhood prefetch hardware consists of a *neighborhood history table* (NHT) and a small *recent miss table* (RMT); see Figure 1. The NHT, direct-mapped and indexed by an *instruction id*, stores neighborhoods and other data needed for prefetching. The RMT, normally accessed using a demand address, is used for finding neighborhoods.

On a level-two cache miss the prefetch hardware determines a set of candidate prefetch addresses at the same time as a request is prepared for the demand fetch. First, the instruction id of the triggering instruction is used to retrieve an entry from the NHT. (The low-order bits of the instruction address are used as the instruction id.)

The NHT entry holds the neighborhood used for constructing prefetch candidates and an instruction tag and tag match fields used to detect NHT misses and determine replacement. See Table 1 for a complete list of fields, descriptions, and typical sizes.

The instruction tag consists of some part (subset of bits

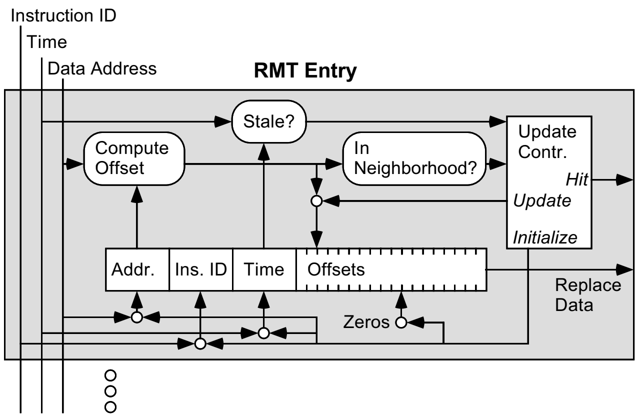

Figure 2. Recent miss table updating. When a level-two miss occurs, the data address is compared to each entry in the RMT (eight entries were used in the simulations). If a non-stale entry is found the offset from the entry's base address is written. If an entry is not found one is evicted and a new one is initialized.

in this case) of the instruction address which is not used in the instruction id. It is compared to the corresponding part of the trigger instruction address; if they differ the entry is for a different instruction, an NHT miss. To save space the tag is made small (2 bits in the base system) and so a match only indicates a possible hit. A *false hit* is a match to an entry which was created for a different instruction. Such a false hit might cause the "wrong" lines to be wastefully prefetched into the cache, but would not cause incorrect operation. On an NHT miss prefetching is aborted.

Most of the entry's space is devoted to the neighborhood field, consisting of two arrays of offset count subfields; one array for loads and one for stores.

The position of a subfield within an array indicates the offset and its value indicates the number of times (not by an exact count) there was a miss at that offset. A single bit per offset count subfield would be sufficient to specify a neighborhood, a multi-bit count is used to identify rarely-used offsets. Details are described further below.

Prefetch candidates are found by adding the demand address to those offsets with counts exceeding a threshold value. Two thresholds were used, a low one for the load offsets (10% of the maximum count value in the base system) and a higher one for the store offsets (50%). Addresses constructed using offsets from the load array are used to prefetch lines in the shared state; addresses constructed from the store array are used for exclusive prefetch or *preupgrade* (speculatively upgrading a line). Note that even when stores are nonblocking and loads proceed out of order with respect to stores, the number of in-flight stores is limited, so prefetching lines in the exclusive state does help performance. A higher store threshold was chosen so that exclusive prefetches would be more conservative since the penalty for prefetching an exclusive copy of a line that will be read elsewhere is higher than prefetching a line that will not be used.

To avoid issuing memory requests for lines that are already present and in the proper state the cache is checked for the presence of prefetch candidates. If present a prefetch request is not dispatched. The time taken for this presence checking can potentially erode performance, but this has not been a problem in the simulated systems. If there is a memory request in progress for the set corresponding to the prefetch candidate or if there is no TLB entry the prefetch is aborted.

The order in which prefetch requests are issued is based on the order of the offsets: first offset 1, then 2, up to the highest offset, followed by -1, -2, down to the lowest. (Other orderings tried, such as 1,-1,2,-2,..., were not as effective.) The level-two cache is checked for the presence of a prefetch candidate, if not present a prefetch request is sent. The level-two cache latency determines the maximum prefetch request rate. Demand-fetch accesses take priority, slowing prefetch requests further.

Prefetch requests can congest memories. To limit congestion prefetch requests are not issued to congested memory modules. Cache controllers learn of congestion by congestion-on and -off protocol messages returned by memories in response to other messages.

Prefetch requests are handled by memory in the usual way, however prefetch and demand fetch requests are (usually) queued separately. An arriving prefetch request is placed in the prefetch queue, an arriving demand fetch is placed in the prefetch queue if its address matches an entry in the prefetch queue, otherwise it is placed in the demand fetch queue. The arrival time of the requests at the head of the queues are compared with a time penalty added to the prefetch queue, the earlier one is dequeued. A 500-cycle penalty was used in the base system. Once dequeued the memory controller handles demand and prefetch requests identically.

The cache controller stores arriving data for demand and prefetches in the same way, except that a prefetch bit will be set for a prefetched line. The prefetch bit is reset when the line is accessed. (The prefetch bit is the only additional per-line storage needed, and is needed by both NP and ASP.) In the coherence protocol used the memory controller will not process some requests for lines in certain transitional states, returning a *busy* message instead. While a request for a demand fetch returning a busy would be retried, a prefetch request would not (to avoid adding contention).

Two mechanisms are used to avoid causing or aggravating contention by prefetching: a simple mechanism is based on the zero offset count, a more elaborate method tracks the fate of prefetched lines. The simple mechanism detects multiple misses to the same line, evidence of contention. The more elaborate method detects eviction and invalidation of unused prefetched lines, if found offset values in the RMT and NHT are adjusted.

The line at an offset of zero is the demand fetched line.

Table 2. Recent Miss Table Fields

Name, Bits	Description
Initiator, 10	Identity (hash of PC or index into instruction cache) of instruction creating this entry.
Base address, 26	Effective address accessed by initiator. Size based on 64-byte lines. Offsets are relative to this address.
Init. time, 20	Time that entry created.
Weight, 4	Sum of bin values up to 4-bit saturation.
Neighb., 128	Field divided into offset count subfields for each possible offset (based on neighborhood size) for both loads and stores. An offset count is the number of times that an effective address was encountered at the corresponding offset. In the base configuration, subfield size is four bits and the neighborhood size is 16 offsets, including zero.

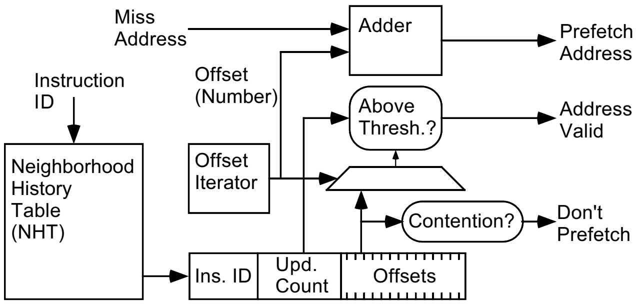

Figure 3. Prefetching using an NHT entry. When a level-two miss occurs the instruction id is used to look up an NHT entry. If the load and store zero offsets are both nonzero (indicating contention) no prefetching is done. Otherwise, each offset is compared to a threshold; if higher a prefetch address is constructed by adding the shifted offset to the current miss address.

If the zero offset count has a non-zero value then (given the way offsets are collected) a cache miss to a line was followed soon by a second miss to the line. This might occur if the line were brought to the cache, accessed, evicted, and accessed again. It might also occur if the line were invalidated or returned to the shared state soon after being cached, evidence of contention. Based on the assumption that there may also be contention for nearby lines, prefetching is aborted if the zero offset count in the read and write arrays is non-zero.

The more elaborate mechanism adjusts the offset counts. When a prefetched line is evicted before being accessed the RMT is searched and if a matching entry is found the the NHT entry is retrieved. If the corresponding bin is zero nothing is done, otherwise the RMT bins are zeroed and the NHT bins are decremented by 2. The procedure is similar but harsher if a prefetched line is invalidated. Because an instruction id is not stored for each line, occasionally the wrong RMT and NHT entries are retrieved.

See Figure 2 for the following discussion. Data for constructing neighborhoods is collected in the RMT, a small, fully associative table using a memory address as a key. Each entry has a field for the instruction id of the *initiator* (the instruction creating the entry), the base address (the effective address generated by the initiator), the time the entry was created, a *weight* field, and a field for each load and store offset count. Fields along with descriptions and typical sizes are listed in Table 2.

When a miss occurs the RMT is searched for an entry holding a neighborhood that can contain the demand address (by checking if the base address falls in a certain range), an eviction candidate, and an empty entry (if any). The eviction candidate is evicted if no empty RMT entry was found. The eviction candidate choice is based on its weight (sum of offset values), age, and whether it was initiated by the accessing instruction. An evicted entry is combined with an existing entry in the NHT or is used to initialize a new NHT entry.

The instruction id in the RMT is used to look up an NHT entry. If the NHT tag indicates a hit the offset counts are updated by adding the RMT offset counts to those found in the NHT entry and, if any sum would overflow the offset count field, dividing all the sums by two. In the base system offset count sizes are 5 bits, so the bins can easily be processed in parallel. If the NHT tag indicates a miss and the match count field is positive, it is decremented and nothing is further is done with the evicted RMT entry. If the match count is zero the NHT entry is initialized using the evicted RMT entry.

The weight and appropriate offset count is incremented in the entry retrieved from the RMT (for the demand address). If an RMT entry for the demand address is not found an empty or evicted entry is initialized.

3.1. Cost

Neighborhood prefetching does add complexity and cost. Here cost and performance effects will be estimated and those estimates will be used in Section 6 to pair systems using neighborhood prefetching with lower-cost ASP and conventional systems.

The cost measure used will be the amount of additional storage needed for neighborhood prefetching. Though a fair amount of logic is also used, estimating its cost is more difficult. The performance comparisons will be conservative in that the storage used by the neighborhood prefetch systems (cache plus tables) will be lower, however no attempt will be made to argue that the margin covers other implementation costs.

In the base system the RMT has 8 entries, each entry is 188 bits for a total of 188 bytes. (See Table 2 for base values.) The NHT in the base system has 256 entries, each taking 165 bits for a total size of 4448 bytes. (See

Table 1 for base values.) The base system uses a 65,536-byte 8-way level-two cache, so that storage is only 8.3% of the level-two cache size. In the experiments described below NP uses 7-way caches while ASP and conventional systems use 8-way caches. The storage needed for a 7-way cache (57,344 bytes) plus the storage needed for NP is less than the storage used by 8-way caches, 65,536 bytes.

3.2. Scaling

The SPLASH-2 programs are based on real applications but with small input sizes, so that simulation is feasible [28]. The base system parameters were chosen to match the SPLASH-2 benchmark's small input sizes. An important question is how well would the cost arguments apply to "real-world" input sizes and systems. Caches on current computers can be 2^{19} bytes or more, about eight times the size of the base system. The neighborhood size, RMT, and NHT would probably not scale proportionately.

Neighborhood size is determined by spatial locality (so that enough prefetch candidates can be found in a neighborhood). Though input size can have a profound effect on how a program executes (for example, changing the proportion of time spent in one phase of execution) its effect on spatial locality is likely small since spatial locality occurs with sequential access and access to different elements of a collection of data (*e.g.*, members of a structure). Therefore the base neighborhood size would probably work well for larger systems.

A similar argument could be made for the RMT size. The size of the RMT is determined by the number of different areas of address space at which misses are occurring. This might be determined by the number of arrays or other data structures being accessed over a short period, a property of the program rather than input data. Increasing input size would then not affect the RMT. The argument is supported by the simulation data, which show that the effect of RMT size variation appears insensitive to benchmark size.

The number of NHT entries needed is determined by the program. Assuming runs using the SPLASH-2 input sizes execute realistic proportions of the code, changing the input size would not affect the size of the NHT needed. A larger NHT would be needed if shared memory instruction usage in the SPLASH-2 programs did not represent real programs. That is, real programs might have and use more code, and so more memory access instructions, testing data for special conditions or handling multiple computation variations. If that is the case a larger NHT would be needed.

If the storage used for neighborhood prefetching in the base system is sufficient for larger systems, the amount of prefetching storage needed would be less than 1% of a half megabyte cache.

3.3. Timing

Neighborhood prefetching hardware should determine prefetch candidates and perform RMT updates without slowing other activities and without requiring especially fast hardware. Presence testing should have only a minimum effect on performance.

On a level-one miss, the RMT, NHT, and level-two cache lookup can proceed in parallel. If the result is a cache miss, prefetch candidates can be computed in parallel with demand fetch request construction-and-dispatch and with RMT and possible NHT updating. The hardware needed for address computation is little more than a simple adder. Little hardware is needed for threshold testing of small offset count values (5 bits), and could easily be done in parallel for a whole neighborhood. Similarly the adding of offset counts needed for NHT update could be done in parallel.

The NHT and RMT are also updated on invalidations and evictions. Both of these are fairly infrequent and should have little impact on performance.

Presence testing will occupy the level-two cache, slowing down accesses. This caused little performance degradation on the base system since at most one outstanding load miss was allowed and stores are nonblocking. The prefetching presence testing performed during a load miss would be completed before the load miss was satisfied, so no other loads would be slowed. Stores are nonblocking so delaying their completion would have little effect. Therefore presence testing would not cause a problem on systems that block on load misses and have nonblocking stores.

The caching of prefetched lines can slow down access to the level-two cache unless a dedicated cache port is added. In the simulated system a single cache port is used, the effect of the added contention was small.

4. Evaluation

4.1. Proteus

The simulations were performed using a modified version of the Proteus simulator [3]. Modifications were made to simulate the prefetch scheme described here, other unrelated modifications were made; that is, they impact the reported performance of systems that do not use the speculative hardware. For details on the changes see [16].

Proteus is an execution-driven parallel computer simulator which simulates a network, memory system, and processors running parallel programs. The modified version of Proteus runs on Sparc systems, and was run on Solaris 2.5.1, and so the simulated system implements the SPARC V9 ISA (though only a 32-bit subset is used).

The simulated system runs parallel programs written in C and which include some parallel programming functions and operations, such as shared memory operations. The C programs are pre-processed and compiled using a host-system compiler, in this case gcc 2.7.2.1. The compiled code (in assembly form) is augmented at at least every basic block with cycle-counting and shared-memory

Table 3. Base Con£guration Parameters

Simulation Parameter	Value
System Size	16 processors
Network Topology	4×4 mesh
VM Page Size	2^{12} bytes
TLB Capacity	64 entries
TLB Replacement	LRU, fully assoc.
Cache Size	2^7 sets
Cache Associativity	8, LRU Repl.
Cache Line Size	64 bytes
L1 Cache Hit Latency	1 cycle
L2 Cache Hit Latency	7 cycles
Total L2 Miss Latency	50 (min), 135 (typ)
Directory Size	full map
Completion Buffer	5 stores
Raw Memory Latency	10 cycles
Protocol Message Size	8 bytes (plus data)
Network Interface Width	4 bytes
Network Link Width	4 bytes
Hop Latency	20 cycles (plus waiting)
Neighborhood Size	16 offsets
NHT Size	256 entries
RMT Size	8 entries

access code. The simulated system runs user (the benchmarks), library, and some OS code. The OS code includes a TLB miss handler and other virtual memory management procedures, so VM management timing is accurate.

The augmentation process inserts code for cycle counting, simulator context switches, and shared memory accesses. The cycle counting code keeps track of time on the simulated system and initiates context switches. Time is advanced at the rate of about one cycle for every four instructions except for load, store, and certain floating-point instructions. Load and store instructions that might access shared memory are replaced with code that tests the address and calls simulator procedures if shared memory is indeed accessed.

Two-level, virtually mapped caches are simulated. The level-one cache is direct mapped, the level-two cache is set-associative with the same number of sets as the level-one cache. LRU replacement is used. The level-one cache hit latency is one cycle, the level-two hit latency is seven cycles (total). Miss latency is determined by network interface, network, memory, cache latencies, and the protocol actions needed to complete the accesses. Minimum miss latency is 50 cycles (local, no contention), typical values are 135 cycles. Further information can be found in [16].

The interconnection network is simulated at the packet-transfer level; 2-dimensional meshes are used for the work reported here. Network nodes effectively consist of a single shared infinite buffer. Waiting time within the network is insignificant but there is some waiting at network output buffers.

The simulated system provides virtual memory, using 2^{12}-byte pages and 64-entry, fully associative TLBs. Memory allocation routines can return a single contiguous block distributed over all memory modules. Stores are nonblocking but complete in program order with respect to other stores; up to five stores per processor can be simultaneously active. Loads do not complete in order with respect to stores, but of course can read values to be written by pending stores, maintaining thread-specified data dependence. The simulated memory system uses a full-map directory similar to the one described in [4], for differences see [16]. Each processor has an associated memory module, sharing the network interface queue with messages bound for the processor.

4.2. SPLASH-2 Suite

The SPLASH-2 suite consists of a representative sample of scientific shared-memory parallel programs for use in testing shared memory systems [28]. Three SPLASH-2 kernel programs ran for the results reported here are Cholesky, FFT, and LU. The fourth kernel, Radix, was used in modified form. (The modifications improved the efficiency of the prefix sum used in the kernel.) Radix is an integer sorting program, Cholesky factors sparse matrices, FFT performs a 1-dimensional fast-Fourier transform using a "radix-$\sqrt{N}$, six-step" algorithm, and LU is a dense-matrix LU factorization program. Four SPLASH-2 applications were also run, Barnes, FMM, Ocean (contiguous partitions), and Water N^2. Barnes simulates particle interactions in three dimensions using the Barnes-Hut method and FMM simulates particle interactions in two dimensions using the Adaptive Fast Multipole Method; both use tree-based data structures, though of different types. Water N^2 simulates forces on water molecules and Ocean simulates ocean currents [28]. The programs were run using the base problem sizes specified in the distribution. The programs' comments specify where statistics gathering might start and stop; the statistics described below are collected in those intervals. The base problem sizes are used except for LU, 128×128 matrix and Cholesky which was run with input file tk14.O.

4.3. Configurations

The experiments tested several different system configurations. The table gives parameters for the *base system* configuration; experiments will be described as modifications to this system.

5. ASP Implementation

The adaptive sequential prefetch of Dahlgren, Dubois, and Stenström as described in [6] was simulated, with the following changes to tune performance for the configurations used. The maximum prefetch degree is limited to four (with 64-byte lines). The lookahead count (degree) is updated after every 40 prefetches and is reset if prefetched lines are invalidated. Lines are prefetched in the exclusive state or preupgraded if the demand fetch is a store. Prefetches are aborted under the same conditions as in the NP implementation (presence, TLB miss, etc.) except

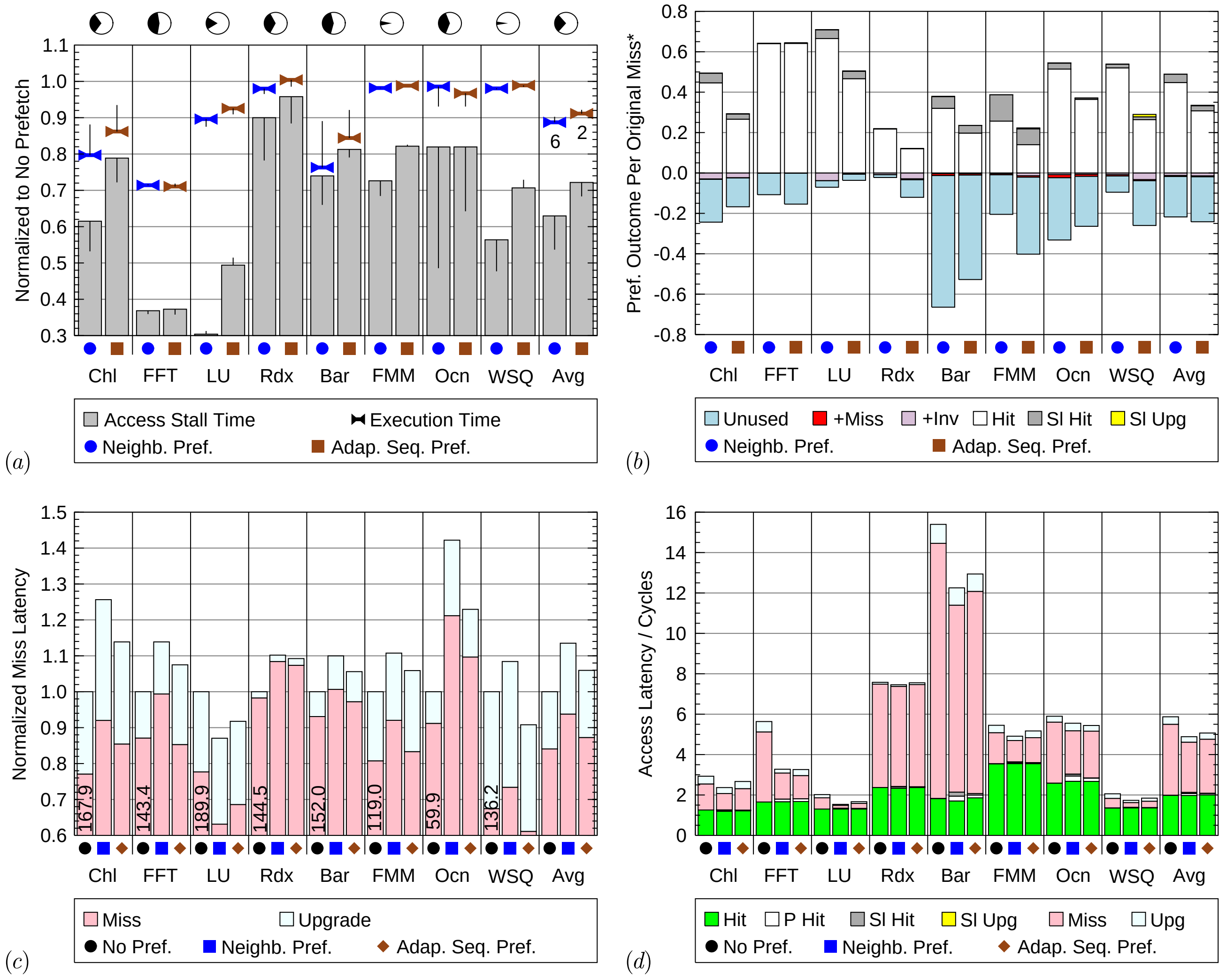

Figure 4. Performance of SPLASH-2 benchmarks on conventional, neighborhood prefetch, and adaptive sequential prefetch systems using base parameters. The fraction of computation time taken by memory access stall cycles shown in pie charts above (*a*); the numbers below the bow ties show the number of benchmarks for which respective system fastest. *Prefetch outcome per original miss that would occur on a conventional system plotted in (*b*) (see text for segment descriptions). Miss latency normalized to a conventional system plotted in (*c*). Access latency with weighted contributions plotted in (*d*).

where NHT data is used. Memory input congestion control is identical.

6. Experiments

6.1. Performance

The effectiveness of neighborhood prefetching and adaptive sequential prefetching can be seen in Figure 4(*a*), where normalized execution time (bow ties) and normalized *memory access stall cycles* (bars) are plotted for each benchmark running on the base system. The additional hardware needed for NP is compensated for by using a smaller, lower-associativity (7-way), cache, as described in Section 3.1. The caches in conventional and ASP systems are the same size.

On average the normalized execution time using NP is 0.89, with two benchmarks below 0.80. Average normalized execution time using ASP is only slightly worse at 0.91 however NP is better for 6 out of 8 benchmarks. For half the benchmarks prefetching had almost no impact on execution time. The reasons vary, though in all cases prefetching removed a substantial number of misses.

Prefetch performance depends upon how many misses are avoided by prefetch, how few misses are added by prefetch, how little prefetching slows normal accesses, and how sensitive the system is to the resulting changes in access latency. These factors and the way they impact execution are shown in Figure 4(*a*) and (*b*).

Potential performance improvement originates with avoided misses, shown by the white segments (above the

Table 4. Prefetching Behavior of Selected Instructions

Benchmark		ASP		NP		
Name	File:Line	Rate	Acc.	Rate	Acc.	Neighborhood
(1) Radix	radix.c:476	2.3	100%	6.7	100%	`-------!99999999`
(2) Radix	radix.c:530	0.2	98%	1.3	98%	`-------!91------`
(3) Radix	radix.c:531	0.2	8%	0.0	3%	`-------!9----1-1`
(4) FMM	interactions.c:168	0.8	49%	1.0	99%	`------9!--------`
(5) FMM	box.c:90	1.0	69%	2.5	88%	`-------!97----79`
(6) WSQ	interf.c:71	0.0	91%	6.8	100%	`8999999!1111111-`

axis) in Figure 4(b). The height gives the fraction of original misses that were avoided, so for example, the first bar indicates that NP on a system running Cholesky avoided 45% of the misses (that occur on a conventional system). The segments labeled SI Hit show accesses to lines that had been prefetched but have not yet arrived, their latency is between that of a hit and a miss. The segments labeled SI Upg show accesses to lines that are begin upgraded (converted to an exclusive state) by a prefetch. Overall NP avoids about half the misses while ASP avoids about 35%.

Only a small number of misses are added by prefetching. The segment closest to the axis, +Inv, shows unused prefetches that were invalidated (possibly increasing miss latency at another processor), the middle segment, +Miss, shows unused prefetches that evicted a line that was needed (probably adding a miss). There are few such accesses because both prefetch schemes can usually detect such misses and slow down or turn off. The bottommost—and usually largest—segment, Unused, shows harmless unused prefetches. On average, NP retrieves fewer unused lines and fewer lines that cause misses even while prefetching more lines. ASP prefetches fewer lines because it cannot detect where a sequence stops, and so must slow all prefetching when a few prefetches turn out to be harmful.

The sensitivity to miss ratio improvement and the actual and expected reduction in access latency are plotted in Figure 4(a). The pie charts and bars describe stall cycles, the time the processor is stalled due to a cache miss. (Stalls occur on all load misses and on write misses that fill the store buffer.)

The bars in Figure 4(a) show memory access stall cycles normalized to the conventional system. On average stall cycles are about 62% for NP but only 72% for ASP. Stalls are due to misses, so the reduction in misses also reduces stall cycles. The relationship would be proportional if miss latency were constant and prefetch affected all instructions equally. If this were so stall cycles would be reduced to 50% (not 62%) for NP and to 65% (not 72%) for ASP. Lines starting at the bar tops lead to these ideal values which are usually lower (better) than the one measured. The difference is due to the higher miss latency caused by prefetching. Miss latency is plotted in Figure 4(c), and access latency is plotted in (d).

The fraction of computation time in the conventional system taken up by stalls, *sensitivity*, is shown by the pie charts at the top of Figure 4(a). These show how much the 62% stall cycle time reduction can impact performance. (Computation includes the time the programs are active but excludes time in barriers.) Sensitivity is small for FMM and Water N^2, explaining why execution time is little changed.

Sensitivity and stall cycle reduction can be used to predict an execution time. The lines originating at the bow ties lead to such a predicted normalized execution time. For most of the benchmarks the prediction is close but Cholesky and Barnes run faster while Ocean is slower. In these benchmarks prefetch favors instructions either on (runs faster than expected) or off (runs slower than expected) the critical path. Cholesky and Barnes spend a substantial amount of time spin waiting (re-checking a location until a certain value is present). Spin waiting can add a substantial amount of misses which are not (and should not be) avoided by prefetching. Most of the reduction in stall cycles is for computation code which affects execution time. (While the semaphore code uses test and test and set semaphores, the barriers are based on messages and so generate fewer misses.) Ocean is slower than expected because code on a critical path has a higher-than-average number of misses (because of the frequent use of barriers), so the elimination of some misses from the critical path is outweighed by the increase in stall cycles from those that remain.

Normalized miss latencies are plotted in Figure 4(c). The miss latencies plotted do not include the initial or final cache access, but do include network and memory delays. This was done to show delays beyond the accessing cache. Ocean suffers the most and so is denied a large speedup. LU and Water N^2 show lower miss latencies, perhaps because the timing of prefetches avoids contention. In the systems simulated added miss latency is due to waiting at the memory modules.

6.2. Neighborhood Examples

NP was designed to discover access patterns on a per-instruction basis. Examples of such patterns appear in Table 4. The table shows prefetch rate (lines prefetched per miss, some values rounded to zero) and accuracy (percentage used) as well as neighborhoods for selected instructions. The neighborhoods are shown using one digit per offset, with an exclamation point at the zero position.

The digit indicates the relative number of misses observed.

In the first example Radix is sequentially accessing a long array without generating many other misses; the neighborhood clearly shows the sequential access pattern. Both NP and ASP do well, NP is closer to its maximum prefetch rate of 8 because the RMT discovers the pattern after 9 misses while ASP's prefetch degree is incremented (by one) only after a window-full of prefetches (40) have been issued. ASP would work better on (1) with a larger maximum degree, but 4 gave the best overall performance.

Instructions (2) and (3) are in Radix's permute loop. Instruction (2) sequentially reads from a single array while (3) writes to multiple locations, each sequentially. Both instruction accesses are sequential however writes by (3) are to smaller areas and so prefetches are more likely to overshoot and retrieve another processor's data. ASP adjusts by nearly stopping prefetch to both instructions (the rate rounded down to zero), while NP prefetches for (2) but not for (3). The neighborhoods are small because RMT entries are quickly replaced by execution of instruction (3).

Instruction (4), for benchmark FMM, accesses a structure element placed near the end of the structure; subsequent instructions access members that fall on a preceding line. Instruction (5) shows an example of stride access.

Instruction (6), for benchmark Water N^2, shows an example of backwards sequential access that is arguably pathological. The code initializes a multidimensional array using ordinary nested loops with increasing indices. Data storage for the array was allocated in extremely small (12-byte) chunks using a dynamic memory allocator based on Gnu malloc. Gnu malloc obtains small chunks from the end of a block of memory, so consecutive small allocation addresses are descending. (Larger allocations, which most programs use, are in ascending order.) NP has no trouble with backward access, while ASP stops prefetching.

6.3. Cache Characteristics

Changing the cache size changes the mix of miss types, strongly affecting prefetch performance. Ignoring variation in parallel program execution, the number of cold and sharing misses is constant with cache size, and dominate at large cache sizes. As cache size is reduced conflict and capacity misses are added. Two effects hinder performance at small cache sizes. First, if only a few lines of a neighborhood or sequence are missing the prefetch hardware will waste time on a miss to one of those lines checking for the presence of already cached prefetch candidates. (Though demand fetches are given priority, they cannot preempt an in-progress access to the level-two cache by the prefetch hardware.) Second, prefetched lines may be replaced before being used. These effects can be seen in Figure 5(*b*) where conventional and ASP cache sizes were varied from 2^4 sets (2^{13} bytes) to 2^{11} sets (2^{20} bytes), while NP cache sizes were $\frac{7}{8}$ these sizes. (The handicap was made a constant fraction of cache size for comparison.) At smaller cache sizes there are fewer useful prefetches per miss and more unused ones. Figure 5(*a*) shows that performance is best at middle sizes and that at large cache sizes sharing misses are still a substantial part of execution time (based on stall time).

Miss ratio is plotted on a logarithmic scale in Figure 5(*c*). Starting at 2^6 sets prefetching reduces the miss ratio by a constant percentage (constant vertical distance in the graph). Below 2^6 sets miss ratio is cut by a smaller amount. Figure 5(*b*) shows that at small sizes there are more unused prefetches, probably evicted before being used. At larger cache sizes there is a greater proportion of sharing misses, these misses have higher latencies since data must be fetched from another processor, apparent in Figure 5(*d*). Mistakes prefetching shared data are more costly since prefetched lines must be invalidated; such mistakes are an increasing proportion of prefetches as can be seen in Figure 5(*b*).

Since lines are (usually) larger than a word size they provide something of a passive prefetch mechanism, but one without any means to stop harmful prefetches. The base line size minimizes the average execution time, as can be seen in Figure 6(*a*) where the harmonic mean of execution times is plotted on a logarithmic scale. Line size was varied from 8 to 1024 bytes while holding cache size constant. To improve performance the maximum degree for ASP is set so that the maximum prefetch amount is 256 bytes for line sizes below 128 bytes, and one line for larger sizes. NP parameters were not tuned for line size (the neighborhood size is fixed at 16 lines).

For smaller line sizes the prefetch mechanisms do comparatively better, at large line sizes the performance of all systems is about equal. NP is better than ASP for all but the largest line sizes. The best performance, with or without prefetch, is with 64-byte lines, the base size. The prefetch systems perform nearly as well with 32-byte lines, an advantage since such systems are less susceptible to false sharing. That sequential access occurs on a small scale is evident in Figure 6(*b*) where miss ratio is plotted on a logarithmic scale. The nearly straight lines below 2^7 bytes indicate that each doubling of line size removes a constant fraction of misses, as would be expected with sequential access.

Performance is plotted in Figure 6(*c*). This again shows that prefetch provides larger relative performance at smaller line sizes. The fraction of stall cycles is smallest at middle line sizes, at smaller line sizes there are more cold and sharing misses, at larger line sizes there are more false sharing misses. As seen in Figure 6(*d*); the prefetch rate drops with increasing line size. Part of that is due to purely sequential access patterns captured by longer lines.

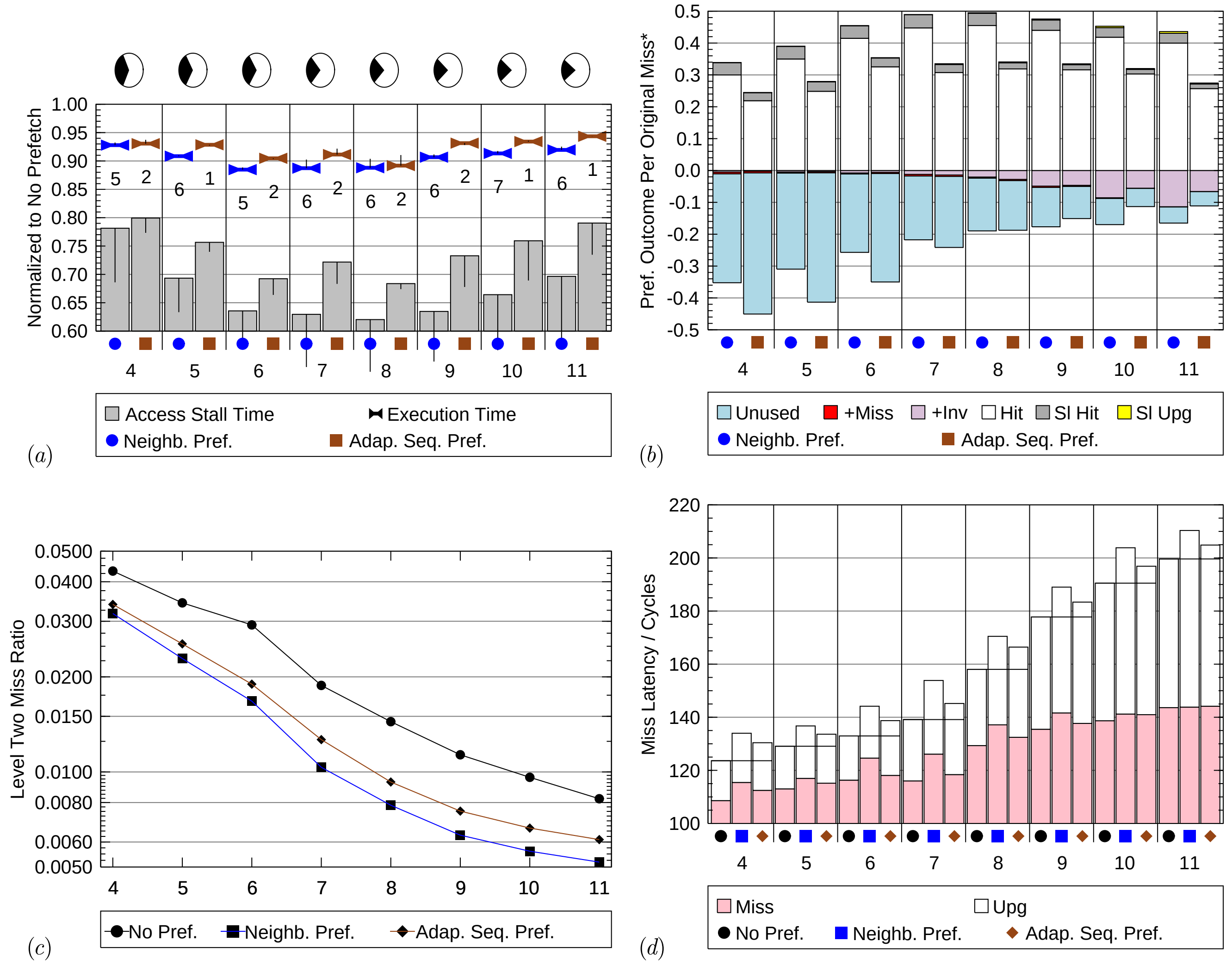

Figure 5. Performance v. cache size; x-axis labels give $\log_2$ number of sets. In (*a*), normalized execution time and stall cycles; (*b*), prefetch outcome; (*c*), miss ratio (including level-one accesses); and (*d*), miss latency with contribution of misses and upgrades. See caption on Figure 4 for more on (*a*) and (*b*). *Prefetch outcome per original miss that would occur on a conventional system.

7. Related Work

Both software and hardware prefetch mechanisms have been studied. Software prefetch is initiated by special software prefetch instructions (available, for example, in SPARC V9 and IA-64 [13,27]) inserted by the compiler or programmer. In contrast, hardware prefetching requires no code modification and suffers no code expansion. See [15,21] for software prefetching on serial systems and [10,19,22] for parallel systems.

7.1. Hardware Prefetching

Work on hardware sequential prefetching for serial systems dates back to the 70's, when the number of instructions that could execute in a miss delay was much smaller than it is today. Sequential prefetching schemes were investigated by Bennett et al [1,2], Gindele [11], and Smith [24] and are summarized and evaluated by Smith in [25]. These schemes vary in how prefetching is initiated, for example, on all accesses, all misses, or on all misses and first hits to prefetched lines. Always prefetching and tagged prefetch reduced half or more misses, while prefetch on miss was less effective.

Fu and Patel [8] have investigated a fixed sequential prefetch mechanism for parallel vector machines; the number of lines prefetched is fixed. They show improved performance over systems having larger line sizes, though on vector programs for which sequential prefetch is well suited. More recent sequential prefetch work has addressed the problem of having the hardware fetch far enough in advance so that lines arrive before they are needed, a growing problem because of increasing processor speed with respect to miss delay. As described earlier, the

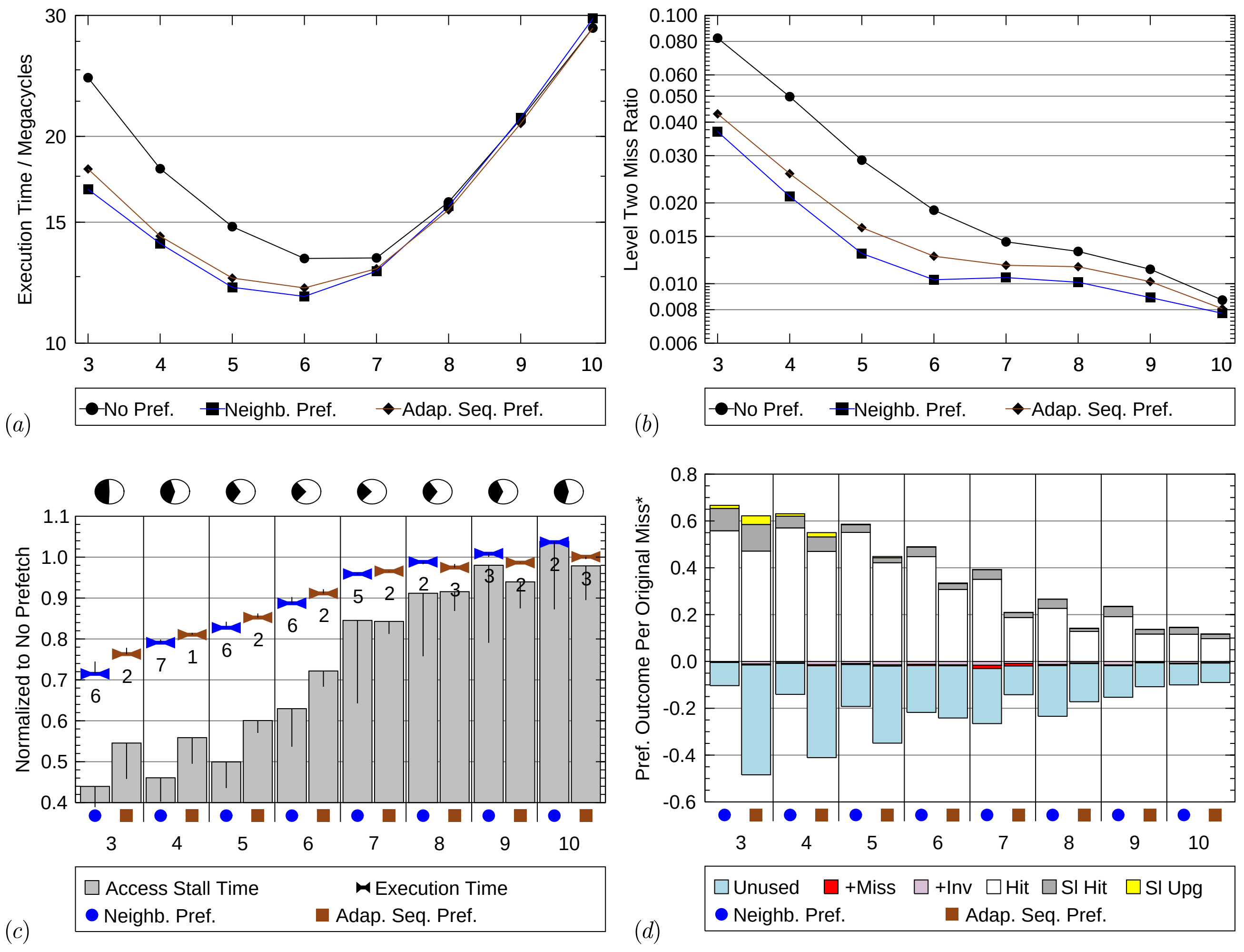

Figure 6. Performance v. line size; x-axis labels give $\log_2$ line size. In (*a*), geometric mean of execution; (*b*), miss ratio (including level-one accesses); (*c*), normalized execution time and stall cycles; and (*d*), prefetch outcome. See caption on Figure 4 for more on (*c*) and (*d*).

adaptive sequential prefetching of Dahlgren, Dubois, and Stenström [6] adjusts the number of prefetched lines based on the fate of recently prefetched lines. Unlike fixed sequential prefetch, their scheme can fetch far enough ahead that lines arrive when needed and can also reduce the degree of prefetching or stop it altogether when prefetches are not accessed [6]. They do not address the problem of prefetching lines in the exclusive state or preupgrading cached lines that might be accessed.

An extensive amount of work has been devoted to prefetching data at some stride. Fu and Patel [8] describe a stride prefetch for a parallel vector machine (the same one they used for sequential prefetching) which uses the stride information already present in vector instructions. In [9] Fu, Patel, and Janssens describe a stride detection mechanism for serial processors which finds patterns in access addresses. In an early work Sklenár proposes a stride prefetch unit for serial machines which records the stride history of accesses issued by each instruction [23], performing stride prefetch without having the stride specified. Sklenár's analysis did not include simulation; such schemes were evaluated via simulation by Chen and Baer [5], also for serial systems, and by Dahlgren and Stenström for multiprocessors [7]. Like neighborhood prefetch they record access history for individual instructions, including a guessed stride and two state bits to describe the status of that guess. Prefetches are only issued when the same stride has been encountered multiple times. The upcoming Sun Microsystems UltraSPARC-III processor incorporates an instruction-correlated hardware stride prefetcher as well as software prefetch [12].

Chen and Baer describe an instruction lookahead mechanism for use with their stride prefetch scheme [5]. A shadow program counter is designed to run ahead of the processor's program counter so load instructions can be identified for prefetch. Because useless prefetches do not lead to incorrect execution, the shadow program counter can boldly advance without concern for recovering from mispeculation, using a branch target buffer to find branches and other control transfers. One problem with such an approach is the need to either dual-port or shadow the branch target buffer. However much of the substantial reduction in miss latency of their stride prediction can be attained without program counter lookahead.

Joseph and Grunwald describe a prefetch mechanism designed to identify previously resident lines to a level-one cache, called the *Markov predictor* [14]. A history table has entries for recent misses, indexed by the address missing the cache. Each entry stores several addresses which had missed the cache following the key address, along with something similar to a probability in a Markov chain. These probabilities are used to prioritize prefetches when a miss to the key address is encountered. The Markov predictor was designed to prefetch items back into a level-one cache, unlike other prefetchers designed to fetch new data. Like neighborhood prefetching, the Markov predictor prefetches a number of addresses in response to a miss. However an entry in a Markov predictor will always prefetch the same lines, while neighborhood prefetching prefetches lines relative to a miss using the instruction address as a key.

In an independent work Kumar and Wilkerson [18] have applied neighborhood prediction to a sectored level-one cache. Rather than prefetch, the cache uses a large line divided into sectors. Normally in a sector cache sectors are loaded at demand fetch misses; using a *spatial footprint predictor* (SPF) a predicted set of sectors (analogous to a neighborhood) is loaded. The technique is less flexible than NP since base addresses must be aligned to entire neighborhoods rather than individual lines. Footprints are correlated with both instruction and data addresses.

SPF was evaluated for serial systems running integer programs. They show an 18% improvement in miss ratio which is not directly comparable with the 50% achieved here on prefetch-friendly scientific programs. They examined fetch bandwidth but did not evaluate execution time.

8. Conclusions

A technique of prefetching an area around a demand fetch based on instruction history has been presented. Prefetching based on a neighborhood covers more access patterns than sequential or stride prefetch. Both adaptive sequential and neighborhood prefetch provided usable speedup, but NP outperformed ASP for six out of eight benchmarks under most test conditions. The higher performance of NP is due to its higher prefetch rate, attainable because a larger fraction of prefetched lines are used. Prefetch-on-miss schemes were tested here. Additional performance improvement (for both ASP and NP) could probably be attained with schemes that also prefetch on hits to prefetched lines.

Prefetching was tested on multiprocessor systems in part because of their high miss latencies. With much smaller hit latencies encountered on present serial systems the speedups would be much smaller, probably not worth the trouble. If the gap between CPU and memory performance continues to widen prefetching might be useful on serial systems too.

9. Acknowledgment

This work was supported in part by the National Science Foundation under Grant No. MIP-9410435.

10. References

[1] B.T. Bennett and P.A. Franaczek, "Cache memory with prefetching of data by priority," *IBM Technical Disclosure Bulletin*, vol. 18, pp 4231-4232, May 1976.

[2] B.T. Bennett, J.H. Pomerene, T.R. Puzak, and R.N. Rechtschaffen, "Prefetching in a multilevel memory hierarchy," *IBM Technical Disclosure Bulletin*, vol. 25, p. 4103, June 1982.

[3] Eric A. Brewer, Chrysanthos N. Dellarocas, Adrian Colbrook, and William E. Weihl, "Proteus: a high-performance parallel-architecture simulator," in *Proc. of the ACM SIGMETRICS conference*, May 1992.

[4] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal, "Directory based cache coherence in large–scale multiprocessors," *IEEE Computer*, vol. 23, no. 6, pp. 49–59, June 1990.

[5] Tien-Fu Chen and Jean-Loup Baer, "Effective hardware-based data prefetching for high-performance processors," *IEEE Trans. on Computers*, vol. 44, no. 5, pp. 609-623, May 1995.

[6] Fredrik Dahlgren, Michel Dubois, and Per Stenström, "Sequential hardware prefetching in shared-memory multiprocessors," *IEEE Trans. on Parallel and Distributed Systems*, vol. 6, no. 7, pp. 733-746, July 1995.

[7] Fredrik Dahlgren and Per Stenström, "Evaluation of hardware-based stride and sequential prefetching in shared-memory multiprocessors," *IEEE Trans. on Parallel and Distributed Systems*, vol. 7, no. 4, pp. 385-398 April 1996.

[8] J. Fu and J.H. Patel, "Data prefetching in multiprocessor vector memories," in *Proc. of the 18th Annual International Symposium on Computer Architecture*, pp. 54-63, May 1991.

[9] J. Fu, J.H. Patel, and B.L. Janssens, "Stride directed prefetching in scalar processors," in *Proc. of the 25th Annual International Symposium on Microarchitecture*, pp. 102-110, 1992

[10] Kourosh Gharachorloo, Anoop Gupta, and John L. Hennessy, "Two techniques to enhance the performance of memory consistency models," in *Proc. of the Intl. Conference on Parallel Processing*, August 1991, vol. I, pp. 355–364.

[11] J.D. Gindele, "Buffer block prefetching method," *IBM Technical Disclosure Bulletin*, vol. 20, pp. 696-697, July 1977.

[12] Tim Horel and Gary Lauterbach, "UltraSPARC-III: designing third-generation 64-bit performance," *IEEE Micro Magazine*, vol. 19, no. 3, pp. 73-85, May 1999.

[13] Intel, "IA-64 application developer's guide," Intel, May 1999.

[14] Doug Joseph and Dirk Grunwald, "Prefetching using Markov predictors," in *Proceedings of the International Symposium on Computer Architecture*, June 1997, pp. 252–263.

[15] A.C. Klaiber and H.M. Levy, “An architecture for software-controlled data prefetching,” *Proc. of the International Symp. on Computer Arch.* May 1991, pp. 43–53.

[16] D.M. Koppelman, “Ver. L3.12 Proteus Changes” Department of Electrical and Computer Engineering, Louisiana State University, (simulator documentation),
http://www.ee.lsu.edu/koppel/proteus/proteusl_1.html
and http://www.ee.lsu.edu/koppel/proteus.

[17] D.M. Koppelman, “Neighborhood prefetching on multiprocessors using instruction history,” in the *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques,* October 2000, pp. 123-132.

[18] S. Kumar and C. Wilkerson, “Exploiting spatial locality in data caches using spatial footprints,” *Proceedings of the 25th Annual International Symposium on Computer Architecture,* June 1998, pp. 357–368.

[19] R.L. Lee, P.C. Yew, and D.H. Lawrie, “Data prefetching in shared memory multiprocessors”.*Proc. of the Intl. Conference on Parallel Processing.* August 1987, pp. 28–31.

[20] David J. Lilja, “Cache coherence in large-scale shared-memory multiprocessors: issues and comparisons,” *ACM Computing Surveys,* vol. 25, no. 3, pp. 303–338, September 1993.

[21] T.C. Mowry, M.S. Lam, and A. Gupta, “Design and evaluation of a compiler algorithm for prefetching,” *Proc. of the Conference on Architectural Support for Programming Languages and Operating Systems.* October 1992, pp. 62–73.

[22] Todd C. Mowry, “Tolerating latency in multiprocessors through compiler-inserted prefetching,” *ACM Transactions on Computer Systems*, vol. 16, no. 1, pp. 55-92, Feb. 1998.

[23] I. Sklenàr, “Prefetch unit for vector operation on scalar computers,” *Computer architecture news*, vol. 20, no. 4, pp.31-37, Sep. 1992.

[24] A.J. Smith, “Sequential program prefetching in memory hierarchies,” *IEEE Computer*, vol. 11, no. 12, pp.7-21, Dec. 1978

[25] A.J. Smith, “Cache Memories,” *ACM Computing Surveys*, vol. 14, no. 3, pp.473-530, Sep. 1982

[26] P. Stenström, “A survey of cache coherence schemes for multiprocessors,” *IEEE Computer*, vol. 23, no. 6, pp. 12–24, June 1990.

[27] David L. Weaver and Tom Germond (eds.), “The SPARC architecture manual, Version 9,” Englewood Cliffs, New Jersey: Prentice-Hall, 1994.

[28] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta, “The SPLASH-2 programs: characterization and methodological considerations,” *Proc. of the Intl. Symp. on Computer Arch.* May 1995, pp. 24–36.