A Performance Study of Software and Hardware Data Prefetching Schemes*

Tien-Fu Chen
Department of Computer Science
and Information Engineering
National Chung Cheng University
Chiayi, Taiwan, R.O.C.

Jean-Loup Baer
Department of Computer Science
and Engineering
University of Washington
Seattle, WA 98195

Abstract

Prefetching, i.e., exploiting the overlap of processor computations with data accesses, is one of several approaches for tolerating memory latencies. Prefetching can be either hardware-based or software-directed or a combination of both. Hardware-based prefetching, requiring some support unit connected to the cache, can dynamically handle prefetches at run-time without compiler intervention. Software-directed approaches rely on compiler technology to insert explicit prefetch instructions. Mowry et al.'s software scheme [13, 14] and our hardware approach [1] are two representative schemes.

In this paper, we evaluate approximations to these two schemes in the context of a shared-memory multiprocessor environment. Our qualitative comparisons indicate that both schemes are able to reduce cache misses in the domain of linear array references. When complex data access patterns are considered, the software approach has compile-time information to perform sophisticated prefetching whereas the hardware scheme has the advantage of manipulating dynamic information. The performance results from an instruction-level simulation of four benchmarks confirm these observations. Our simulations show that the hardware scheme introduces more memory traffic into the network and that the software scheme introduces a non-negligible instruction execution overhead. An approach combining software and hardware schemes is proposed; it shows promise in reducing the memory latency with least overhead.

1 Introduction

Prefetching has been shown to be one of several effective approaches that can be used to tolerate large memory latencies. Prefetching hides (part of) the memory latency by exploiting the overlap of processor computations with data accesses. Whether prefetching should be hardware-based or software-directed or a combination of both is an interesting question for the architecture community.

Hardware-based prefetching [1, 8] requires some support unit connected to the cache but little modification to the processor. Its main advantage is that prefetches are handled dynamically at run-time without compiler intervention. The drawbacks are that extra hardware resources are needed and that memory references for complex access patterns are difficult to predict. In contrast, software-directed approaches [4, 9, 11, 13, 14, 15] rely on compiler technology to perform static program analysis and to selectively insert prefetch instructions. The drawbacks are that there is some non-negligible execution overhead due to the extra prefetch instructions and that some useful prefetching cannot be uncovered at run-time.

Mowry and Gupta's software [13, 14] and Baer and Chen's hardware [1] approaches are two representative prefetching schemes. The hardware scheme that we use in this paper is a slight enhancement, described in the next section and in more detail in [3], to the one we proposed originally. The software scheme is our ''interpretation'' of Mowry et al.'s compiler algorithm and does not reflect advances in the algorithm posterior to its publication. We first compare the two schemes qualitatively, focusing on design aspects. A quantitative evaluation is then performed by a direct-execution simulation of three SPLASH benchmarks and of the Matmat kernel in a shared-memory multiprocessor environment. The metrics of interest include the effectiveness of prefetching, the increase in network traffic, and the performance sensitivity to a range of memory latencies. We also discuss means of combining both approaches.

In the domain of linear array references both hardware and software schemes are able to generate prefetches to reduce cache misses. When complex data access patterns are considered, the software approach may have more compile-time information to perform sophisticated prefetching whereas the hardware scheme has the advantage of manipulating dynamic information. The software scheme might suffer from a code expansion problem but the predictability that the prefetched data will be used is greater than in the hardware solution. Our performance results confirm these qualitative observations. Our results also show that hardware prefetching introduces more memory traffic into the network than software prefetching. Our simulations indicate that an approach combining software and hardware schemes is very promising in reducing the memory latency with least overhead.

The rest of the paper is organized as follows: the next section gives some background information on data prefetching. In Section 3, we compare the two schemes in a qualitative fashion. Section 4 describes the evaluation methodology. Section 5 presents simulation results and explores the impact of varying memory latencies, and side effects that prefetching can bring up. Section 6 proposes a way of combining the software and hardware schemes. Finally, we conclude in

*This work was supported in part by NSF Grant CCR-91-01541 and by Apple Computer, Inc.

Section 7.

2 Data Prefetching- The two selected schemes

Software-directed Prefetching approaches are implemented as an optimization phase of a compiler. Prefetch instructions, loading data in the cache in a non-binding fashion, are inserted several cycles before their corresponding memory instructions. Porterfield [15] showed that the intuitive idea of inserting prefetches for array references one iteration ahead in the most nested loops led to too much overhead. The time to prefetch should depend on memory latency and loop execution time [11, 4]. Gornish *et al.* [9] proposed a conservative algorithm to find the earliest point before a loop that an entire subarray could be prefetched.

In Mowry et al.'s approach [14], a compiler algorithm identifies those data references that are likely to be cache misses and prefetches are inserted only for them. Specifically, the focus is on array accesses whose indices are linear functions of the loop indices. The algorithm performs data reuse analysis and then derives, based on parameters such as cache and block sizes, a set of accesses that belong to an iteration space in which locality is preserved among accesses. Once the locality is known, a prefetch predicate for each reference that would lead to a cache miss is introduced in the loop for determining if the prefetch should be executed in a particular iteration. Loop splitting may be performed in order to reduce the computational cost of evaluating the predicates. Then prefetches are scheduled within the loop by taking into account the memory latency and estimated loop execution time.

To our knowledge, Mowry *et al.*'s approach is the best software prefetch algorithm currently available. We will use their framework as the basis of our comparison. We refer to our interpretation of their approach as the ''software scheme.''

In many **Hardware-based Prefetching** schemes, prefetches are generated on the basis of the access to the current cache block. Smith [17] studied variations on the one block lookahead (OBL) policy, i.e., upon referencing block *i*, block $i + 1$ is to be prefetched. An extension to OBL where several consecutive data blocks are prefetched in FIFO *stream buffers* has been proposed by Jouppi [10]. In OBL and extensions, miss rates can be reduced at the expense of some increase in memory traffic. These schemes take advantage of limited (sequential) spatial locality but do not deal with large strides. The use of stride information, e.g., carried by vector instructions, led Fu and Patel [7, 8] to propose prefetch strategies for vector and scalar processors.

We have proposed a more elaborate approach[1], called *lookahead data prefetching*, of which a slightly expanded version will be the ''hardware scheme'' for our evaluation. This scheme combines the advantages of stride information and instruction lookahead.

The essential hardware component is a support unit for a conventional data cache whose design is based on the prediction of the instruction execution stream and associated operand references in load instructions. The support unit is not on the critical path and therefore will not contribute to an increase in the processor cycle time. A reference prediction table (RPT) (cf. Figure 1), organized as a regular cache, records the referencing patterns. The RPT will be accessed ahead of the regular program counter (PC) by a lookahead program counter (LA-PC). The LA-PC is incremented and

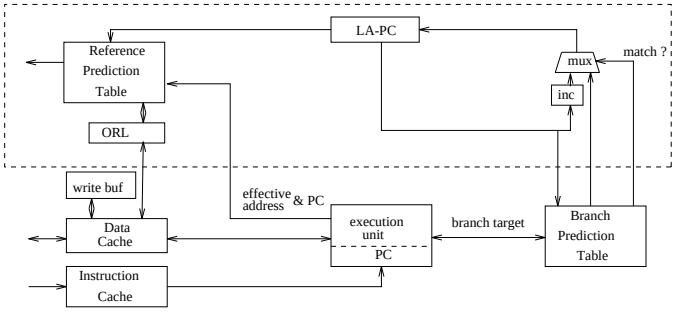

Figure 1: Structure of the hardware prefetching

maintained in the same fashion as the PC with the help of a dynamic branch prediction mechanism. Each RPT entry contains the reference prediction information for the corresponding access instruction. A *times* field is provided in each entry to indicate the number of iterations between the LA-PC and PC when a prefetch is generated [3]. The key to hiding memory latency is to keep enough distance, at least the memory latency time, between PC and LA-PC so that the prefetched data arrives just, or slightly before, it is needed. A system parameter, the LA-limit, set to a value slightly larger than the memory latency serves as an upper bound on the distance between PC and LA-PC.

3 Qualitative Evaluation

3.1 Identifying Cache Misses

The success of software prefetching depends primarily on identifying and inserting prefetch instructions only for those accesses that are most likely to generate cache misses. To that effect, the software scheme exploits three types of reuse: *temporal*, *spatial*, and *group*. Since reuses do not guarantee locality [19], these reuses are mapped to data locality by taking into account the loop iteration count and the cache size. To illustrate the concept of reuse, let us consider the loop in Figure 2 (a). The accesses to X[i] have spatial reuse since the same cache line is reused in consecutive iterations. Accesses to Y[i] and Y[i+1] share group reuse and the access Z has temporal reuse since it is referenced in different iterations.

While misses for memory accesses with spatial reuse are easily determined, the identification of cache misses for accesses with temporal and group reuse is rendered more complicated by other factors such as set associativity and replacement policy. Moreover, conflict misses due to self-interference from the same array references or cross-interference from different arrays are not predictable at all. Overall, the software scheme can be successful in identifying most compulsory misses and some of the capacity misses for linear array references, but is unable to handle conflict misses.

In contrast, the hardware scheme has no information that allows it to avoid unnecessary prefetches. However, there is no CPU-overhead associated with these extra prefetches as long as they are not on the critical path of the processor. Although prefetches are suppressed when the data block is already found in the cache, there remains the drawback that the additional lookup of the cache tag directory may still delay demand cache accesses or data refills from memory modules. Furthermore, since the prefetches have no knowledge of potential reuse, the hardware scheme is more likely to bring

(a) A loop example

```
for i = 0 to 255
      X[i] = Y[i+1] + Y[i+2] - Z
end
```

(b) Instrumented code

```
for i = 0 to 3 by 2                                  } prologue
      prefetch(&X[i])
      prefetch(&Y[i+1])
end
for i = 0 to 251 by 2                                } main loop
      prefetch(&X[i+4])
      prefetch(&Y[i+5])
      X[i] = Y[i+1] + Y[i+2] - Z
      X[i+1] = Y[i+2] + Y[i+3] - Z
end
for i = 252 to 255 by 2                              } epilogue
      X[i] = Y[i+1] + Y[i+2] - Z
      X[i+1] = Y[i+2] + Y[i+3] - Z
end
```

Figure 2: Example of instrumented loop

data that are not useful. On the other hand, the hardware mechanism can prefetch data that have been replaced due to conflict misses.

3.2 Prefetch Instruction and Predicate

Once a potential cache miss has been identified, the software scheme inserts a prefetch instruction. If accesses have spatial or group locality in the same cache line, only the first access to the line will result in a cache miss and only one prefetch instruction should be inserted. However, testing for this condition, i.e., computing a prefetch predicate, can be very expensive mostly if it occurs in an inner loop. Instead, the compiler will generally perform loop splitting and loop unrolling (or loop peeling).

The instrumented code of the previous example is shown in Figure 2 (b). We assume that a cache line holds two array elements, and that the memory latency requires the prefetch to be scheduled four iterations ahead. We split the original loop in three sections: prologue, main, and epilogue. The prologue prefetches the initial data set for the first four iterations. The main loop consists of the largest portion of the loop execution where the loop is in a steady state, that is, the demand of data can be satisfied by those prefetches occurring several iterations ahead. Finally, the epilogue finishes the last four iterations without any prefetching. After the original split, the loops are unrolled by a factor of two in order to eliminate the execution of the prefetch condition (*i mod* 2) = 0. One consequence of loop splitting and unrolling is that the code will expand significantly. This may result in an increase in the I-cache miss ratio and may introduce extra spilling of store/load instructions due to an increase in register pressure.

Another potential difficulty is that the prefetching references are not necessarily aligned on a cache line boundary (cf., X[i] and Y[i+1] in the example).

An advantage of the hardware scheme is that it executes the original loop without modification. However, at least two iterations are required before obtaining correct strides. There is no equivalent to the prologue as the hardware scheme prefetches the (initial) data by letting the LA-PC move gradually several iterations ahead of the PC. When the loop is in steady state, i.e., in the main loop, prefetching is performed in a similar way in both schemes. One important drawback of the hardware approach is that the system still continues to prefetch data even in the last iterations (corresponding to the epilogue), since the hardware is unable to know when the loop will end.

3.3 Scheduling Prefetches

Prefetches should be issued early enough to hide memory latency but not too early so that they do not displace useful data or are replaced before use. The software algorithm usually schedules prefetches ahead by a number of iterations: $\lceil \frac{\delta}{s} \rceil$ where δ is the memory latency and s is the estimated execution time of the loop body. As a result, the software scheme prefetches a data item at least one iteration before it is used. The prefetch is usually placed immediately before or after a corresponding reference to minimize the computation cost of the effective address. A window of vulnerability is left open between the arrival time of the prefetched data and its actual use. During this time window, the prefetched data can be displaced or it can displace some data that is accessed during that time.

The hardware scheme has a more flexible scheduling. As in the software scheme, a prefetch can be identified several iterations ahead if the memory latency is greater than the loop execution time (recall the *times* field in the RPT). If the latency is small, as for example in a multi-level cache hierarchy, the prefetching can occur in the same iteration as the load. Thus, in general, the data will arrive at the cache at a time closer to its actual use than in the software scheme. Therefore there will be fewer unwanted replacements in the cache, and the prefetches will be more spaced in time. We should note however a drawback of the hardware scheme, namely its reliance on good branch prediction to predict the look-ahead stream.

A potential advantage of the software scheme is that more aggressive program-specific prefetches can be supported. The software solution may be able to provide more flexible prefetching, such as pointer-chasing for linked lists, block prefetches (the prefetching size being determined in terms of semantic object instead of cache line size), and can take advantage of data reorganization. Mowry and Gupta [13] have shown the success of several strategies by code-specific and programmer-directed techniques. However, it is still unknown whether the techniques can be automated for general applications without programmers' intervention.

3.4 Prefetching in Multiprocessors

Thus far we have been focusing on prefetching for uniprocessors. When we consider a multiprocessor environment, additional factors come into play: (1) prefetches increase memory traffic, (2) prefetching of shared data items can bring additional coherence traffic, (3) invalidation misses are not predictable at compile time, and (4) dynamic task scheduling and migration policies are detrimental to the efficiency of prefetching.

The first factor, additional memory traffic, stems from the prefetching of unnecessary data (mostly in the hardware scheme) and from the early displacement and later recall of useful data (mostly in the software scheme). The increased memory traffic has more of a performance impact

in a multiprocessor environment since it contributes to the saturation of the interconnect between the processors and main memory. Tullsen and Eggers [18] have shown that the prefetching benefits are limited if memory bandwidth is a primary resource (e.g., in a bus-based shared memory multiprocessor). We will examine in Section 5.3 the issue of increase in the memory traffic when the available bandwidth is not as limited as that of a single shared-bus.

The increase in coherence traffic is difficult to avoid in both approaches. The problem arises from the same sources as that of the increase in memory traffic. A prefetched data item may need to be invalidated before it is used and an exclusive-prefetch causes invalidation misses on data that might yet have to be used in other processors. If a relaxed consistency model is assumed, write propagations are usually delayed until synchronizations. In this case, the first situation is equivalent to the attempt at controlling data that arrive at the cache just in time for its use. The second situation occurs when there is high contention for some shared writable data. Approaches, such as binding prefetch [9], avoid the problem by suppressing prefetches that may have data and control dependencies of accesses in other processors but they are overly conservative.

The fact that invalidation misses are not predictable at compile time is a weak point of the software approach, since it lacks the dynamic information necessary to initiate prefetches for missing data that have been invalidated. Restructuring the data to alleviate the effects of false sharing [6] might partially remedy the situation. On the other hand, the hardware scheme should be able to fetch back the data that were invalidated, if the state information mandates the prefetching.

Prefetching in parallel programs scheduled statically can be handled by both approaches. Dynamic task scheduling and task migration, and in particular fine grain task scheduling, will be very detrimental to the efficiency of prefetching since processor assignments may change before the prefetched data in the cache has been used. The problem is more critical to the hardware scheme which requires past access histories stored in the RPT, a cache-like table.

In summary, the software scheme is able to identify access locality for generating prefetches in the domain of linear array references at compile time, whereas the hardware scheme dynamically determines when and what to prefetch. However, the software scheme may suffer from a code expansion problem while the hardware scheme cannot predict as well the usefulness of the data it prefetches. The two approaches face the problems of increasing memory and coherence traffic in a multiprocessor environment.

4 Quantitative Evaluation Methodology

4.1 Architectural Models

The architecture that we assume is a shared-memory multiprocessor. It includes 16 MIPS R3000-like processors connected to memory modules through an interconnection network. We assume that instructions and private data references hit in a local memory with the processor incurring no time penalty. The cache hierarchy is used only for storing shared data. Cache coherence is maintained using a full directory protocol [2] distributed among the memory modules. Prefetched data are put into the caches so that the data still remain visible to the cache coherence protocol.

We experimented with three architectural choices: baseline caches, caches with hardware prefetching, and caches with software prefetching. In prefetching caches, prefetching was performed for read misses only, not including exclusive-prefetching. The default consistency model is weak consistency [5] under which most of the write latency can be hidden. Each cache-network interface has a prefetch issue buffer which can hold up to 16 prefetches. The prefetch request will check the tag directory in the cache and will be forwarded to the memory system if there is no matched cache line. When the buffer is full incoming prefetches are just discarded. Each processor has a 64K-byte data cache, which is direct-mapped and copy-back with a cache line size of 16 bytes. The caches are *lockup-free* [12], thus allowing multiple outstanding data requests. A 16-entry outstanding request list (ORL) is used to keep track of pending requests, some of which might then become *hit-wait* accesses when an actual load hits on a pending request but still stalls waiting for the data. The stall time incurred by a hit-wait access will be referred to as *hit-wait* time. Reads on actual misses or on hit-wait accesses are blocking.

We assume that the memory bandwidth is sufficient and that a fixed latency time is used when a request travels through the network. The one-way latency time between caches and the global memory modules is 40 cycles. Hence, a reference that misses in caches incurs a total latency of at least 80 cycles (L_m). A read miss to a dirty block owned by another cache or a write request to a block that is already cached elsewhere will need at least two network round trips, i.e., 160 cycles. Although we do not model the contention in the network, we do take into account interference at the caches and at the memory directories since each cache and directory module can process only one request per cycle. Lock/unlock and barrier requests are handled using a queue-based protocol in the directory. A request waiting on a synchronization operation will not cause extra traffic for the caches and the network.

4.2 Benchmarks

We developed a direct-execution simulator that simulates important events of interest in a shared-memory multiprocessor, while the computation instructions are directly executed by the host machine. The benchmarks we used are Matmat and three SPLASH benchmarks [16]. Table 1 summarizes the statistics collected on these benchmarks once their parallel sections are started up to the completion of the program. Only shared references are recorded in the table and the column below ''shared data size'' indicates the total size of global shared area which is explicitly allocated in the program. Matmat is a blocked matrix multiplication program, run with two 300×300 matrices with proper cache buffer and block setting so that the effects of cache size and block size can be balanced. Mp3d is a particle-based fluid flow simulation program. We ran Mp3d with 100,000 particles in a $14 \times 24 \times 7$ space array for 10 time steps. Water, an N-body molecular application, was run with 288 molecules for 4 time steps. Cholesky performs parallel factorization of a sparse matrix, run with the test set `bcsstk15`.

In order to implement the software scheme, we instrumented the original SPLASH benchmarks. Through profiling, we identified those accesses with the highest cache miss rates. The instruction addresses of the cache misses (candidates for prefetching) were recorded by running each program on a configuration similar to that of the prefetching study and with the same data set. We surmise that this

Table 1: Benchmarks characteristics - average numbers for a single processor in the 16 processor simulation

	Inst (K)	shared reads (K)	shared writes (K)	shared data (Kb)
Matmat	8,723	1,355	421	2109
Mp3d	7,231	1,334	426	3673
Water	36,036	1,609	162	195
Cholesky	38,233	6,809	524	6403

approach allowed us to determine realistic prefetching candidates as well as a thorough compiler analysis. For instance, a reference which has temporal locality in a loop will not be prefetched because of its low miss frequency.

After the accesses for prefetching are identified, we manually insert prefetch instructions related to these high miss frequency items based on the following strategies:

1. A data item accessed in a loop is prefetched one or more iterations ahead depending on the relative values of the estimated loop execution time and the memory latency.

2. Taking the block size into account, we perform loop unrolling and loop splitting. Additional spilled code resulting from an increase in register pressure will contribute to the prefetching overhead.

3. By default, each prefetch will bring one cache block. If our profiling information detects that prefetching a whole data object at once would be beneficial, block prefetching is performed. An additional instruction is needed to specify the prefetch size in that case.

4. If a prefetch is originated from an indirect load, we attempt to schedule the source load ahead in the instruction stream to provide as large a non-blocking span as possible. The address computation of prefetch instructions is generally combined with that of the corresponding loads thus resulting in no overhead. However, when the prefetches are moved away from their loads the cost of computing prefetch address expressions cannot be completely eliminated.

In summary, we emulate a compiler algorithm that will carefully generate effective prefetches. The overhead in our implementation is relatively low (just over one instruction per prefetch instance).

5 Quantitative Evaluation

5.1 General results

Figure 3 shows the simulation results of the average execution time of the 16 processors with respect to various approaches. The left-most bar shows the breakdown of the execution time of the baseline cache (BASE). The next two bars are for the hardware (HW-pf), and software (SW-pf) schemes respectively. We present the data by normalizing the total execution time with respect to the baseline organization. Each bar contains several sections. The *exec* section denotes the time to execute instructions -- it also includes the extra instruction overhead for executing software prefetching instructions, necessary address/size computations, and execution of possible extra spilling loads; *read* and *write* indicate the fraction of processor stall time for reads and writes; *delay* shows the delay of demand accesses resulting from handling prefetch and tag updates in the cache; and *synch* gives the time waiting for lock and barrier accesses.

Let us examine each stall time component. The instruction execution time, corresponding to processor utilization, is between 13% in Mp3d and 75% in Water. By looking at the results for BASE, we note that there is much room for improvement for reducing the read access penalty. This is borne out by the results showing remarkable reductions in read stall time for both schemes: 10%-39% for HW-pf and 15%-43% for SW-pf of the original total cycles.

The portions of stall time due to writes and synchronizations are almost negligible in the BASE case. Writes can be efficiently buffered since we operate under a weak consistency model. The stall time due to synchronizations is very small in all cases except Water where it reaches 5.5%. Neither HW-pf nor SW-pf modify significantly these figures. The last component in the overall execution time, i.e., the delay due to contention in the cache between prefetch and regular accesses, is clearly an overhead introduced by the prefetching. As seen from the *delay* section in Figure 3, the number of cycles lost because of this interference are very small (only 0.05%-0.6%). Hence, this side effect is almost negligible.

Extra instruction execution time is yet another overhead, which is present only in SW-pf. As shown in the *exec* section of SW-pf, the SW-pf instruction overhead can be substantial. The portion of normalized time due to the software overhead ranges from 0.9% in Mp3d to 8.6% for Matmat and may offset part of what was gained in reducing the read penalty.

5.2 Detailed Analysis

We examine further the effectiveness of prefetching by looking in more detail at the individual behavior of the four benchmarks (cf. Table 1).

Matmat is a blocked matrix multiplication program in which almost all references are regular and sequential. Both HW-pf and SW-pf perform quite well on the Matmat benchmark since data access patterns are regular (read penalty reduced by 77% and 87% respectively). Even so not all of the read penalty has been eliminated. In HW-pf some of the read penalty is contributed by a portion of *hit-wait* cycles in the first iterations. Another portion of the remaining read penalty stems from the fact that the blocking technique tries to localize the referenced domain of inner loops and thus data blocks prefetched at the last iteration of an inner loop are generally unused. Similarly, SW-pf has a portion of *hit-wait* cycles. Moreover, the loop splitting introduced because of the prefetching increases the register pressure that is already very tight because of the tiling of the inner-most loop. Looking in more detail at SW-pf shows that the execution time of one iteration of the inner-most loop (unrolled by a factor of 2) takes 85 ideal cycles. It has been increased by 11%, compared with the execution time of the original code (76 cycles for two iterations). The increase comes from the prefetch instructions and extra spilling code. This explains the magnitude of the instruction overhead (8.6% of total time) for SW-pf. It indicates that SW-pf should be more conservative when taking into account optimizations arising from locality considerations.

In **Mp3d**, the two data structures that account for most of

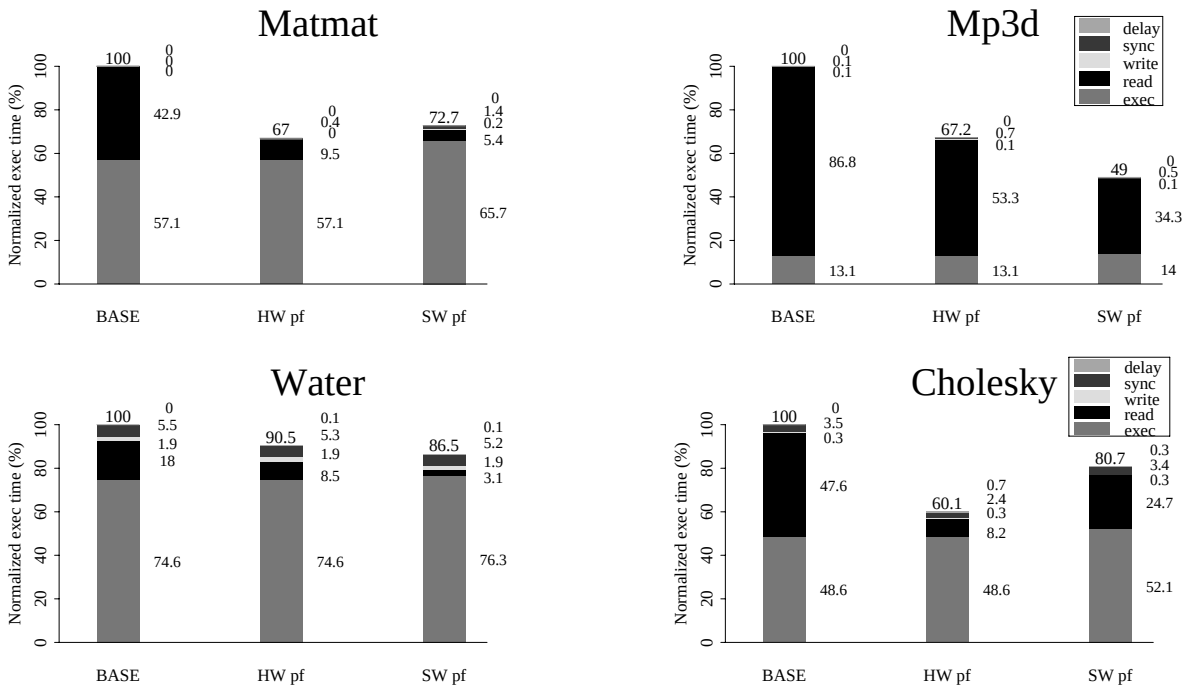

Figure 3: Simulation results

the references are particles and space cells. The particles are statically allocated; the space cells are accessed in a relatively random manner depending on the location of the particle being moved. In such an application where data structures are more complex, SW-pf exhibits better performance in reducing the read penalty than HW-pf (38% for HW-pf in Mp3d vs. 60% reduction for SW-pf). Although HW-pf has no difficulty in prefetching a particle record, it is not good at dealing with space cells because their locations vary with time. Thus only roughly half of the cache misses are covered through HW-pf. In contrast, SW-pf performs much better than HW-pf. SW-pf can statically prefetch particle data and use indirect load prefetches to get the space cell when the address of an associated particle is determined. Moreover, particle objects and space cells can be prefetched by a single block prefetch instruction. Consequently, several memory access requests triggered by only one prefetch instruction can be pipelined to the memory system. The prefetching of space cells is scheduled so that it can be performed in parallel with other computations. Therefore the latency of the indirect load prefetch is hidden further. The use of block prefetches is also the reason that Mp3d has a negligible instruction overhead.

In **Water**, the main data structure is an array of molecules where each element holds all the data for one molecule. Each molecule requires about 38 cache lines. Data accesses preserve spatial locality in the *intramolecular* computations and data access patterns are predictable in the *intermolecular* computation phases. Since the ratio of the number of shared references to instructions is very small, the instruction time accounts for a large portion of the total execution time (cf. Table 1). In addition because the cache can hold almost the entire working set, most of the accesses result in cache hits. Therefore the read penalty contributes only 18% of the total execution time. In this benchmark with predictable access patterns and small nested loops, the read penalty reduction is very good but does not improve performance that much since the read penalty is relatively small. SW-pf moderately outperforms HW-pf (52% for HW-pf vs. 83% for SW-pf). Both schemes can easily handle the shared references in the *intra* and *intermolecular* computation phases. The main reason for the superiority of SW-pf is that each computation of a molecule involves two or three nested small inner loops with only a small number of iterations in each level of loop. SW-pf simply prefetches data for all the iterations at one time, whereas the small loops hinder HW-pf from gaining sufficient prefetching distance.

Cholesky is dynamically scheduled with coarse task granularity (about 86,000 shared references per task). Each task works on supernodes, which are sets of columns of a very large sparse matrix. The input data file is a 3948-by-3948 matrix with only 56934 non-zero elements. The primary operation is a column modification algorithm which involves the addition of two columns in order to cancel a non-zero element in the upper triangle of the matrix. Since all non-zero elements belonging to a certain column are stored contiguously in an array and the row numbers of these non-zero elements are stored in a compressed manner, the program iterates on the array of row numbers to find matching rows and then fetch the non-zero elements to perform the computations. As a result, the starting and ending values of loops are generally unknown at compile time. In this benchmark, the hardware scheme performs better than SW-pf (82% vs. 48%). The HW-pf scheme can benefit from the assignment of large supernodes to the processors by sequentially prefetching the array and dynamically extracting data access patterns for the accesses of non-zeros. Similarly, SW-pf can prefetch the data for accesses to the array holding row numbers. However, our implementation is conservative in prefetching the non-zeros by using indirect load prefetches only after the

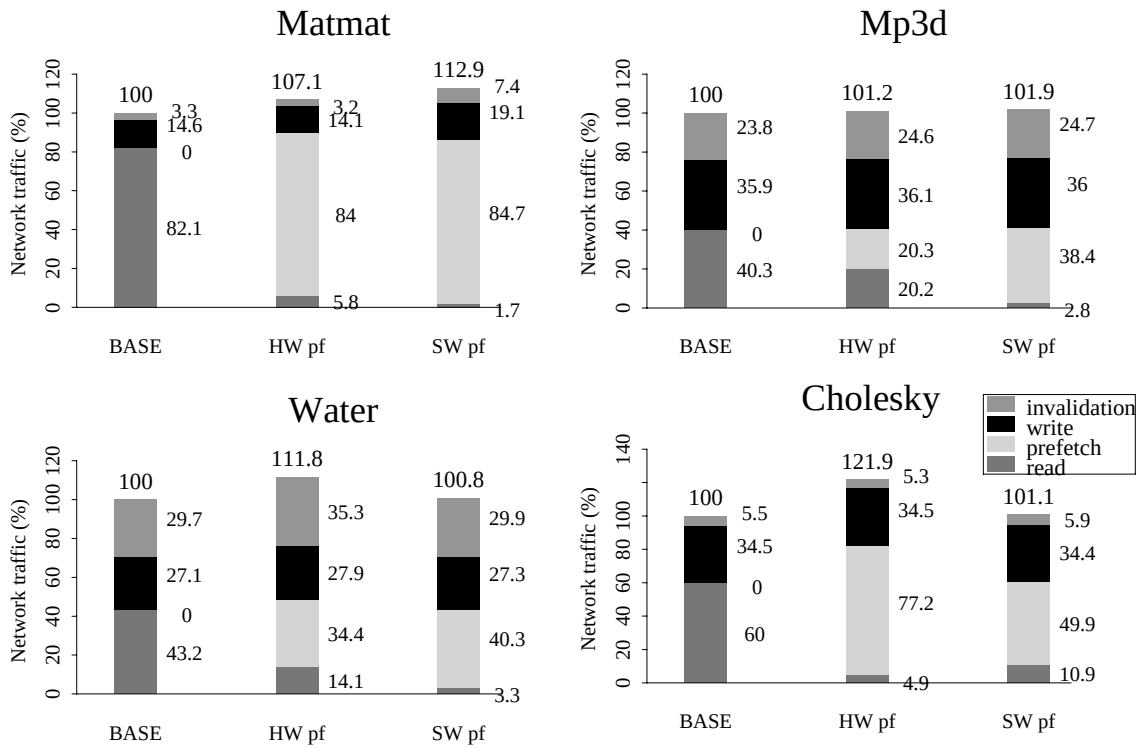

Figure 4: Network traffic

row pointer is known[1]. This will usually cause prefetched blocks to arrive in the cache too late and thus to contribute a large portion of *hit-wait* cycles to the read penalty. In addition, because the starting and ending values are run-time variables, the code is significantly expanded as a result of loop unrolling and splitting as well as prefetch insertion. For example, an IF statement is required in the prologue to align the prefetch access on the cache line boundary. Hence, the instruction execution time is increased.

To summarize, our data show that SW-pf and HW-pf can achieve good performance improvements in programs with regular access patterns. HW-pf can handle applications with input data dependence if the loop granularity is not too small. SW-pf is flexible and can deal with programs with complicated but well-organized data structures. However, the benefit of software prefetching may be offset by the extra overhead it incurs.

5.3 Negative Effects of Prefetching

As mentioned earlier, prefetching increases memory traffic. The main sources for the increase are: (1) prefetches of unused data lines, (2) extra cache misses due to conflicts with the current working set, (3) extra invalidates due to additional write-sharing caused by prefetching, and (4) the increase of invalidation misses due to exclusive prefetches. Since we do not perform prefetch for writes, the last problem does not occur in our study.

In Figure 4, we present the increase in network traffic. We consider four kinds of requests for the network: read misses, prefetch requests, write requests (write misses and write hits on clean), and invalidates. While the number of memory requests increase, as expected, for both types of prefetching for all benchmarks, the increase due to prefetching (especially SW-pf) is relatively insignificant with respect to the total traffic. Most of the memory traffic increase stems from the fact that the total requests of read misses and prefetches are greater than those of read misses for the baseline cache. Since prefetching may fetch write-shared data, a slight increase of write requests and invalidates can be also observed in the figure. In general, SW-pf is more conservative in introducing memory traffic than HW-pf. The reasons are that HW-pf has less information to avoid sending unnecessary prefetches to the system and that data blocks prefetched during the last iterations are generally unused. The traffic increase is more significant in benchmarks with small iterations, such as Water, where the penalty reduction by HW-pf is less than that by SW-pf, but where HW-pf brings more network traffic. One exception is Matmat, where SW-pf results in more network traffic than HW-pf. However, the increase is mainly because more writes and invalidates are issued since there is more prefetching of write-shared data.

To examine the impact of prefetching on the working set in the cache, we estimate the negative effect by measuring conflicts between the working set and prefetched data. We record the information on replaced data lines in a ‘‘shadow’’ direct-mapped cache with the same size as the data cache. If a cache miss finds a matched entry in the shadow cache, we record the status of both replaced and current blocks. As most cache misses are reduced by prefetching, we are interested in conflict misses. Table 2 gives the proportions of those conflict misses among three categories: conflicts within the current working set itself, between the working set

[1] A referee has pointed out that Cholesky has been rewritten so that the compiler can deal with this problem.

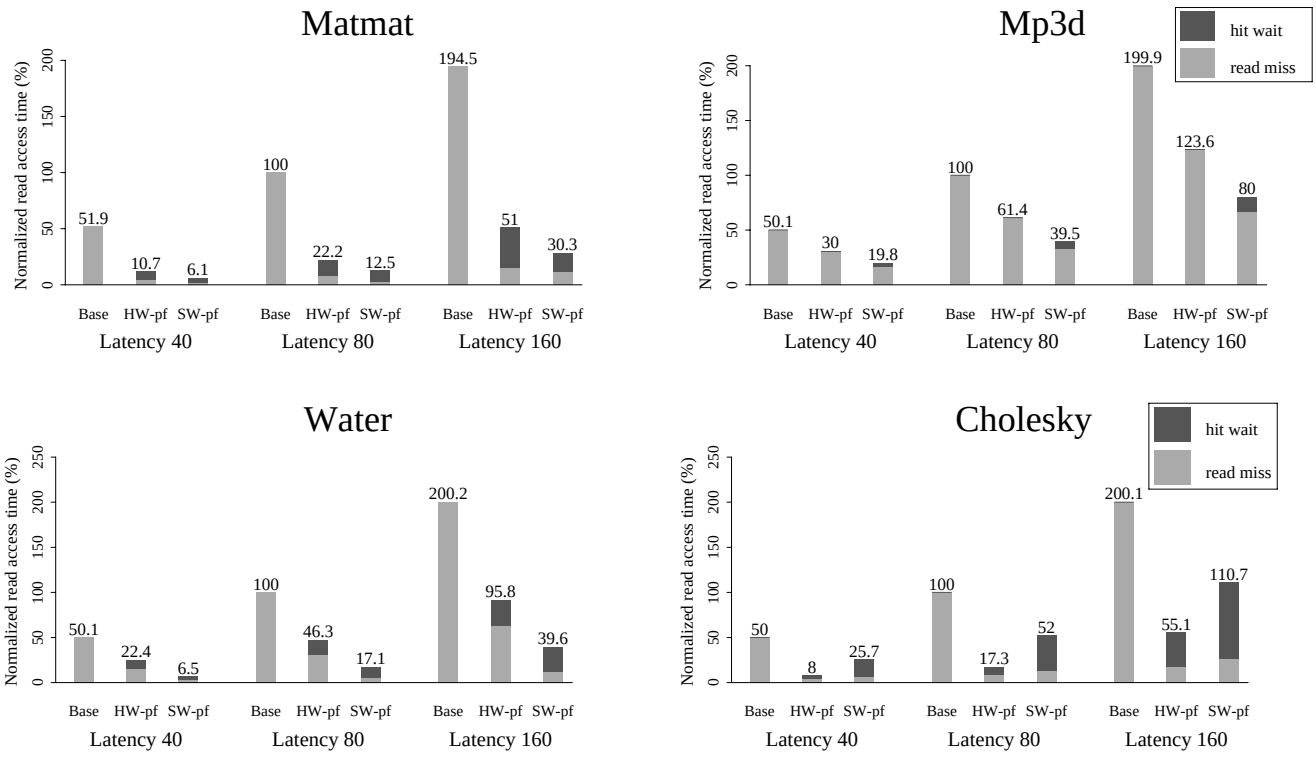

Figure 5: Effect of memory latency

Table 2: Proportions of conflicts in direct-mapped cache

Programs	Hardware			Software		
	ws ⇕ ws	ws ⇕ pf	pf ⇕ pf	ws ⇕ ws	ws ⇕ pf	pf ⇕ pf
Matmat	.924	.076	0	-	-	-
Mp3d	.976	.024	0	.925	.062	.013
Water	-	-	-	-	-	-
Cholesky	.710	.174	.117	.961	.034	.005

and prefetched blocks, and between prefetched blocks themselves. In the table, the miss ratio of software prefetching for Matmat is very small (< 0.001) and in Water, there are very few conflict misses left, since most of the data set fits in the cache and misses are mainly caused by invalidation misses. The results show that a large portion of conflicts occurs among data in the working set itself. When a prefetched item arrives in the cache at a time close to its actual use, the probability of conflicts with the current working set is small. It is only in the case of HW-pf in Cholesky that significantly more prefetched data than necessary is brought into the cache. In that benchmark we can observe a non-negligible amount of conflict between the prefetched data and the working set. This explains partially the increase of data read (read misses and prefetches) traffic in the network, as shown in Figure 4.

To sum up, we observe that the negative effect of prefetching in network traffic and conflicts with the working set is not severe. The increase of network traffic is very small for SW-pf, whereas HW-pf may give a slight increase. Most conflict misses are caused by the working set itself.

5.4 Effect of Memory Latency

In this section we explore how variations in the secondary cache and main memory latencies influence the performance of the three prefetching schemes. We consider three sets of latencies: the one used previously (L_m = 80), one where we consider a processor twice as slow (L_m = 40), and one where the main memory latency is doubled (L_m = 160) with the rationale here that our 16-processor system might be a subset of a larger multiprocessor. In Figure 5, we show the read access times for these three organizations normalized with respect to the no-prefetch BASE default case (L_m = 80). The read access penalty is decomposed into two sections: *read miss*, the stall time due to cache misses, and *hit-wait*, the waiting time for a prefetch which is issued too late. In order to have a fair comparison for SW-pf, we modified and moved around some prefetch instructions in an attempt to provide a sufficient prefetching span for large latencies.

As can be seen in Figure 5, the reduction in the read penalty slightly degrades as the memory latency increases. This illustrates that both HW-pf and SW-pf still can be effective, to a lesser extent, in tolerating large latencies by adjusting prefetching to occur several iterations ahead of the actual use. Note that since the number of instruction executed is generally fixed, the slight increase in the read penalty in SW-pf is more than compensated by the relative decrease in the overhead of the prefetch instructions. For example, when passing from L_m = 80 to L_m = 160, the overall execution time increases and the overhead from software prefetching (not shown in the figure), an almost constant number of instructions for each benchmark, decreases from 8.6% to 6% in Matmat, from 0.9% to 0.04% in MP3D, from 1.7% to

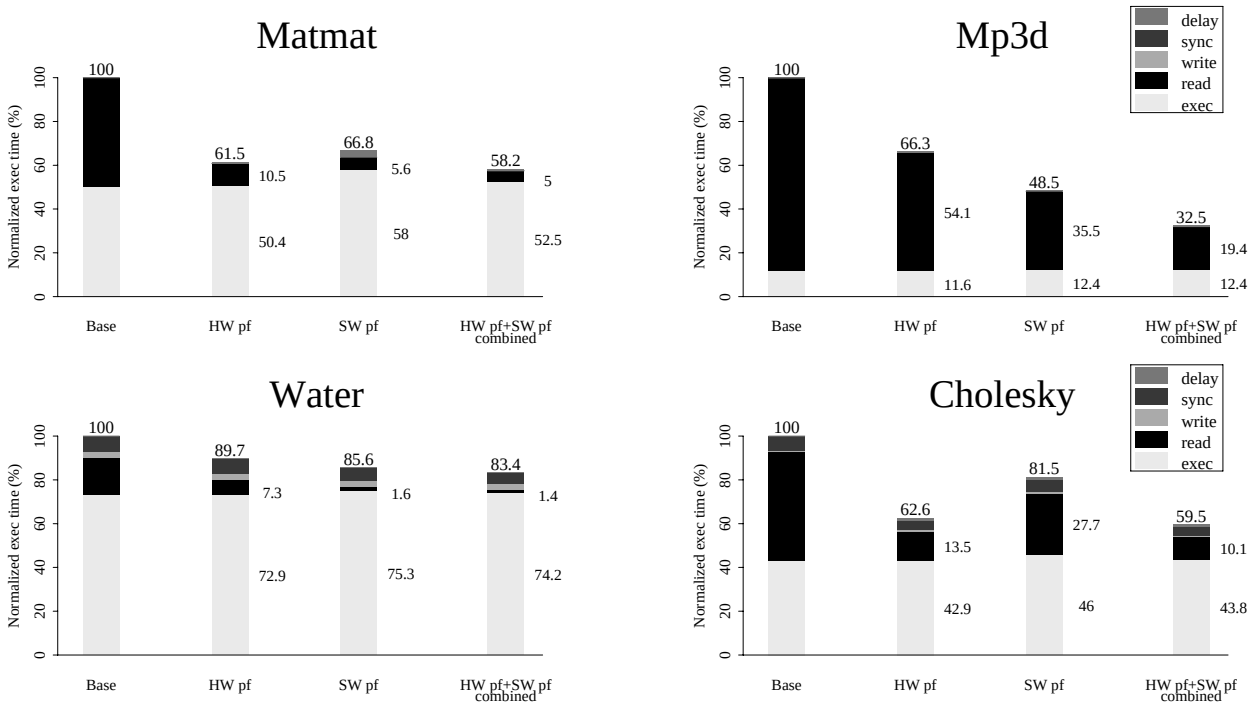

Figure 6: Effectiveness of combining HW-pf and SW-pf

0.013% in Water, and from 3.5% to 0.3% in Cholesky. This leads us to conjecture that software prefetching should be more advantageous as the prefetch overhead becomes less significant with an increase in latency.

The cost of the *hit-wait* cycles is particularly important in prefetching. The read penalty in HW-pf contains a fair amount of *hit-wait* time. In this scheme, the lookahead mechanism needs to be reset to the value of the PC after each incorrect branch prediction. Therefore, the first few prefetches are not yet one ''memory latency time'' ahead of when their data will be used. This phenomenon tends to be serious in those programs with nested inner loops with only a few iterations such as Matmat and Water. For SW-pf (cf. Cholesky), the *hit-wait* cycles are mostly contributed by the indirect load prefetches, which are constrained by the data dependencies. While SW-pf is generally able to identify most of the cache misses, the stall time for prefetches in the prologue loop becomes more significant when the latency increases. There remains the challenge of scheduling useful computations to overlap with the prefetches, a task that becomes more difficult as latencies get larger.

6 Combining Hardware and Software Prefetching

In this section, we propose a combination of hardware and software prefetching techniques. The main idea is that the compiler inserts prefetches for user's semantic data objects that can be of any size, not necessarily a cache line, in a manner more related to the program information available to the compiler, and that the hardware supporting unit takes care of individual element accesses in loops. The advantage of this combined scheme is that the amount of software prefetching instructions is considerably reduced and loop splitting can be avoided. Thus, the compiler simply finds a proper prefetching program point for each data object to be used in loops. To achieve maximum gains, the hardware part is aimed at prefetching data from the secondary cache to a relatively small primary cache -- a portion of the design space where it is recognized that the hardware scheme performs best [1] -- and the software part is aimed at a large block fetch from memory modules to the secondary cache. By adding a special control instruction to the instruction set, some unnecessary prefetches in the hardware prefetching scheme can be further reduced by using the instruction as a control hint to enable (and disable) the hardware mechanism. Such control hints can be inserted around a loop body so that the hardware unit will operate only during loop execution.

We performed experiments for studying the effectiveness of the HW-pf and SW-pf combined architecture. In the experiment, we consider an architecture similar to the previous ones, except that each processor has a 32K-byte primary cache (C1) backed up by a 256K-byte second-level cache (C2). Both caches are direct-mapped, copy-back with a cache line size of 16 bytes and are *lockup-free*. The one-way latency time between C1 and C2 is 5 cycles and thus the delay for a miss in C1 with a hit in C2 is 10 cycles. Misses in C2 trigger requests to the global memory modules. The one-way network latency is 35 cycles. Hence, a reference that misses in both caches incurs a total latency of at least 80 cycles as before. In the experiment, we modify the strategy for prefetch insertion in software prefetching: we do not prefetch data in inner-most loops, we do not perform loop unrolling and splitting, we insert prefetches for user data structures to be used (regardless of cache size, line size), and we move prefetches far ahead of actual use (they may even move to locations before the loop).

Figure 6 gives the simulation results of the new architecture with the combined hardware and software schemes. The read access penalty has been further decreased when compared

to either the hardware approach or the software approach. The total reductions of the read penalty are 90% for Matmat, 78% for Mp3d, 88% for Water, 80% for Cholesky. The instruction overhead of the new scheme is relatively small when compared with the software approach (note that SW-pf already performed block prefetching in Mp3d). The portion of total normalized time due to the overhead ranges from 0.8% in Mp3d to 2.1% in Matmat. Overall, the total execution time is significantly improved by the combination of software and hardware schemes.

7 Conclusion

In this paper, we have studied the performance of hardware-based and software-directed prefetching schemes. Our qualitative comparisons indicate that in the domain of linear array references, both hardware and software schemes are able to generate prefetches for cache misses. However, the software scheme may have a code expansion problem, while the hardware scheme has less information on the usefulness of the prefetched data. The software approach may use compile-time information to perform sophisticated prefetching, whereas the hardware scheme has the advantage of manipulating dynamic information.

The quantitative evaluation was performed by running direct-execution simulations of a shared-memory multiprocessor using four benchmarks. Our experiments confirm the above observations. We observed that the cache interference incurred by prefetching is almost negligible. The software approach has less negative effect on network traffic and conflicts with the working set than the hardware approach. However, the overhead due to the extra prefetch instructions and associated computations is substantial in the software-directed approach and can offset the performance gain of prefetching. Our results show that the relative effectiveness of prefetching is slightly degraded by the increase of memory latencies, with the software prefetching suffering less.

Finally, we proposed and examined a technique for combining the software and hardware solutions. The main idea is that software will use program user's semantics to prefetch data objects into a secondary cache and that the hardware supporting unit will take care of accesses in the loop and fetch the data into the primary cache. The new approach can combine advantages of both hardware and software approaches and at the same time avoid most of their negative effects. Our experimental results show that the new solution is very attractive in reducing the data access penalty without incurring much overhead.

References

[1] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proc. of Supercomputing '91*, pages 176--186, 1991.

[2] L. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, C-27(12):1112--1118, 1978.

[3] T.-F. Chen. *Data Prefetching for High-Performance Processors*. PhD thesis, Department of Computer Science and Engineering, Univ. of Washington, 1993.

[4] W. Y. Chen, S. A. Mahlke, P. P. Chang, and W.-M. Hwu. Data access microarchitectures for superscalar processors with compiler-assisted data prefetching. In *Proceedings of the 24th International Symposium on Mircoarchitecture*, 1991.

[5] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *Proc. of the 13th Annual Intl. Symp. on Computer Architecture*, pages 434--442, 1986.

[6] S. J. Eggers and T. E. Jeremiassen. Eliminating false sharing. In *Proc. of the Int. Conf. on Parallel Processing*, pages I:377--I:381, 1991.

[7] J. W. C. Fu and J. H. Patel. Data prefetching in multiprocessor vector cache memories. In *Proc. of the 18th Annual Intl. Symp. on Computer Architecture*, pages 54--63, 1991.

[8] J. W. C. Fu and J. H. Patel. Stride directed prefetching in scalar processors. In *Proc. of the 25th Int'l Symp. on Microarchitecture*, pages 102--110, December 1992.

[9] E. Gornish, E. Granston, and A. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *Proc. 1990 Intl. Conf. on Supercomputing*, pages 354--368, 1990.

[10] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proc. of the 17th Annual Intl. Symp. on Computer Architecture*, pages 364--373, May 1990.

[11] A. C. Klaiber and H. M. Levy. An architecture for software-controlled data prefetching. In *Proc. of the 18th Annual Intl. Symp. on Computer Architecture*, pages 43--53, 1991.

[12] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proc. of the 8th Annual Intl. Symp. on Computer Architecture*, pages 81--87, 1981.

[13] T. Mowry and A. Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87--106, June 1991.

[14] T. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algoritm for prefetching. In *Proc. of the 5th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 62--73, 1992.

[15] A. K. Porterfield. Software methods for improvement of cache performance on supercomputer applications. Technical Report COMP TR 89-93, Rice University, May 1989.

[16] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20(1):5--44, March 1992.

[17] A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473--530, September 1982.

[18] D. M. Tullsen and S. J. Eggers. Limitation of cache prefetching on a bus-based multiprocessor. In *Proc. of the 20th Annual Intl. Symp. on Computer Architecture*, 1993.

[19] M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *Proc. ACM SIGPLAN 91 Conference on Programming Language Design and Implementation*, pages 30--44, 1991.