Data Prefetching: A Cost/Performance Analysis

Chris Metcalf*
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract

This paper attempts to answer the question, "To what extent is prefetching effective in hiding memory latency, and what is the minimal amount of hardware required to support prefetching?" We begin by providing a classification of the different kinds of prefetching, and reconciling the various common performance metrics to allow fair comparisons. We then put forward an analytical model that gives the potential speedup with prefetching. We next detail the non-binding software prefetch technique and examine its performance, both with hand-inserted and compiler-inserted prefetches. We consider an elaborate hardware scheme meant to replace the software schemes entirely; then look at more reasonable schemes requiring only minimal extra hardware, and assess how much they add to the simple software prefetching model. We conclude with recommendations for CPU/cache architects.

1 Introduction

As processor speeds have increased over time, managing the memory hierarchy has become increasingly more important. Ten years ago, a VAX 11/780 had memory faster than the average instruction time [15]. Today, a typical CPU may be 20 cycles away from local memory, and a hundred or more cycles away from memory in a distributed multiprocessor. Increasing the cache size helps to mask part of the problem, but the initial reference to a given memory address will still require a long processor stall no matter how large the cache, and real data sharing among processors will still require that data traverse the machine.

Prefetching can be used to eliminate those latencies; the compiler, or the processor, will try to bring the address to be referenced into the cache before it is needed, effectively parallelizing memory access with productive computation. In principle, most memory references can be predicted in advance; some can be predicted a long ways in advance, like regular-stride array accesses, while others, like linked-list accesses, can only be predicted shortly before their use. Even when the full latency can't be masked, prefetching can be used to shorten the latency for the references that would otherwise by delayed. As processors get faster and multiprocessors get larger, latencies will increase, and the advantages of using prefetching will increase as well.

Many different solutions have been proposed to predict references enough in advance to prefetch them. A variety of hardware schemes have been suggested, ranging from just lengthening cache lines [3, 19, 30] to simple next-block prefetching [3, 4, 10, 29, 30, 31] to complex lookahead mechanisms such as stream-buffer preloading [15], instruction pipeline prefetching [18, 19, 20], and vector-style prefetching [8, 9]. Compilers are also implementing prefetching based on program analyses of increasing sophistication. Simple prefetching was available in hardware in the late '70s (*e.g.*, in the IBM 370/168 [29]), but only recently have microprocessors begun providing the necessary hooks to allow prefetching to take place. Such processors include the IBM RS6000 [13], Motorola 88100 [21], the DEC Alpha family [28], and the MIT Alewife group's Sparcle, among others.

In this paper, we will begin by laying out a groundwork to assess various schemes, beginning in Section 2, and including an attempt to disentangle the various metrics used in prefetching papers. In Section 3, we will consider a simple analytical model to predict the possible benefits of prefetching. Section 4 details a method of software prefetching into the cache, looking first at hand-insertion of prefetches to assess the technique's potential, and then examining two compiler algorithms that implement software prefetching. Section 5 discusses an alternative hardware technique that attempts to replace software methods completely. In Section 6 we explore the possibility of adding a small amount of additional hardware to improve performance, and then in Section 7 we examine some possible refinements to the proposed hardware. We close with an assertion of how much hardware really is necessary to get good performance from prefetching.

2 Background

Prefetching, broadly speaking, means shortening the time useful computation must spend waiting for data. In this section, we first lay out the criteria that will be applied to various prefetching models, and then present a taxonomy of prefetching. Finally, we examine how to handle the varying metrics used in the papers themselves.

When examing an architectural feature such as prefetching support, there are a number of important criteria to be applied. We will generally use cost/performance as our primary criterion, but will mention other criteria when their use would give different conclusions.

- Performance. If adding a given feature increases the overall speed of the programs you are interested in, it is typically beneficial. Our primary interest is with models that handle

*Area Exam. October, 1993.

longer latencies effectively (100 cycles or more), since that is the direction the technology appears to be heading. We compare some of the common performance metrics below.

- Cost. For this problem, cost is typically of the form of increased on-chip hardware requirements. Extra hardware has at least two drawbacks: firstly that space used for prefetching hardware could be used for cache memory or other processor functionality, and secondly that implementing complex schemes in silicon may significantly increase the time-to-market, negating at least partially any relative performance improvements of prefetching.

- Compatibility. In some markets, compatibility is an important concern; in that case, solutions that require modifying existing code to use prefetching may be less useful than solutions which can accelerate 'dusty deck' binaries.

Prefetching can in general be done in one of two ways. Either the hardware can attempt to do it all by itself, or it can be helped by software. The simplest example of pure hardware prefetching is the behavior of ordinary caches, which bring entire cache lines from memory when a single word in the line is referenced; by spatial locality, this usually results in bringing in data that will be needed soon. Alternatively, software can play a role: for example, in some current RISC architectures a load operation has one or more required wait states, and a good optimizing assembler will move a data load earlier and use the wait states for computation.

We can identify three major differences between hardware and software prefetching:

- Effectiveness: software prefetching can use global knowledge, typically compiler-derived, to place prefetches where they will do the most good, and avoid issuing them when redundant; by contrast, hardware prefetching can perform prefetching for run-time memory access patterns not visible to a compiler. However, hardware prefetching may also be limited by wired-in table sizes—both in its own mechanisms and in interactions with other hardware mechanisms, such as speculative execution—and may be difficult to match to different memory hierarchies or to uniprocessor or multiprocessor requirements.

- Overhead: software prefetching typically adds extra instructions into the code flow; hardware may slow the processor cycle time if it is implemented on the critical path.

- Complexity: software prefetching requires little hardware support (typically just a prefetch operation or non-blocking load of some sort), but significant compiler effort; hardware prefetching requires no software support, and thus supports 'dusty-deck' code compatibility, but does require chip area and a design effort with little room for mistakes.

Prefetching can also be divided into *binding* and *non-binding* types. Binding prefetching brings the data in and assigns it to a specific location, from which it can later be used—*e.g.,* a register or a location in local memory. By contrast, non-binding prefetching is a hint to the memory system to try to bring the given datum in to a closer, faster level of memory, such that a later binding load will complete much faster. While the issue of where the data is brought to (*naming*) is an important one, orthogonal to binding, most recent work considers non-binding prefetch as bringing data to the primary cache (or sometimes to a subsidiary prefetch buffer), and binding prefetch as bringing data to a named register.

Non-binding loads can be dropped on faults, or annulled by other mechanisms, while binding loads can't be without causing the application to notice. In general, non-binding prefetches allow much more flexibility and margin of error, since a prefetch of a bad address doesn't have any effect on the execution of the code, and a missed or aborted prefetch only causes a slowdown. In multiprocessors, non-binding prefetches allow fetching data even across synchronization boundaries, since cache coherency mechanisms will undo the prefetch if necessary. The ability to issue a prefetch earlier becomes more and more important as memory latencies increase. By the same token, of course, a valid prefetch can be evicted accidentally by other data, causing unwanted delays in program execution.

By contrast, binding prefetch mechanisms typically require less overhead, since no 'undo' mechanism is necessary. Binding prefetches also typically incur a smaller overhead of instructions issued, since a single binding prefetch may suffice to make the data available for computation. (With increasing data latencies, however, instruction bandwidth may become less of a problem; and some microprocessors, such as the HPPA, have provisions for non-binding prefetch as part of an instruction that has semantic content, such as `ADD`.) Binding prefetch can be used when a compiler can guarantee that a piece of data will be used; however, they also consume resources between the time the binding prefetch is issued and the time the data is used. For example, prefetching to registers is limited by the size of the register set—increasing the amount of prefetching increases the register pressure within the computation. Binding prefetch also requires loop unrolling if the code prefetches more than one loop iteration ahead, a limitation to which non-binding prefetch is not subject. Finally, note that binding prefetches are subject to aliasing problems unless explicitly handled by software or hardware.

The various papers in the literature assess all these mechanisms, but use a wide variety of different metrics; as a result, we often can't directly compare performance claims. We lay out the ones we are interested in below, and compare them.

- wall-clock time for the program to run from start to finish. This is the metric we want.

- CPI. This is almost exactly as useful as runtime, except for the occasional program which varies its behavior depending on how long it runs for—but this effect can be ignored.

- CPI due to data access penalty (CPI_{da}). This is expressed as

$$\text{CPI}_{da} = \frac{\text{total data access penalty}}{\text{instructions executed}}$$

  where the data access penalty is defined as the number of cycles spent above and beyond the number required if the code could run unmodified completely out of cache. The ideal value for this metric is 0. While this metric, and the next one, are good for evaluating memory hierarchy performance, they do not provide an appropriate value for computing overall cost/benefit ratios, unlike wall-clock time or CPI. For the ideal RISC machine, we can approximate CPI as $\text{CPI} = 1 + \text{CPI}_{da}$.

We don't model things like floating-point unit bandwidth; this can be estimated by using a base CPI > 1. Accordingly, our estimates of the influence of the memory hierarchy on run time will be optimistic.

- average memory reference time (t_{mr}). The ideal value for this metric is either 0 or 1, depending on the number of wait states to access local cache; for the papers discussed here, it is always 1. To get CPI from this value, we need to know the percentage of memory-referencing instructions in the original, unmodified code; we will call this p_m. Thus, to approximate CPI, we use $CPI = 1 + p_m t_{mr}$. If we don't know p_m for a given application, we have to estimate it; real codes range from 30 to 45 percent [22], and numerical kernels are typically 50 to 60 percent. Thus a t_{mr} improvement from 3 to 1.1 on a typical numerical kernel with $p_m = 0.5$ might translate to an improvement of CPI from 2.5 to 1.55. Again, with a base CPI of 1, our estimates of the influence of the memory hierarchy on run time will be optimistic.

- miss rate (m). This is a relatively poor metric, and not much used in more recent papers. It can't be converted directly into a runtime or CPI value without knowing a great deal about the structure of the application and simulation model in question. In general, miss rate may not be well correlated with real speedup [19], especially in multiprocessors, where decreased miss rate may come at an unacceptably high cost in terms of memory bandwidth. It can be converted to average memory reference time by multiplying by the latency to memory; however, particularly with prefetching, this latency is likely to be a function of prefetching success and overhead, network load, and so forth.

Furthermore, note that different papers typically use different simulation environments. Memory latency (and memory models) vary dramatically in different papers, which can change speedup results by factors of three or more [27]. Cache sizes are also often widely different, as are problem sizes. These factors can be partly compensated for by examining papers which examine their architecture under a wide range of cache sizes or memory latencies, and then applying rough scaling factors to other results.

Finally, it is worth bearing in mind that different papers target different models of computing. In many papers on prefetching, scientific numeric code is considered exclusively. Such codes typically have regular memory access patterns and do most of their computation within loop constructs. Symbolic codes, and other non-scientific-numeric programs, typically receive less attention, but cannot be ignored if the goal is general-purpose latency tolerance.

3 Modelling Prefetching

One question that needs to be asked is how well we can expect to do with prefetching in systems with a given latency. To understand how well a given architecture performs, we must have some sense of how well any architecture *could* perform given the application and memory subsystem in question. Mowry *et al.* give a good model of prefetching in their paper [22]; for a more detailed analysis of non-prefetching caches, see [1].

Let us use N to represent the total number of instructions, and W, R, and S to represent the number of write, read, and synchronization operations. For a non-prefetching cache, we can model the execution time as the sum of the instruction times, the write stall time due to full write buffers, the read stall time, and the synchronization time, where synchronization time can be taken to be zero for uniprocessors. The total run time of the program can thus be expressed as:

$$\text{Time} = N + Wl_{wb} + Rm_p(m_s l_{miss} + l_{fill}) + Sl_{sync}$$

Where

- l_{wb} is the average time per write spent stalling on a write due to a full write buffer;
- m_p and m_s are the fraction of reads missing in the primary cache, and the subsequent fraction missing in the secondary cache;
- l_{fill} and l_{miss} are the average latency for satisfying a primary cache miss hitting in the secondary cache, and the latency for a secondary cache miss, respectively; and
- l_{sync} is the average synchronization operation latency.

We can specify l_{miss} in more detail as a function of the number of memory units m, assuming each processor has one local memory and global references are evenly distributed across all processors' memories, as

$$l_{miss} = \frac{l_{local}}{c} + \frac{c-1}{c} l_{remote}$$

Prefetching decreases read latency, but adds an average overhead of o_{pf} per prefetch. Write latency and synchronization times may also change, but this can be regarded as a second-order effect. We define f_{pf} (prefetch coverage) as the fraction of primary read misses that are fetched into the primary cache, and l_{pf} as the average latency of a prefetched read (hopefully close to zero). The resulting model for the overall execution time then is:

$$\begin{aligned}\text{Time} = {} & N + Wl_{wb} + Rm_p f_{pf}(l_{pf} + o_{pf}) + \\ & Rm_p(1 - f_{pf})(m_s l_{miss} + l_{fill}) + Sl_{sync}\end{aligned}$$

Converting to CPI gives an equivalent formula (using w, r, and s to represent the percentage of each kind of instruction overall):

$$\begin{aligned}\text{CPI} = {} & 1 + wl_{wb} + rm_p f_{pf}(l_{pf} + o_{pf}) + \\ & rm_p(1 - f_{pf})(m_s l_{miss} + l_{fill}) + sl_{sync}\end{aligned} \quad (1)$$

Mowry and Gupta [22] give data for N, W, R, S, m_p, m_s and l_{sync} for their three benchmarks; appropriate data is provided in some other papers as well. (Data on the breakdown of cache hits into capacity, conflict, and compulsory misses can also be found in [12].) Assuming optimistically that write-buffer prefetching can be eliminated by read-exclusive prefetching, and that read latency can be reduced to a single cycle by prefetching, they suggest maximum speedups of 4 to 6 for some typical applications. In general, these assumptions are optimistic enough to exaggerate the performance gain by a factor of about $2\times$.

4 Software-Driven Prefetching: Minimal Hardware

For most code a compiler can extract information necessary to prefetch data and represent it explicitly in the code. This allows for intelligent prefetching based on knowledge of the code as a whole—for example, the ability to prefetch doubly-indexed arrays or linked lists—and the potential for dramatically less hardware to achieve similar or better performance.

4.1 Hand-Inserted Prefetches

Mowry and Gupta [22] study non-binding prefetching in a shared-memory multiprocessor. Their architectural model is based on the Stanford DASH multiprocessor, a shared-memory machine with memory distributed per-processor and with coherent caches maintained with a hardware directory protocol. The CPU consists of a MIPS R3000 processor with a 64K write-through primary data cache. The secondary cache is a 256K write-back cache 12 cycles away, with main memory an additional 10–68 cycles away depending on where in the machine it is located. Prefetching is performed with a prefetch instruction, which can specify either *read* or *read-exclusive*; the latter prefetches data in the exclusive-ownership mode necessary for writing. 16-deep write buffers are provided in the simulation. The benchmarks used are MP3D, a particle simulator; an LU decomposition program; and PTHOR, a parallel logic simulator. A 16-processor configuration is modelled throughout. To be able to simulate the benchmarks in a timely manner, they use much smaller datasets, and reduce cache size to match, down to 2K first-level and 4K second-level.

To get their results, they insert prefetching by hand, in an attempt to optimistically bound the performance of an equivalent compiler. They examine in detail each of the three benchmarks. They use *software pipelining* for loop prefetches, issuing prefetches for data one or two loop iterations before it is needed. In the LU benchmark, simple prefetching *increases* execution time by 6% due to hotspotting in the network when all the processors request the same pivot column. More sophisticated prefetching, with the prefetching of the cache lines in columns distributed evenly, is used in the improved version. They find that in applications that are dependent on linked lists, such as PTHOR, that a much smaller speedup can be gained.

They also assess the performance results of several different machine architectures. Increasing the cache size from 2K/4K up to 64K/256K does not qualitatively change the benefits of prefetching until the point where the working set is fully cacheable, as in LU with a large cache; at that point prefetching's advantage drops to only 5% or so. Sequential and release consistency memory models are compared and found to perform similarly under prefetching. A prefetch issue buffer separate from the write buffer is considered, and found to increase performance slightly. Prefetching directly into the primary cache is also considered (as opposed to fetching into a "cluster-local" memory controller as is done in DASH and simulated above), and found to improve performance significantly. They also tested suppressing the read-exclusive prefetch mode used elsewhere, and found that read-exclusive did, in fact, provide performance benefits by reducing write stall times and cache-coherence network traffic.

Using non-blocking (or *lockup-free*) caches is also considered. A non-blocking cache is one which can have multiple outstanding cache requests simultaneously; Kroft's implementation [17] uses *miss information/status holding registers* (MSHRs), which are used to record information relevant to outstanding cache requests. The MSHRs store information on the address referenced, the cache line that is being fetched, and where the returned datum should be sent to when it arrives in the cache. Lockup-free caches are found to dramatically reduce stall times, since at a minimum, reads no longer have to stall waiting for the write at the front of the write buffer to complete. In the base model, this stalling accounts for 10–20% of the overall execution time. Together with using a release consistency model, lockup-free caches eliminate almost all stall time due to writes, leaving only the issue of read prefetching.

Overall, when prefetching into a primary lockup-free cache, MP3D, LU, and PTHOR showed performance improvements of 2.5×, 2×, and 1.3×respectively. We can use Equation 1 from Section 3, copied below, to see how close they came to the theoretical maximum. We are provided with the values of w, r, s, m_p, m_s, and l_{sync}, as well as their actual values of f_{pf}. We assume l_{wb} is zero, since we have no data for it, likewise we will take l_{pf} to be zero, o_{pf} to be one, and f_{pf} to be one (initially). l_{fill} is 12 cycles, and l_{miss} is about 65 cycles. The optimistic assumptions yield an upper bound on the possible performance.

$$\begin{aligned} \text{CPI} &= 1 + wl_{wb} + rm_p f_{pf}(l_{pf} + o_{pf}) + \\ &\quad rm_p(1 - f_{pf})(m_s l_{miss} + l_{fill}) + sl_{sync} \\ &= 1 + rm_p f_{pf} + \\ &\quad rm_p(1 - f_{pf})(m_s l_{miss} + l_{fill}) + sl_{sync} \end{aligned}$$

Using the values on p. 93 of [22], we compute the upper bounds on speedup for each application in Table 1. (We do this by taking the ratio of the computed CPI value with $f_{pf} = 0$ and with $f_{pf} = 1$.) We also use the given prefetch coverage values to compute the predicted speedup for that coverage, and contrast it with the actual speedup to get a better idea of how optimistic our assumptions are.

Benchmark	Max Speedup	Actual f_{pf}	Predicted Speedup	Actual Speedup
MP3D	3.95	0.95	3.44	2.50
LU	4.56	0.93	4.26	2.00
PTHOR	3.80	0.56	1.70	1.30

Table 1: Predicted Performance for Software Prefetching

The maximum speedups, unsurprisingly, are inflated 2×–3× of the speedup seen. This is primarily due to the fact that the prefetch coverage (f_{pf}) and the average latency of prefetched data (l_{pf}) cannot be made to disappear in all applications. Some read latency remains due to insufficient time between address availability and data use or due to insufficient memory bandwidth. Note also that the data for the three benchmarks given does not include private data references, which adds some CPI to both the no-prefetch and prefetch models; modelling this would probably reduce speedup predictions by about 5%. Somewhat more significantly, while the assumption of one-cycle overhead may be appropriate for computing an upper bound, if we use a more realistic value of three cycles the predicted speedup decreases another 10%.

Modelling the remaining difference between predicted and actual speedup requires examining the f_{pf} and l_{pf} values. Since we don't have access to l_{pf}, we can simply apply a fudge factor to f_{pf} to get a value for the *successful prefetch coverage*,

f'_{pf}. This value combines f_{pf} with some non-zero l_{pf}, returning in effect a value of f_{pf} that represents $l_{pf} = 0$. Formally, $f'_{pf} = f_{pf}(1 - l_{pf}/(m_s l_{miss} + l_{fill})$; this is quite close to f_{pf} when l_{pf} is low, *i.e.* prefetching is successful, but lower when many prefetches are issued close to the instructions that read the prefetched data. If we factor out the above-mentioned 15% from local data and one-cycle prefetch overheads, we can use values of f'_{pf}/f_{pf} equal to 85%, 70%, and 60% respectively to remove the remaining difference between predicted and actual CPI. While these values are obviously ad-hoc, they do reflect the qualitative discussions in the text concerning which prefetches were expected to provide low-latency references. For example, many prefetches in PTHOR were added simply to increase coverage, not to provide effective latency hiding; the final round of substantive prefetching in PTHOR brought the coverage to only 27%, which compares well with our 34% f'_{pf} value used here.

The bottom line is that the actual speedup values above demonstrate the usefulness of non-binding software prefetching. A factor of 2 or 2.5 for numerical code is an impressive gain, with only a minimum amount of code modification (25–40 extra lines of code in each application). Furthermore, the amount of hardware required for this algorithm is minimal. Prefetches can be issued through the write buffer with only a minimal amount of extra hardware required. However, given the good results seen with using lockup-free caches, it is worth including in the hardware. Separate prefetch buffers, which increase performance slightly, can be included in the hardware if their addition does not involve much design time or layout area.

4.2 Compiler-Assisted Software Prefetching in Loops

Despite the favorable results from Mowry and Gupta's hand-placed prefetch instructions, we won't be able to confidently state that software prefetching is viable without compiler technology to back up the claim. In [23], Mowry *et al.* describe the compiler support implemented after [22] was sent to press. They use a uniprocessor MIPS R4000-like model with *pixie*, with 8K primary and 256K secondary caches, both direct-mapped with 32-byte lines. First-level miss penalty is 12 cycles, and a miss to memory takes 75 cycles, with requests at most one every 20 cycles. A 16-entry prefetch buffer is used. The cache is lockup-free with prefetches encoded as loads to R0. Their benchmark suite includes some SPEC, SPLASH, and NAS benchmarks; the compiler used is the Stanford University Intermediate Form (SUIF) compiler.

Their prefetch algorithm primarily examines loops in two phases, first determining intrinsic data reuse of a loop nest, then applying that information to a given cache. They define three kinds of reuse: *spatial reuse* across loops, like spatial locality; *temporal reuse,* like temporal locality; and *group reuse,* which encodes which references may actually refer to the same address. This information is encoded as a *reuse vector space*; for a given kind of reuse, encoded as a matrix, reuse takes place between two iteration vectors if their difference lies in the nullspace of the reuse matrix; *e.g.*, if

$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i_1 \\ j_1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i_2 \\ j_2 \end{bmatrix}$$

Then, we define the *localized iteration space* as the set of inner loops that can actually exploit reuse for a given cache. By representing this as a vector space as well, we can determine locality exists when the reuse vector space is a subspace of the localized vector space. From the reuse information, we can derive *prefetch predicates* for a loop with index i: for temporal locality, we only need to prefetch a given reference when $i = 0$; for spatial locality, we only need to issue a prefetch when $(i \bmod l) = 0$, where l is the cache line length. Group reuse is applied to determine the *leading reference,* which is the only one that needs to be prefetched. The compiler applies this information to the loops by splitting the loops such that for each loop, all the prefetch predicates have the same value. For $i = 0$, we *peel* the first iteration; for $(i \bmod l) = 0$ we *unroll* the loop by a factor of l. Code blowup is prevented by compiler heuristics that revert to conditionalizing or dropping prefetches when the code size gets too large. The prefetches are placed the right number of iterations in advance to compensate for memory latency.

They get good results with their compiler algorithm, comparable to the results achieved by hand in [22]. The speedup in their benchmark set ranges from 1.05× to 2×, with 6 of the 13 benchmarks improving by over 1.45×. 50% to 90% of the original memory cycles are eliminated. They show that the instruction count overhead of prefetching is usually less than 15%; some programs show overhead of 25–50%, but the overheads are always low enough to leave a good net improvement overall. They compare a non-selective prefetching algorithm and show that it has much higher overhead with very little improvement in memory stall time. Similarly, they compare their loop-splitting algorithm with run-time conditionals, and find that loop-splitting is almost always significantly better. Various parameters were tweaked (*e.g.,* the compiler's notion of the cache size and prefetch latency), and the algorithm was found to be robust. They also found that dropping prefetches when the prefetch buffer was full performed slightly worse than stalling for a free entry to become available. Finally, they experimented with having their compiler perform its full locality optimizing (loop blocking and the like), and found that the compiler's prefetching algorithm interacted well with the locality optimization, removing more memory stall cycles than either alone. Overall, the prefetching was extremely successful for their model.

In a separate paper from the same lab [11], the authors find that while prefetching interacts well with relaxed consistency models in their DASH-like shared-memory multiprocessor, it does not work consistently well with multiple contexts per processor. Both prefetching and multiple contexts are latency-hiding schemes; prefetching attempts to increase the hit rate, while multiple contexts create useful work when the misses occur. However, if either technique would suffice, you have to pay the overhead for both anyway. Furthermore, the two techniques can interfere, since prefetched data is more likely to be invalidated during the longer delay before use imposed by multiple contexts. They show that two contexts can work well with prefetching, but by four contexts the negative effects overwhelm the positive ones.

Prefetching a single loop ahead has also been examined by Callahan, Kennedy, and Porterfield in [3, 4, 25]. A somewhat more sophisticated analysis was carried out by Klaiber and Levy [16], who examined how to place prefetching instructions several loops ahead rather than a single loop. Chen and Mahle [7] examine prefetching in a multi-issue architecture; while the number of cycles between prefetch and use decrease, the otherwise idle issue slots provide ample room for adding prefetch instructions at no cost in cycle count. More esoteric strategies have also been discussed in the literature, but have reported no hard facts; one such is Johnson's

article [14] on precomputing and prefetching (or prebroadcasting) entire working sets in multiprocessors.

4.3 General Compiler-Assisted Software Prefetching

A recent thesis by Selvidge [27] proposes a compiler algorithm that has similar loop prefetching as Mowry *et al.*, including support for sparse arrays and loop indexes. However, his research goes beyond traditional scientific code, and includes a separate algorithm to handle prefetching in non-loop contexts. Results are presented for both 20-cycle latencies to main memory and 160-cycle latencies.

The first phase is identifying references for which prefetching will be productive. He finds that for the vast majority of programs, a given reference either mostly hits or mostly misses, and that while the low-miss-rate references (*goodrefs*) account for most of the dynamic references, the remaining high-miss-rate references (*badrefs*) account for most of the program misses. This partitioning has two advantages: firstly, prefetching can be applied just to the badrefs, avoiding the overhead of generating prefetches for references which will hit in cache anyway; and secondly, the goodrefs themselves can be used to provide parallelism to cover the memory references for the badrefs. Rather than using static techniques to partition the references, Selvidge uses a first-pass compilation and restricted execution environment to identify the two classes of references. This information is fed back to the second-phase compiler which performs the latency tolerance optimizations.

The loop-based prefetching technique used is called *RAAD prefetching*, for Regularly Accessed Aggregate Data structures. RAADs are characterized by the form of their strongly-connected components in the dependency graph of the program. Three distinct forms of RAADs are recognized: loop-constant stride array accesses (*e.g.*, `X[i++]`); accesses through loop-constant stride indirection arrays (*e.g.*, `Y[X[i++]]`); and linked-list dereferencing (*e.g.*, `x->a; x = x->next`). For array RAADs, prefetch instructions are inserted after the loop reference, targetting a loop the appropriate number of iterations ahead. Indirection RAADs are handled by prefetching the nested index two cycles in advance to compensate:

```
load a[index[i]]
prefetch a[index[i+1]]
prefetch index[i+2]
```

Linked-list RAAD prefetching is done using a sequence of the following type (note that this can only be applied a single iteration ahead, unlike the other techniques):

```
b = x->a
prefetch(newx->a)
...
x = x->next
newx = x->next
prefetch(newx->next)
```

Prologue code is generated as necessary for first-iteration prefetches; similarly, for indirection arrays and linked-list prefetching, the last loop is peeled to prevent loads to potentially illegal addresses. In conditional constructs, the prefetches are performed at the most infrequently occurring point in a loop that occurs whenever both an address update and a badref occur, to save on overhead.

When loop scheduling is not applicable, the compiler uses *miss scheduling*. This technique essentially tries to interleave badref references into the rest of the program with a frequency equal to the inverse of the memory bandwidth. The generation of addresses on the one hand, and the use of the referenced addresses on the other, prevent the interleaving from being optimal. Since the procedure operates on traces, however, address generation can be moved back across basic blocks (and duplicated if need be). Note that having identified the goodrefs earlier, it is possible to use them to hide the latencies of the badrefs.

The simulation framework was an extended SPARC instruction set, using code generated by a modified version of `gcc`. System performance data is reported for detailed simulations. Memory is modelled with two parameters, latency and bandwidth. A prefetch buffer of size one is used, and a 4K fully-associative cache is assumed. The workload used is the SPEC benchmark suite. The compiler chooses RAAD or miss scheduling as appropriate; miss scheduling in loops may be useful for loops with very large bodies, or in loops with many badrefs that execute a small number of times.

Selvidge's results were comparable to to those of Mowry and Gupta. For a system with 20-cycle latency and no bandwidth for overlapped requests (much more restrictive than Mowry and Gupta's memory model), speedups range from $1.1\times$ to about $1.3\times$, with numerical programs showing more speedup than symbolic code. Increasing latency to 160 cycles, with a request still issuable every 20 cycles, yields speedups of almost $2\times$ in the worst case, and up to $4\times$–$6\times$ for the more successful benchmarks. Overhead is seen to be from about 5% to 25%, with the highest overhead corresponding to the codes that also exhibited the highest speedups. Latency versus bandwidth limitations are characterized for the workload. Selvidge shows that both RAAD and miss scheduling are necessary to provide coverage for all the variety of misses in the SPEC suite, but that RAAD prefetching does a better job when applicable.

Overall, it is clear that compilers are capable of living up to the promise of Mowry and Gupta's initial paper. Using algorithms like those above, or in Section 4.2, makes it easy to take advantage of prefetching in environments with distant memories, and it can be done with only a minimal amount of hardware.

5 Throwing Hardware at the Problem: RPT and LA-PC

As an alternative to the software prefetching techniques outlined above, we can use purely run-time, hardware mechanisms to try to predict the data references that will be made in the near future based on references that have already been made. There are two clear advantages to purely hardware prefetching. The first is backwards compatibility: hardware prefetching has the potential to speed up unmodified old code. The second advantage is that the hardware can respond to patterned accesses that are difficult or impossible to predict at compile-time, such as data-dependent data accesses, or complex but regular recursive accesses. The corresponding disadvantage is that the hardware has no reliable way of looking ahead in the code flow to generate prefetches, unlike a compiler; and even when it can look ahead, on some problems it will be bounded by the size of hardware tables as to how much prefetching it can perform. Additionally, hardware prefetching is expensive in terms of design time and chip area.

5.1 Architectural Model

Chen and Baer [2, 6, 5] propose what is essentially a vector-stride-prefetching scheme, and spell out the details of how to identify vector strides in cache. They break down accesses for each instruction into four categories, as shown in Table 2; we will assume for the sake of example that the instruction is nested in loops indexed by i, j, and k.

Pattern	Example
scalar	simple variable
zero stride	$A[i][j]$ within k $S[i].$*off* within j
constant stride	$A[i]$ within i $A[i][j]$, $A[j][i]$ within j
irregular	$A[B[i]]$; $A[i,i]$; linked list

Table 2: Data Reference Types

Caches are clearly good for scalar and zero-stride references; constant-stride references, however, are prime targets for prefetching. The goal, then, is to prefetch for the constant-stride references without generating prefetches in any of the other three categories. To determine what kind of access a given address in the instruction stream is performing, a small finite state machine is associated with it, as shown in Figure 1. These FSMs are stored in a hardware *reference prediction table* (RPT).

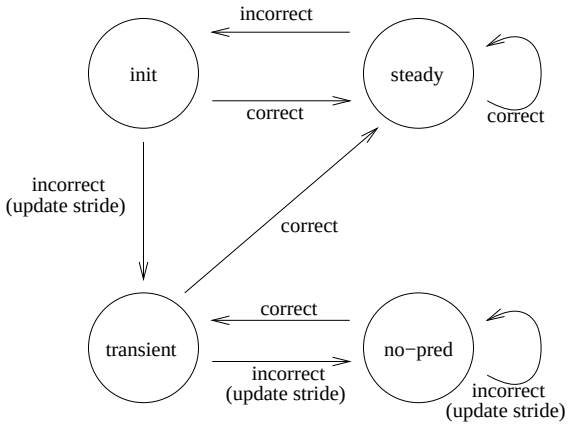

Figure 1: Reference Prediction FSM

In addition to the FSM state, the RPT stores information on the kind of stride that is believed to be occurring at the given instruction address: thus, the last address referenced, and the current stride, are both stored. The stride is determined simply as the difference between the last two references (specifically, the last two when a state transition from *init* to *transient* occurred, as below). Subtraction and addition hardware are included as part of the RPT. The different states of the FSM are as follows:

- The *init* state is set when an entry is first added to the RPT, or after a misprediction has occurred in the *steady* state—that is, when an ongoing vector access has suddenly stopped using the same stride.
- The *transient* state is for when the system isn't sure whether it is predicting correctly. A new stride is set by subtracting the previous address from the current reference address.
- The *steady* state indicates that the prediction should be steady for a while.
- The *no-pred* state is when the system is not making any predictions for the time being.

The four states allow for the prefetching hardware to ignore a single misprediction in the stride, and at the same time make it less likely to start issuing prefetches until a pattern of references with a given stride has clearly begun.

However, as described, this scheme only allows us to issue prefetches for, at most, a single loop ahead. If the loop is fairly short, or if the memory latency is quite long, this may not be enough to mask the loop latency; what we would like to do is issue prefetches well in advance of when they will be needed. To do this we use a *branch prediction table*, or BPT, such as described in [24]. The BPT keeps track of what direction a branch in the code is likely to go based on where it has jumped to in the past. Given that information, one can prefetch the instructions likely to be needed by using a *lookahead program counter* (LA-PC), which runs ahead of the real PC. The LA-PC is then used to try to prefetch the predicted next value of the reference directly; it runs forward until it finds an instruction stream address with a *transient* or *steady* RPT entry and a non-zero stride, and issues a preload for the next expected address. An *outstanding request list* (ORL) is used to suppress prefetches on cache lines that have already been prefetched.

The LA-PC is restricted to a fixed number of instructions ahead of the real PC to avoid generating excessive prefetches, and its accuracy is constrained by its ability to guess branches correctly. A branch-prediction table such as that in [24] is used to improve the accuracy of the LA-PC. In theory, we would like to have the *look ahead distance*, d, be about the same as the latency to the next level of the memory hierarchy, δ. Experimental verification shows that $d \approx 1.5\delta$ is a good value. Note that in the event of a branch table misprediction, we flush the buffered preload requests. Similarly, note that we need to check the ORL for ordinary reads and writes as well for prefetches.

5.2 Results

Chen and Baer use a trace-driven simulation, using *pixie* on a DEC-station 5000 to trace a set of benchmarks from SPEC (in [6]), and Perfect Club and a few others (in [2]). Cache warm start was simulated by ignoring the first 500,000 references. They use direct-mapped, 16-byte-line caches for all the various caches and tables. Various different memory models are used: *nonoverlapped* (10 cycles/line), *overlapped* (20 cycles/line, of which 14 are available to issue or transfer other requests), and *pipelined* (30 cycles/line, with a request issuable every cycle). They consider caches of size N as opposed to a same-size cache with a 256-entry RPT, and as opposed to an $N/2$ sized cache with a 256-entry RPT. N varies from 4 to 64. Results are presented in terms of cycles per instruction (CPI) for data access, *i.e.* total data access time divided by number of instructions executed. The silicon cost for a 256-entry RPT is given as equivalent to 2K of cache memory.

In [6], they compare CPI_{da} for the SPEC89 benchmarks. We converted the CPI_{da} values to plain CPI for both their baseline

model (using a write-buffer with read bypass, since most of the results used the bypass model), and for the same model with hardware prefetching enabled. The converted results and resulting speedups are in Table 3, using a pipelined 30-cycle latency memory model.

Benchmark	Baseline CPI	Prefetch CPI	Speedup
Matrix	2.20	1.04	2.12×
Tomcatv	1.66	1.03	1.61×
Spice	1.89	1.73	1.09×
Espresso	2.00	1.72	1.16×
Doduc	1.09	1.08	1.01×
Nasa	3.11	2.16	1.44×
Fpppp	1.05	1.05	1.00×
Gcc	1.12	1.12	1.00×
Xlisp	1.12	1.07	1.05×
Eqntott	1.24	1.20	1.03×

Table 3: Speedups with Hardware Prefetching

Using formula 1 for predicted CPI in Section 3 (repeated below), and the instruction ratios and memory models provided, we can attempt to derive the prefetching coverage (f_{pf}) attained by the hardware prefetching algorithm.

$$\begin{aligned} \text{CPI} &= 1 + wl_{wb} + rm_p f_{pf}(l_{pf} + o_{pf}) + \\ &\quad rm_p(1 - f_{pf})(m_s l_{miss} + l_{fill}) + sl_{sync} \end{aligned}$$

We set synchronization time to zero, since Chen and Baer are modelling a uniprocessor, and consider a bypass model only so that l_{wb} is also (effectively) zero. l_{fill} is 30 cycles in the memory model that we are using. Finally, since we have no cycles of prefetching overhead, $o_{pf} = 0$; and since we aren't modelling a secondary cache, either, we get

$$\text{CPI} = 1 + rm_p f_{pf} l_{pf} + rm_p(1 - f_{pf}) l_{fill}$$

For the optimal case, we have 100% prefetching coverage with no latency on prefetched data, which removes all data delays and gives us CPI = 1. More usefully, we can use m_p to derive a figure for the prefetching coverage and/or the prefetching latency. We take l_{pf} to be zero for simplicity, and then use the successful prefetch coverage, f'_{pf}, as defined in Section 4.1.

$$\text{CPI} = 1 + rm_p(1 - f'_{pf}) l_{fill} \quad (2)$$

$$f'_{pf} = 1 - \frac{\text{CPI} - 1}{rm_p l_{fill}} \quad (3)$$

Table 4 derives successful prefetch coverage values. Chen and Baer's paper provides us with the fraction of instructions that are reads, but not with m_p, the read miss ratio. Instead, we are given the combined miss ratio, which can be significantly different from the read miss ratio due to the influence of write misses. So we derive the 'nominal read miss ratio' by applying Equation 2 with $f'_{pf} = 0$ and using the baseline CPI values, to get $m_p = (\text{CPI} - 1)/(rl_{fill})$. With this nominal read miss ratio, we use Equation 3 to get a value for the prefetch coverage that the hardware algorithm was able to provide. The nominal read miss ratio will be somewhat high due to our assuming all the CPI_{da} is due to read misses. Since we have no reported change in CPI with prefetching for Fpppp and Gcc, I have not attempted to apply these equations to them.

Benchmark	Pref. CPI	Read Frac.	Comb. Miss %	Read Miss %	Pred. f'_{pf}
Matrix	1.04	0.307	8.7%	13.0%	95%
Tomcatv	1.03	0.326	6.3%	6.7%	95%
Spice	1.73	0.209	11.6%	14.2%	20%
Espresso	1.72	0.167	18.4%	20.0%	30%
Doduc	1.08	0.223	1.7%	1.3%	10%
Nasa	2.16	0.152	28.1%	46.3%	45%
Xlisp	1.07	0.315	1.4%	1.3%	40%
Eqntott	1.20	0.265	3.3%	3.0%	20%

Table 4: Predicted Prefetch Coverage with Hardware Prefetching

The scientific kernels (Matrix and Tomcatv) show a 95% coverage for prefetches, which matches the software coverages of 95% coverage for MP3D and 93% for LU by Mowry and Gupta. Their circuit simulator, PTHOR, had a 56% coverage, but they admit that the expected value of most of the prefetches is limited; the f'_{pf} value is probably closer to 20% or 30%, which corresponds to the 20–40% range for the non-numerical benchmarks in the table.

While the coverages are roughly comparable, the speedup results are not quite as good as Mowry and Gupta's; however, they are at least not far off. The scientific codes show a 1.6–2.1× speedup (as compared to 2–2.5× for Mowry and Gupta). Spice shows a 1.1× speedup, as compared to PTHOR's 1.3× with software prefetching. This is no doubt due to the fact that the hardware prefetch does not attempt to prefetch non-vector accesses, unlike the software technique used in PTHOR of placing a block prefetch before the elements of the block begin to be referenced. The hardware method's inability to do this accounts for the lower speedup it gets on applications with irregular memory access patterns. Further performance degradation may be due to the relatively high rate of mispredicted prefetches, up to 44% for one benchmark, and around 10–20% for all but the most regular numerical problems. Such a misprediction rate would be especially bad for throughput on a multiprocessor machine. Also note that we have used an optimistic base CPI of 1.0 throughout. If we use a base value of 1.5 to account for floating-point bandwidth restrictions, hazards, and so forth, the range of speedups for their scientific code drops to 1.4–1.75×, a 15–20% drop in overall performance improvement.

We would like to rule out memory latency as a significant factor in this comparison. Mowry and Gupta use a model with remote memory taking 60–80 cycles, depending on cache-line ownership; Chen and Baer use a 30-cycle latency through most of their examples. However, they do show one graph comparing 30-cycle and 100-cycle latencies; unfortunately, it does not provide enough information to compute actual speedups, since it gives only percentage change in CPI_{da} without giving the base value for 100-cycle latency. Nonetheless, we can surmise that the speedup is not much changed by noting that the percentage improvement for prefetch at the two latencies are always within about 5%. The exception is Espresso, which fails because the branch-prediction technique does poorly with its short blocks; prefetching at longer latencies saves only 15% of CPI_{da}, unlike the 30% savings we see with 30-cycle latency.

Chen's thesis [5] provides more timing details. In particular, it provides real speedups for four applications, Matmat, Mp3d, Water, and Cholesky, ranging from 1.1× to 1.66× in a multiprocessor with 80-cycle miss latency. Chen simulates a software model based

on hand-inserted prefetches with some by-hand loop-unrolling and rescheduling, finding roughly equal or better results for software, except for Cholesky, which has a loop that generates vectors with dynamic strides, causing the hardware technique to perform about 34% faster than software. With larger latencies (160 cycles), the software advantage improves (to up to 45% better than the hardware technique), while the hardware advantage's in Cholesky remains at 34%.

Interestingly, Chen also provides some initial results from combining the hardware prefetching mechanisms with cache prefetching via software. In the four benchmarks shown, the combination is more effective than either one alone; compared with pure software prefetching, in an 80-cycle latency environment, improvements of up to 1.5× can be had by adding hardware prefetching. While a more sophisticated software model would probably decrease this figure, there will always be room to increase performance via hardware on purely-dynamic, input-dependent access patterns if cost is no object.

Overall, it appears that the significantly more complex hardware scheme does not do significantly better than the simple software prefetch described in the previous section. If compiler technology such as that in [23] is available, and dusty-deck codes are not an issue, the RPT with LA-PC architecture appears not to be worth the extra cost for general-purpose machines. For machines with performance (rather than cost/performance) as the primary metric, the dynamic prefetching hardware may be worth the cost.

6 Adding Slightly More Hardware: Non-Blocking Loads

In the previous two sections we have examined software and hardware prefetching techniques. In this section, we consider a very slight hardware addition with a reasonable impact on performance: non-blocking loads. This feature allows several loads to be issued without waiting for them to complete one by one. Such a feature requires a non-blocking cache, but we have already seen that such a cache is beneficial for software prefetching. Additionally, we need to provide some form of support for non-blocking loads in the execution unit: either some form of interlocks for a statically-scheduled microprocessor, or scoreboarding if dynamic scheduling is in use. Such non-blocking loads can be used to prefetch data well before they are needed for a given computation.

In [6], Chen and Baer use this technique along with compiler reorganization (instruction scheduling and register renaming) to maximize how far in advance a word can be requested from memory. Note that this technique is a binding prefetch, since we request the data ahead of time but specify where its destination is to be. Non-blocking loads are supported by the use of a status bit attached to each register; this bit is cleared on a cache miss, and access to that register causes the processor to stall until the cache line is retrieved and the status bit is set. Non-blocking stores are also implemented by the simple mechanism of having an 8-entry write buffer; reads depending on the outstanding writes correctly return the value waiting to be written, and amount to a cache hit. For details on the memory hierarchy and metrics used by Chen and Baer, refer to the previous section.

The *non-blocking distance* that can be achieved between a non-blocking load and its first reference is fairly small. The load cannot be lifted above a conditional, for example. Chen and Baer use an algorithm intended to create as much distance between a load and its first use, while at the same time trying to even out the memory bandwidth use as the program runs. They attempt to rename registers to remove false (output- and anti-) dependences in the code first. One disadvantage of this approach is that the need for extra registers creates more *register pressure*, requiring more register spills and reloads during the execution of the code. Code with data loading phases followed by data use benefits from this reorganization (*e.g.*, non-blocking distance increases from 2–3 cycles to 8–10). Other code has only moderate improvements. However, it is important to note that these algorithms work not just on loop-structured code but on general-purpose non-numeric code as well.

Chen and Baer show 'normalized data penalty' figures for the improvement that their scheduled routing makes on the non-blocking cache CPI_{da}, but since this time we have the baseline values, we can backtrack to derive the actual code speedups seen from the baseline model (with only a standard cache) to their non-blocking load model, with instruction scheduling and register renaming; see Table 5.

Benchmark	Baseline CPI	Non-Block CPI	Speedup
Matrix	2.21	1.35	1.63×
Tomcatv	1.78	1.26	1.41×
Spice	1.90	1.74	1.10×
Espresso	2.01	1.94	1.03×
Doduc	1.14	1.04	1.10×
Nasa	3.20	1.67	1.92×

Table 5: Speedups with Non-Blocking Loads

As can be seen, we can get considerable speedup using this technology; in particular, both Nasa and Doduc did better with rescheduled, non-blocking loads than with the hardware prefetching techniques of the last section. (Though note again that increasing the base CPI to 1.5, as in Section 5.2, would decrease the overall speedups from 1.4–1.9× to 1.3–1.7×.)

In fact, Chen and Baer show that it is possible to combine prefetching with non-blocking loads to get significantly increased performance. Essentially, they make use of prefetching to achieve non-binding pre-miss overlap with computation, and then make use of non-blocking loads to get a binding, post-miss overlap with computation. There are several different scenarios for which non-blocking loads are appropriate even with software prefetching available:

- when data addresses are not available much before the data is needed, a non-blocking load will mask the latency with lower overhead (only one instruction is needed to prefetch, rather than at least two);
- even if the addresses are predictable, if register pressure allows, it takes less overhead to use non-blocking loads; and
- using non-blocking loads as early as practical in the instruction stream will reduce latency in the fact of consistency or conflict cache misses, either by reading from the cache before the cache line is evicted, or by initiating a cache fill earlier than a blocking load would.

Chen and Baer assess two of their benchmarks, Tomcatv and Nasa, using both prefetch and non-blocking loads; for the former,

prefetch does much better than non-blocking loads, while for the latter, the reverse is true. For Tomcatv, they observe a slight improvement, but since prefetching already has the CPI_{da} down to just a few hundredths of a cycle, there isn't much room for improvement. For Nasa, on the other hand, prefetching performs less well, reducing CPI from 3.11 to about 2.16, while non-blocking loads reduce it to 1.72. Combining the two, however, reduces it down to about 1.1, which is an overall performance improvement of almost three-fold.

Further data on combining binding and non-binding prefetch is available in [27]. Selvidge models the effect of his prefetching algorithm on both *stall* and *interlock* memory models; in the stall model, reads and writes block, and in the interlock model, full/empty or scoreboard information is used to allow non-blocking loads. In his 160-cycle latency model, up to a 1.16x speedup is achieved by using the interlock model rather than the stall model—and his algorithm does no optimizing to make use of the non-blocking loads, so the measured improvement is entirely serendipitous. With compiler techniques to lift non-blocking loads, better performance should be achievable by preventing otherwise unmasked misses from adding latency when the data is used.

Non-blocking loads have the potential to contribute to existing prefetching algorithms, whether software or hardware. Based on the data from Chen and Baer and from Selvidge, simple non-blocking load support appears worth the cost of a simple interlock or scoreboarding scheme.

7 Adding Yet More Hardware: Speculative Loads

A way of increasing the possible non-blocking load distance is outlined by Rogers and Li in [26]. They explicitly use *speculative load* instructions instead of loads when they are prefetching. Speculative loads function just like the binding, non-blocking loads of [6], but they do not cause a trap or page fault if they cannot be satisfied; instead they just set a special *poison bit* in the register. This lets a code block such as this be rewritten:

```
if (condition) {
  ld Rx, A;
  ...
} else {
  ld Rx, B;
}
```

If we know that `condition` is usually true, it can be safely rewritten as the following, even if the address A is illegal when the condition is false:

```
sld Rx, A;
if (condition) {
  ...
} else {
  ld Rx, B;
}
```

The register file is augmented with both a *presence* bit, like that discussed in Section 6, and a *poison* bit, as mentioned above. The bits control reads and writes of the register as outlined in Table 6; note that both bits are never set at the same time. A read when the poison bit is set can simply reissue the load; that way a page fault will be handled appropriately, and an illegal reference will cause a trap. A write to a register with the poison bit set simply overwrites the poison bit and the register contents. A write to a pending register stalls so as not to overwrite the written value with the read value when it is finally retrieved.

State	Read Action	Write Action
pending	stall until present	stall until present
present	proceed normally	proceed normally
poison	force load	proceed normally
both bits	N/A	N/A

Table 6: Speculative Read/Write Actions

In the architecture presented by Rogers and Li, speculative loads bypass the cache; if the value desired is already in the cache, however, the speculative load is treated like an ordinary load. The pending loads are stored in special queues, one per memory bank, and issued only when no other traffic (*e.g.*, load misses and write-through traffic) is on the bus. Returned speculative load traffic is queued and written into the register file during unused write-back stage cycles.

The simulation model used was a MIPS R2000 simulator with a baseline 16K direct-mapped write-back cache with 32-byte lines and no second-level cache. When comparing with the speculative model, the cache size and line size are reduced by a factor of two. The memory model is quite sophisticated. Memory latency for a cache line is 14 cycles for the 32-byte lines, and 10 cycles for the 16-byte lines; direct speculative loads take only 7 cycles since they read just one word, bypassing the cache. The write buffer is depth 6, the speculative load queues are depth 12, and the return queues are depth 16. The workload is a set of Livermore Loop kernels compiled by the MIPS C compiler and then hand-converted to use speculative loads. They use conversions similar to those in Section 6; these are augmented by lifting above conditional branches and lifting across loop iterations, both of which are safe using speculative loads. They discuss techniques for lifting across multiple loop iterations, using multiple register sets for each loop in the pipeline.

Rogers and Li present their results in a particularly challenging format. Their metric is average memory reference cost in processor cycles (t_{mr}), with one as the ideal case. However, they do *not* provide any statistics on memory-referencing instruction frequency, which makes it difficult to compute real speedup from their metric. Conveniently, though, they are using the MIPS C 2.10 compiler to get their results, which is a compiler we have available here, and they provide code fragments for each benchmark they use. Accordingly, we compiled their code to assembly to determine the percentage of memory references in each code fragment's inner loop, and used that value to bias the t_{mr} values given.

Table 7 presents the t_{mr} values for their baseline and speculative load value, as well as our measured memory referencing instruction percentage and the resulting speedup between their baseline and speculative load results. The percentage of memory operations is an upper bound, as it represents only the inner loop of each kernel. We show the range from 4-way to 16-way interleaving values for the speculative t_{mr} values; 16-way corresponds more closely to the 'pipelined' memory architecture of Chen and Baer.

These results, which are heavily memory-intensive, are best compared to the regular scientific codes of Chen and Baer (which they dynamically measured and found to be in the 40% to 55% range for the top four memory-referencing codes). Chen and Baer's

Kernel	Base t_{mr}	Spec t_{mr}	Mem %	Speedup
1	3.2	1.06–1.20	50%	1.6×–1.7×
2	2.7	2.45–2.50	60%	1.04×–1.06×
6	1.9	1.04–1.45	57%	1.15×–1.30×
9	3.4	1.60–1.60	52%	1.5×–1.5×
23	6.6	2.00–2.70	54%	1.9×–2.2×

Table 7: Speedup for Speculative Loads

speedups for those codes ranged from 1.4× to over 1.9× with roughly similar latencies; these values appear quite similar to the ones derived from the speculative load data given above.

Unfortunately, there is no attempt made by Rogers and Li to separate out the components of their speedup due to:

(a) the *speculative* nature of the loads (no validity checks necessary before issuing a potentially faulting load); and

(b) the *non-caching* nature of the loads (lower latency loads but without depositing the value in the cache).

Personal communication with Rogers confirms that they did not perform experiments to separate these two factors. Their research group has, however, studied item (b) and concluded that speculatively loading into the cache does in fact do better in all the studied benchmarks, though not dramatically. This suggests that any improvements which their speculative load architecture has over a more 'traditional' non-blocking load architecture lie primarily with (a), the purely speculative nature of the loads. However, the examples that Rogers and Li provide could all be trivially recast to use non-blocking loads instead of speculative loads, since there are no conditionals in their benchmarks; the non-blocking loads in the final iteration can be peeled or the arrays can be made slightly larger to allow references to the items just past their ends. The only potential slowdown would then be page-faults from the final references, which non-blocking loads will force while speculative loads will drop. While the group is currently working on a compiler to allow them to compile more substantive benchmarks, they have thus far provided little evidence that speculative loads can perform significantly better than non-blocking loads.

It appears, then, that while speculative loads and poison bits may be an elegant solution to the problem of increasing non-blocking distance, we do not currently know if they will result in any significant improvement. I suspect we will see some performance improvement in subsequent results from this group, but perhaps not enough to justify the extra mechanism needed: the poison bit on each register, extra trap handling, as well as (possibly multiple) speculative load request queues and the return queue.

8 Conclusion

As memory latencies increase, it becomes more important to deal with latency easily and effectively. There are many strategies that deal with memory latency—blocking and other compiler transformations, memory consistency models, and multi-threading, for example—but prefetching performs consistently well, on its own or in concert with other strategies. Various mixes of hardware and software support have been proposed, ranging from minimal hardware and sophisticated software to complex hardware with no software support.

The elaborate hardware solutions seem useful primarily for two markets: markets whose criteria is performance only, with less regard for costs, or markets concerned exclusively with 'dusty-deck' codes. The former market is generally a small one, and the latter can be expected to shrink with the steady migration of users to source-level portable systems, such as POSIX and Windows/NT, which can easily support software-oriented prefetching schemes.

Software prefetching provides good latency coverage. Compiler-based schemes such as [23] or [27] provide the best results seen so far, with, *e.g.*, speedups of about 5× in systems with 120-cycle latencies. The hardware requirements are minimal: support for a prefetch instruction, possibly a prefetch-issue buffer, and a lockup-free cache.

Finally, additional speedups can be gained by using non-blocking loads to minimize prefetch overhead and guarantee data availability, subject to the constraints of register pressure; further work needs to be done to assess how much benefit can be gained. The speculative load mechanism, which is of moderate complexity, can be added as well should further research justify its cost.

9 Acknowledgements

Thanks to Kirk Johnson for his comments on the February draft, and to the Reading Room staff (particularly Maria Sensale) for their help on tracking down a few of the more obscure references.

References

[1] Anant Agarwal, Mark Horowitz, and John Hennessy. An analytical cache model. *ACM Theory of Computing Systems*, 7(2):184–215, May 1989.

[2] Jean-Loup Baer and Tien-Fu Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of Supercomputing '91*, pages 176–186, November 1991. Also available as U. Washington CS TR 91-03-07.

[3] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, April 1991.

[4] David Callahan and Allan Porterfield. Data cache performance of supercomputer applications. In *Proceedings of Supercomputing '90*, pages 564–572, November 1990.

[5] Tien-Fu Chen. *Data Prefetching for High-Performance Processors*. PhD thesis, University of Washington at Seattle, July 1993. Also available as TR 93-07-01.

[6] Tien-Fu Chen and Jean-Loup Baer. Reducing memory latency via non-blocking and prefetching caches. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 51–61, October 1992. Also available as U. Washington CS TR 92-06-03.

[7] William Y. Chen, Scott A. Mahlke, Pohua P. Chang, and Wen-mei W. Hwu. Data access microarchitectures for superscalar processors with compiler-assisted data prefetching. In *The 24th Annual International Symposium on Microarchitecture*, pages 69–73, November 1991.

[8] G. C. Driscoll, J. J. Losq, T. R. Puzak, G. S. Rao, H. E. Sachar, and R. D. Villani. Cache miss directory—a means of prefetching cache 'missed' lines. *IBM Technical Disclosure Bulletin*, 25(3A):1286, August 1982.

[9] John W. C. Fu and Janak H. Patel. Data prefetching in multiprocessor vector cache memories. In *The 18th Annual International Symposium on Computer Architecture*, pages 54–63, May 1991.

[10] J. D. Gindele. Buffer block prefetching method. *IBM Technical Disclosure Bulletin*, 20(2):696–697, July 1977.

[11] Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry, and Wolf-Dietrich Weber. Comparative evaluation of latency reducing and tolerating techniques. In *The 18th Annual International Symposium on Computer Architecture*, pages 254–263, May 1991.

[12] Mark D. Hill and Alan Jay Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, December 1989.

[13] IBM. *IBM Journal of Research and Development, Special Issue on RISC System/6000*. IBM, January 1990.

[14] Eric E. Johnson. Working set prefetching for cache memories. *ACM Computer Architecture News*, 6(17):137–141, December 1989.

[15] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *The 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[16] Alexander C. Klaiber and Henry M. Levy. An architecture for software-controlled data prefetching. In *The 18th Annual International Symposium on Computer Architecture*, pages 43–53, May 1991.

[17] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *The 8th Annual International Symposium on Computer Architecture*, pages 81–87, May 1981.

[18] Roland L. Lee, Pen-Chung Yew, and Duncan H. Lawrie. Data prefetching in shared memory multiprocessors. In *International Conference on Parallel Processing*, pages 28–31, August 1987. Also as Illinois CSRD TR 639, January 1987.

[19] Roland L. Lee, Pen-Chung Yew, and Duncan H. Lawrie. Multiprocessor cache design considerations. In *The 14th Annual International Symposium on Computer Architecture*, pages 253–262, June 1987.

[20] Roland Lun Lee. *The Effectiveness of Caches and Data Prefetch Buffers in Large-Scale Shared Memory Multiprocessors*. PhD thesis, University of Illinois at Urbana-Champaign, May 1987. Also available as Illinois CSRD 670.

[21] Motorola. *MCS88100: RISC Microprocessor User's Manual*. Motorola Inc., 1988.

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

[23] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, October 1992.

[24] Shien-Tai Pan, Kimming So, and Joseph T. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In *The Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 76–84, October 1992.

[25] A. K. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Rice University, May 1989. Also available as Rice COMP TR 89-93.

[26] Anne Rogers and Kai Li. Software support for speculative loads. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 38–50, October 1992.

[27] Charles Selvidge. *Compilation-Based Prefetching for Memory Latency Tolerance*. PhD thesis, MIT, May 1992.

[28] Dick Sites and Rich Witek. Alpha architecture technical summary. Available for anonymous FTP from `gatekeeper.dec.com` as `pub/DEC/Alpha/technical-summary.txt`, 1992.

[29] Alan Jay Smith. Sequential program prefetching in memory hierarchies. *IEEE Computer*, 11(12):7–21, December 1978.

[30] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[31] Alan Jay Smith. Cache evaluation and the impact of workload choice. In *12th Annual International Symposium on Computer Architecture*, pages 64–73, June 1985.