


See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/216801810>

Software Prefetching

Conference Paper in ACM SIGARCH Computer Architecture News · April 1991
DOI: 10.1145/106972.106979

CITATIONS
351

3 authors, including:




David Callahan
University of Washington Seattle

60 PUBLICATIONS 3,706 CITATIONS

SEE PROFILE

READS
321



Allan K Porterfield
University of North Carolina at Chapel Hill

41 PUBLICATIONS 1,643 CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:

- Project

QUARC : An Array Programming Approach to HPC [View project](#)
- Project

Qthreads [View project](#)

Software Prefetching *

David Callahan[†]
Ken Kennedy[‡]
Allan Porterfield[†]

Abstract

We present an approach, called *software prefetching*, to reducing cache miss latencies. By providing a nonblocking *prefetch* instruction that causes data at a specified memory address to be brought into cache, the compiler can overlap the memory latency with other computation. Our simulations show that, even when generated by a very simple compiler algorithm, prefetch instructions can eliminate nearly all cache misses, while causing only modest increases in data traffic between memory and cache.

1 Introduction

As commodity microprocessors become faster and memory requirements continue to grow unabated, the latency for accesses to global memory will increase. Already, we are seeing latencies as large as 25 cycles for fast microprocessors using the Intel i860, and some supercomputers exhibit latencies in the range of 50 to 100 cycles. To ameliorate the effect of long latencies on machine performance, most systems offer some form of cache memory. Cache memory can be accessed very rapidly (one to three cycles on most machines) but has limited storage capacity.

Our research has indicated that enormous benefits can be achieved by rearranging computations to maximize the potential for reuse of data in cache [Por89, CKC90, CK89]. However, these approaches

cannot avoid the cost of loading a datum for the first time. In essence, any cache-based machine must pay the overhead of loading each data item, or cache line, at least once. Furthermore, it may have to pay the cost many times if accesses to cache lines are noncontiguous and unpredictable at compile time.

In this paper, we present an alternative approach, called *software prefetching*, to reducing cache miss latencies. The idea is to provide a nonblocking *prefetch* instruction that causes data at a specified memory address to be brought into cache. Our simulations show that, even when generated by a very simple compiler algorithm, prefetch instructions can eliminate nearly all cache misses, while causing only modest increases in data traffic between memory and cache.

The remainder of this paper is divided into 5 sections. The next section gives background on PFC-Sim, the simulation tool used in this research, and on the RiCEPS benchmark suite test programs, used as input for the studies. In Section 3, we discuss the properties of traditional methods for reducing the effects of latency—long cache lines and hardware prefetching. Section 4 introduces software prefetching and shows that it outperforms hardware prefetching in both hit percentage and data traffic. Finally, Section 5 discusses the costs of software prefetching and suggests ways that they might be overcome. Conclusions are drawn in Section 6.

2 Background

The research described in this paper would not have been possible without two important assets—the PFC-Sim simulation system, which makes it possible to study the effect on an arbitrary cache structure of executing any program, and RiCEPS, a suite of scientific benchmark programs collected expressly for use in compiler research.

*This research was supported in part by a grant from the IBM corporation while the authors were at Rice University.

[†]Tera Computer Company, 400 North 34th Street, Suite 300, Seattle WA 98103

[‡]Rice University, Department of Computer Science, P.O. Box 1892 Houston TX 77251-1892

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1991 ACM 0-89791-380-9/91/0003-0040...\$1.50

```

DO I = 1, N
  DO J = 1, N
    CALL STORE(A(I,J),4,TIME,1)
    A(I,J) = 0
    DO K = 1, N
      CALL LOAD(A(I,J),4,TIME,2)
      CALL LOAD(B(I,K),4,TIME,3)
      CALL LOAD(C(K,J),4,TIME,4)
      CALL STORE(A(I,J),4,TIME,5)
      A(I,J) = A(I,J) + B(I,K)*C(K,J)
    ENDDO
  ENDDO
ENDDO

```

Figure 1: Matrix Multiply - After the PFC-Sim Pre-processor

2.1 PFC-Sim

PFC-Sim is a program-driven event-tracing facility made up of three parts; a preprocessor, a run time simulator, and data collection/visualization routines [Por89, CKP90]. The preprocessor reads the original program and produces an annotated version of the program, complete with calls to the simulator. The annotated program is then compiled and executed. During execution, the simulator records important information, occasionally generating a trace record.

PFC-Sim was designed to eliminate the need for long trace files. A program with a one hour execution time can easily access a cache billions of times; even with very small trace entries this would exceed the available disk storage. PFC-Sim is able to write records less often by executing the simulator in real time—as the program is executing. Rather than record information in a trace file, PFC-Sim executes a simulation routine at each memory reference. The routine is passed the information normally in the trace entry, plus information about the original source location. Figure 1 shows an annotated version of matrix multiplication, in which only references to subscripted variables are being traced. The procedure parameters specify whether the value is being loaded or stored, the address being accessed, the length in bytes of the access, the simulated time (which is also incremented at basic block boundaries in code not shown), and a unique id for the call.

We eliminate the need to save any trace entries by pipelining the creation of the trace entry directly into the simulator. Since the detection of events is performed in the source program, relationships between the actions of the simulator and the source are

easy to maintain.

Note that, since calls to simulation routines are made at specific points in the source, it is possible to summarize information about particular references in a program. For example, it is easy to determine the percentage of cache hits for any given memory reference in a program being simulated. In Figure 1, this might reveal that, for small caches, the reference to C(K,J) is almost always a miss. To capitalize on this observation, we built a visualizing browser that can display a program using different colors for references with different hit percentages [CKP90].

In the research described here, PFC-Sim recorded information only about subscripted variables. Although this simplification might effect the accuracy of the results, the effects will not be significant because good global register allocation, as found in most product compilers, allocates most of the scalar variables to registers between accesses. When we turned on tracing for scalar variables, we discovered that the number of accesses increased noticeably but the number of misses was almost unchanged. The main motivation for ignoring scalar references is a practical one: when we do so, the simulated program runs three times as fast, making it possible to run simulations for reasonably large programs.

PFC-Sim can simulate a wide variety of caches. The efficiency of the simulator depends on the parameters picked. A standard simulation increases the execution time of the simulated program by a factor of 10 to 20. This factor is larger for a fully-associative cache or one that uses an “optimal” replacement policy (available as a standard for comparison). Although the increases are significant, they do not preclude the use of PFC-Sim on long, large programs. Programs with original execution times up to several hours have been simulated without problem.

2.2 Rice Compiler Evaluation Program Suite

PFC-Sim was used to examine the cache performance of programs in a new supercomputer benchmark set, the Rice Compiler Evaluation Program Suite (RiCEPS)¹ [Por89, CP90], which is designed as a benchmark for high performance compilers. Although there is some overlap, It differs from the well-known Perfect Club benchmark suite [BCK⁺89], which is aimed at performance evaluation of computer systems, rather than compilers. The programs in RiCEPS represent a broad spectrum of computational and coding styles. Each program was selected

¹RiCEPS is currently available (Dec. 90) though anonymous FTP from titan.rice.edu

to be representative of a group of applications. The programs are substantial in size and require a significant amount of execution time. Each member of RiCEPS includes, in addition to the Fortran source, a brief description of what the code does (what problem it solves and what algorithms it uses) and at least one set of data on which to run the program.

Although RiCEPS will eventually grow to include over twenty programs, the experiments reported here are based on a preliminary version containing twelve programs. The programs' execution times range from one minute (MATRIX) to several hours (SIMPLE, BARO, BOAST) on an IBM 3081D. While most programs are 1 to 3 thousand lines, they range in size from 15 lines to over 23,000 lines. All of the programs were compiled with the IBM VS2 FORTRAN compiler and to run on an IBM 3081D.

The benchmark programs used for this work included:

- MCMB** - microbial biodegradation
- MATRIX** - 100 x 100 matrix multiply
- BARO** - weather simulation
- SIMPLE** - hydrodynamics benchmark from Livermore National Laboratory
- EFIE304** - calculation of the current distribution on an arbitrary body
- BOAST** - black oil reservoir simulator
- EULER1** - solver for one dimensional unsteady Euler equations
- SHEAR** - three dimensional turbulent fluid dynamics simulation
- MHD2D** - solver for 2D MHD equations with periodic boundary conditions
- ONEDIM** - one dimensional Schroedinger equation solver
- LINPACKD** - the standard LINPACK benchmark [Don88]
- WANAL1** - boundary control of wave equations program

3 Performance of Traditional Cache Prefetching Schemes

As a preliminary study, we evaluated the effectiveness of hardware prefetching schemes on computationally intensive programs. In addition to determining how the hit ratio for the program is affected by the cache structure, the total data traffic between the cache and the main memory was computed. Every reference in the programs had its individual hit ratio dynamically measured. Hundreds of possible cache configurations exist, and testing all possible configurations is impractical. To test the effect of a

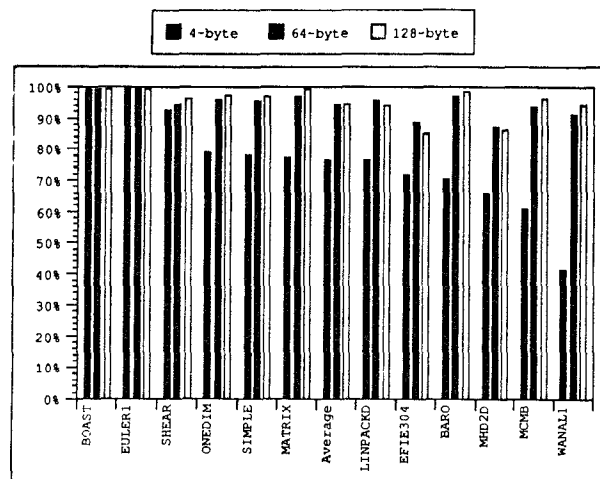


Figure 2: Effects of Cache Line Lengths on Hit Ratios

particular cache parameter, a base cache was defined and a single parameter was varied on each simulation. The selected base cache was a 32K LRU, 4-way set-associative cache, with a one-word cache line, using a write-back store policy and performing no prefetching.

Rather than present all of the cache performance results here (more complete results can be found elsewhere [Por89, CP90]), we will examine the effective of two different hardware mechanisms that fetch data into a cache before they are requested: long cache lines and hardware prefetching.

Long cache lines are the most common means of prefetching. In this scheme, a cache miss results in retrieval of every datum in a block of standard size, at least large enough to contain two data items. Note that a cache miss must be taken before a line can be brought to cache.

An alternative scheme, called hardware prefetching, avoids this miss penalty in many cases by fetching the next data item sequentially in memory when any data item is fetched from cache. In effect, if item A(I) is fetched, the cache will also issue a fetch of A(I+1). The method implemented is very close to the *tagged prefetch* described by Jouppi [Jou90].

3.1 Long Cache Lines

To determine the degree of locality among the references and the effects of long cache lines on memory performance, we examined both the hit ratio and the data traffic observed for the various line lengths on the programs in RiCEPS.

Three cache line lengths (4, 64, and 128 bytes) were simulated. In Figure 2, we see that the longer cache lines substantially reduced the misses that oc-

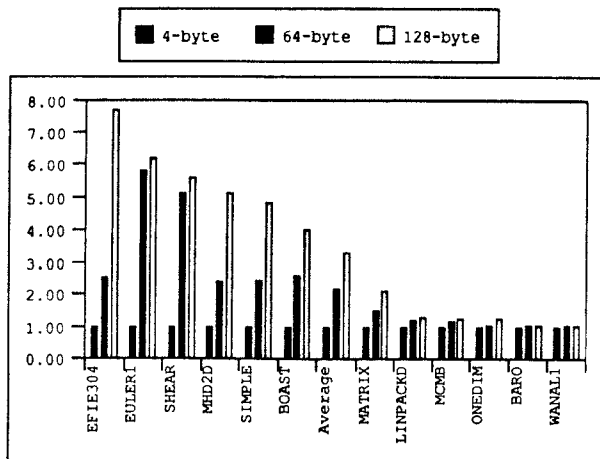


Figure 3: Effects of Cache Line Lengths on Data Traffic

cur during execution. Several programs, including BARO, showed almost perfect prefetching, (i.e., the number of misses for 4 byte cache lines was 16 times the number of misses for 64 byte lines). Increasing the line length to 128 bytes produced mixed results. Several programs (e.g., BARO) had the number of misses reduced by almost 50%. Others, (e.g., MATRIX), showed little difference in the number of misses between 64 and 128 byte lines. A third group, including EFIE304 and MHD2D, had fewer misses for 64 than 128 byte lines.

The programs fall into two distinct groups upon examining the total amount of transferred data (Figure 3). For programs where long-line prefetching was spectacularly successful, the increase in traffic was minimal (less than 10% for WANAL1 and BARO). The more common case was for traffic to increase from 300 to 600% for 64 byte lines and 500 to 800% for the 128 byte lines.

Overall, prefetching with long cache lines was successful for the computationally intensive programs in the benchmark. The cost of long cache lines showed up in the increased bandwidth required to support resulting data traffic. Many architectures hide this cost by building wide busses and interleaving main memory so that the wider cache lines can be moved a single transfer. This allows a higher bandwidth to be supported for wide cache lines than for single word lines.

3.2 Hardware Prefetching

A second mechanism for prefetching in the hardware is explicit prefetching. When a cache entry is accessed the cache can assume that the address of

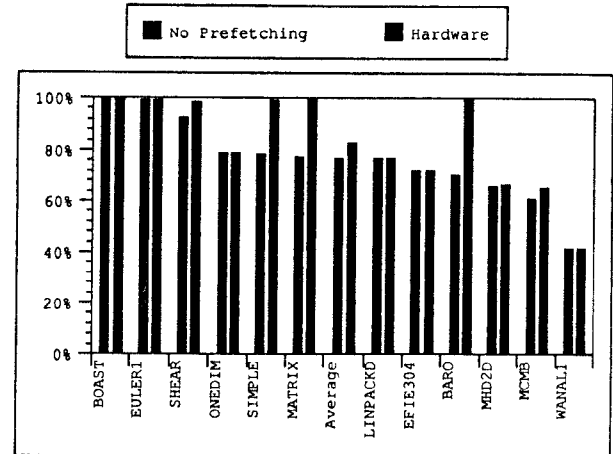


Figure 4: Effects of Hardware Prefetching on Hit Ratios

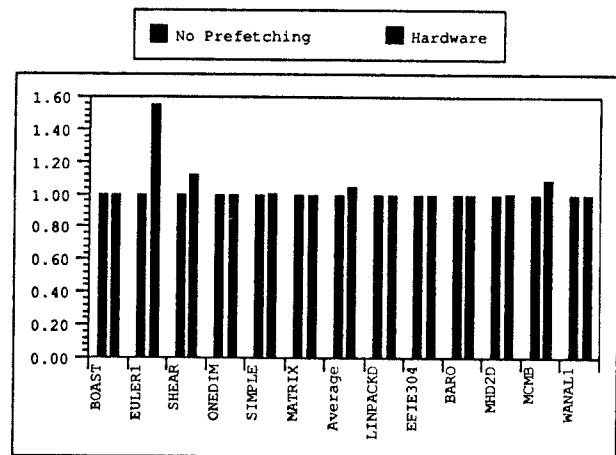


Figure 5: Effects of Hardware Prefetching on Data Traffic

the next datum in the address space will be accessed in the near future. When the last entry in a cache line is accessed the hardware automatically retrieves the next cache line and will have it in the cache when (if) it is accessed. (Tagged prefetch would have performed the prefetch when the first element is used [Jou90].) This method has the advantage of allowing sequential array accesses to all be fetched with only one miss (rather than a miss every 8, 16, or 32 entries). When long cache lines are used, the hardware prefetching can be initiated only when the entries in a cache line are accessed sequentially. (i.e. if the cache line holds four elements: access A, A+1, A+2, A+3 and prefetch line starting at A+4).

When hardware prefetching was enabled, the programs separated into two approximately equal sized

groups. Figure 4 shows one group of programs where misses were almost completely eliminated. Two of the programs in this group already had very few misses, but three programs (SIMPLE, MATRIX, and BARO) showed significant improvements in hit ratios. BARO appears to have been modified to allow easy vectorization. The accesses to memory are arranged to allow all vectors to have strides of one. With vector strides of one, the prefetching accurately predicted the next required value. The second group showed almost no improvement using hardware prefetching.

The overhead in data traffic for hardware prefetching was uniformly very low. Only three programs (Figure 5) showed more than 1% increase in data traffic. MCMB increased by 3%, SHEAR by 10% and EULER1 by 50%. EULER1's increase in traffic occurs because hardware prefetching actually produced slightly more cache misses than the non-prefetching version. Whenever a prefetch occurs that is not used, it pushes some value out of the cache. Occasionally hardware prefetching pushes out a value that EULER1 would have otherwise reused. The small number of extra loads from this effect are magnified because EULER1 does not otherwise push items out prematurely. The very small number of misses and data traffic causes a small absolute increase over time to appear very large on this graph.

Overall, hardware prefetching is a win for programs with column-wise accesses and produces very little overhead for any program.

4 Software Prefetching

Not wanting to incur the bandwidth requirements of long cache lines, but desiring very high hit ratios, we looked for ways to improve the hardware prefetching technique. When examining, by hand, the programs for which memory performance was not improved by hardware prefetching, we discovered all had patterns of array access that could be predicted during execution by simple code generated by the compiler. Every program showed some type of predictable access pattern in the inner loops. Since most of these patterns could be determined at compile time, a mechanism that allows a compiler to manage prefetching is likely to be effective. The rest of this paper studies the cost and effectiveness of a very simple addition to the hardware, a cache load instruction, to allow software management of prefetching. Using this instruction, a straightforward heuristic is used to bring data into the cache before the actual load occurs. The effectiveness of the algorithm for eliminating misses, while positioning the prefetches a substantial distance from actual loads, is studied.

4.1 Cache Load Instruction

For the compiler to assist the processor in prefetching, the compiler must have a mechanism to inform the cache that a memory address will be needed. For the purposes of this paper, we postulate a *cache load* instruction, which can be viewed as a no-wait load to a nonexistent register. A cache load should have all of the address modes of a machine's regular load instruction. The prefetch for an address looks just like a normal load except no register is specified as a destination. To the program, a cache load looks like a NO-OP. The only effect on execution is that an instruction issue slot is used and the instruction counter is incremented. On machines that allow multiple instructions to be issued during a cycle, the cache load could be overlapped with instructions.

To implement a no-wait load, we must build a cache that can have multiple outstanding requests. Even if it is acceptable to queue the prefetches so that they are serviced sequentially by the main memory, it will still be necessary to allow multiple requests. If the prefetching fails to prevent a miss, the hardware certainly should not wait for a prefetch to complete before issuing the required load.

Caches that allow more than one outstanding request are being designed [Kro90, SD88] and implemented [GM87]. Scheurich and Dubois present the design for a lockup-free cache for hiding the delays involved in accessing remote locations in a distributed memory multiprocessor. In their paper, one of the methods described for improving processor performance uses a special cache load instruction. RISC architectures, in their attempt to make every instruction be one cycle and make that cycle as short as possible, have already implemented non-blocking load instructions (e.g., IBM RT). Since the nearest memory is more than one machine cycle away, by not blocking on memory, other computations not involving the load can be executed in cycles that would otherwise be dead. Thus, the hardware problems involved in the design and implementation of a no-wait cache load instruction seem to be manageable.

A more difficult question for the architect and compiler designer is how to handle run-time faults on cache load instructions. The hardware could be constructed so that a fault on a cache load would not be reported and the load aborted. It remains to be seen how difficult this is to implement. When the hardware cannot prevent faults, the compiler will have to prevent cache loads from generating either memory protection faults or page faults that would not otherwise occur. Most of the faults can be prevented by attempting to guarantee that prefetching does not

```

forall statements S in program
  if S is a DO
    then
      iv = loop induction variable
      s = loop step
    end
    forall array references R in statement S
      if iv appears in subscript of R
        then
          replace iv with iv + s and prefetch
        end
      end
    end
  end
end

```

Figure 6: Insert Prefetch Instructions

occur for iterations that will not execute. If the last iterations of every loop that attempted prefetching were peeled out into an epilogue, most spurious faults could be eliminated.

As defined, a cache load instruction does not represent a major addition to the complexity of a processor. For some processors (like the IBM RT, which already allows more than one outstanding load), it may even be possible to add the instruction using functions already present on the silicon. A cache load can be used in the compiler like a regular load instruction at any point previous to the actual load. In particular, VLIW and RISC architectures with non-blocking loads and delayed branches may have many empty instruction issue slots that can be replaced with cache loads, making the cache prefetching free.

4.2 Software Strategy

We now outline a simple method for identifying data to prefetch. Most accesses during execution are from within the inner loops. The references with the greatest possibility of generating a large number of misses are array references that refer to different elements on each iteration. For example, any array subscript that uses the innermost loop induction variable will be accessing different values on every iteration of the inner loop. These are the likely candidates for prefetching. For prefetching to be effective at reducing miss delays for the processor, the prefetch must precede the actual load by enough time to allow the load from memory to cache to complete, but not so far that the data might be flushed out of the cache before being used.

The algorithm (Figure 6) for software prefetching is simple—it is based on the observation that a single loop iteration will usually provide enough exe-

```

DO I = 1, N
  DO J = 1, N
    CALL STORE(A(I,J),4,TIME,1)
    CALL PREFET(A(I,J+1),4,TIME,2)
    A(I,J) = 0
    DO K = 1, N
      CALL LOAD(A(I,J),4,TIME,3)
      CALL LOAD(B(I,K),4,TIME,4)
      CALL PREFET(B(I,K+1),4,TIME,5)
      CALL LOAD(C(K,J),4,TIME,6)
      CALL PREFET(C(K+1,J),4,TIME,7)
      CALL STORE(A(I,J),4,TIME,8)
      A(I,J) = A(I,J) + B(I,K)*C(K,J)
    ENDDO
  ENDDO
ENDDO

```

Figure 7: PFC-Sim with Prefetching — Matrix Multiplication

cution time to allow a cache load to complete, but it is unlikely to access enough data to flush items that have recently been prefetched. The heuristic is: if the innermost induction variable is present anywhere in an array subscript, then add the loop step to the innermost induction variable and issue a cache load instruction for the resulting expression.

The preprocessor for PFC-Sim was modified to perform prefetching, using a cache load instruction and the described heuristic algorithm. There are two important details of the implementation. In a single pass over the program, any array subscript that used the explicit induction variable associated with the textually most recent DO loop is considered to be an array access that could profit from prefetching. This causes initialization in an outer loop to generate prefetches (since there is yet no knowledge of the inner loops), but cleanup after an inner loop does not cause prefetches to be generated (since the inner loop is now the most recently defined). The other detail is the placement of the cache load in the loop. Every statement that requires a cache load is immediately followed by its prefetch. This effectively predicts that if loop iteration *I* follows a certain control flow path through the inner loop, then iteration *I* + 1 will follow the same path.

Figure 7 shows PFC-Sim output with prefetches for matrix multiplication. The only three prefetch instructions are in the innermost loop for the next values of the *B* and *C* arrays and in the middle loop for the next element of *A* (the prefetch of *A* shows a situation where PFC-Sim has yet to detect the inner

```

DO I = 2, NXP1
  IF (I .LT. NXP1) THEN
    CALL LOAD(E(NVAR,I-1),8,TIME,766)
    CALL PREFET(E(NVAR,I+1-1),8,TIME,765)
    CALL LOAD(E(NVAR,I+1),8,TIME,764)
    CALL PREFET(E(NVAR,I+1+1),8,TIME,763)
    EDIFF = E(NVAR,I+1) - E(NVAR,I-1)
  ENDIF
  IF (I .EQ. NXP1) THEN
    EDIFF = 0.0DO
  ENDIF
ENDDO

```

Figure 8: PFC-Sim with Prefetching - a portion of SHEAR

loop). A second example, Figure 8 (from the subroutine FFTB in SHEAR), demonstrates the placement of prefetches in IF clauses. It also has a redundant prefetch. A loop carried input dependence, with distance of two, exists between the accesses of array E. On the third iteration of the loop, the value accessed by the first load will be exactly the value fetched during the first iteration by the second load. Except the first iteration, cache loads issued to prefetch for the first instruction will discover the value to be in the cache already.

A minor modification to the prefetching algorithm allows successful prefetching of data that cannot be handled by either long cache lines or hardware prefetching. In particular, when induction variables are located in subscripts of nested array accesses, the correct value to prefetch can be calculated. The correct increment to add to the induction variable in the prefetch instruction is the depth of the array. This allows the subscript, which is itself an array reference, to be prefetched one iteration before it is used. Nested array indexes occur frequently in programs that calculate using sparse matrices. The code fragment in Figure 9 demonstrates prefetching of nested array references.

Examples like the one in Figure 9 would cause problems for long cache lines and hardware prefetching—both are likely to bring data into the cache that will never be accessed (unless the mechanism is lucky) when confronted with programs that use one array to determine the location in a second array. As the example shows, software prefetching works well even in these cases.

```

DO L = 1, N
  DO I = 1, M
    CALL LOAD(A(INDEX(I),L),8,TIME,364)
    CALL PREFET(A(INDEX(I+1),L),8,TIME,363)
    CALL PREFET(INDEX(I+2),4,TIME,362)
    CALL LOAD(B(I),8,TIME,361)
    CALL PREFET(B(I+1),8,TIME,360)
    B(I) = A(INDEX(I),L)
  ENDDO
ENDDO

```

Figure 9: PFC-Sim with Prefetching — Index Array

4.3 Performance

To be effective, software prefetching must succeed in two ways. It must eliminate misses, and there must be enough computation between each prefetch and the actual load to permit the data to arrive from the main memory. In addition, a desirable property of any prefetching scheme is that it not cause enormous increases in traffic between memory and cache, lest the savings on cache size be offset by the increased bandwidth requirements.

4.3.1 Hit Ratio

The effectiveness of software prefetching was tested on the programs in RiCEPS. Figure 10 compares the hit ratios of three caches for each program: a 32K LRU 4-way set associative with 4-byte cache line, the same cache with hardware prefetching, and the same cache with software prefetching. 10 of the 12 programs have hit ratios of over 98% for caches with software prefetching. Software prefetching does at least as well as hardware prefetching on every program and successfully prefetches on many programs with non-sequential array accesses. Software prefetching even generates higher hit ratios than very long cache lines for many of the programs and is within a small percentage for the other programs.

The two programs that had hit ratios below 98% both pointed out a limitation in the placement of prefetches, not in the mechanism itself. In PFC-Sim, the memory references are detected in a pass over the parsed program before any other passes occur. Because of this the only induction variables that are prefetched are those explicitly declared in by a DO loop structure. Both MHD2D and MCMB use auxiliary induction variables quite heavily. When the prefetches are added for all of the induction variables detected during PFC's induction variable substitution phase, the number of remaining misses is below

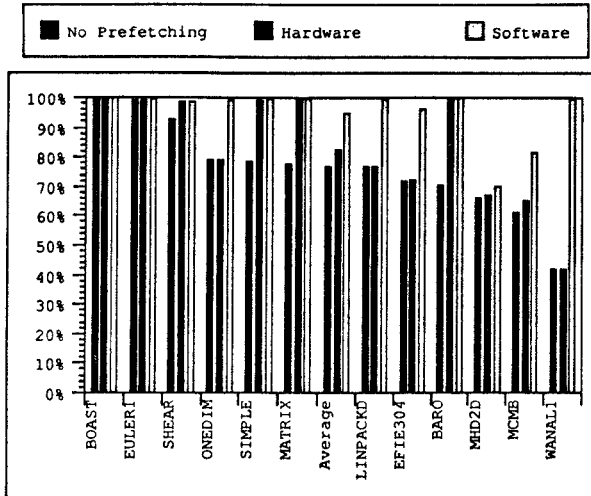


Figure 10: Effect of Software Prefetching - Hit Ratios

3% for both programs.

To summarize, if the compiler algorithm for software prefetching had incorporated “induction variable substitution”, a common capability for vectorizing compilers, almost all cache misses would have been eliminated. Even without detecting auxiliary induction variables, the average miss ratio for a prefetching cache was 95.6%. Excluding the two programs that used alternative induction variables, the average hit ratio would be 99.5%. Thus, prefetching array values one loop before they are needed is very effective for computationally intensive programs, particularly if loop induction variables are identified.

4.3.2 Time Between Prefetch and Load

If cache loads always immediately precede actual loads, then the actual loads will never cause a memory transfer. However, every load would have to wait for the memory latency of the prefetch. Moving the prefetches one loop iteration away from the actual load provides some amount of execution time to hide the memory latency.

In the run-time routines for PFC-Sim, every prefetch was marked with the program execution time at which it was issued. Whenever a load used a value that had a time field, it recorded how much time had elapsed between the cache load and the actual load. After recording the difference, the time field was zeroed to prevent later accesses from recording their delays.²

Figures 11 and 12 show the number of cycles by

²Time was generated assuming a processor where instructions take different amounts of time to complete: floating point add — 3 cycles, floating multiply — 5 cycles, integer add, loads and stores from/to cache — 1 cycle.

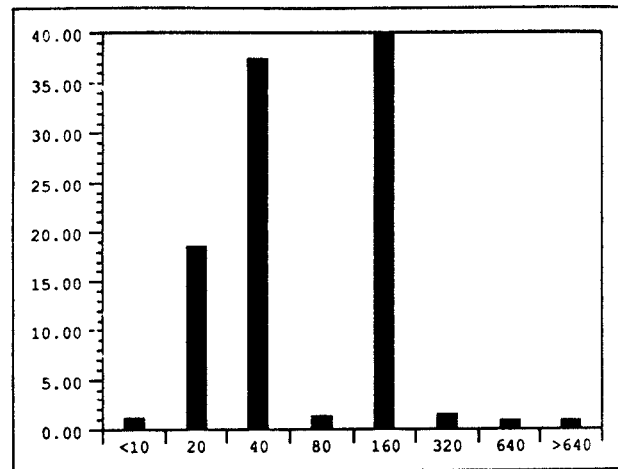


Figure 11: Software Prefetching - Time Between Prefetch and Use (Absolute Totals)

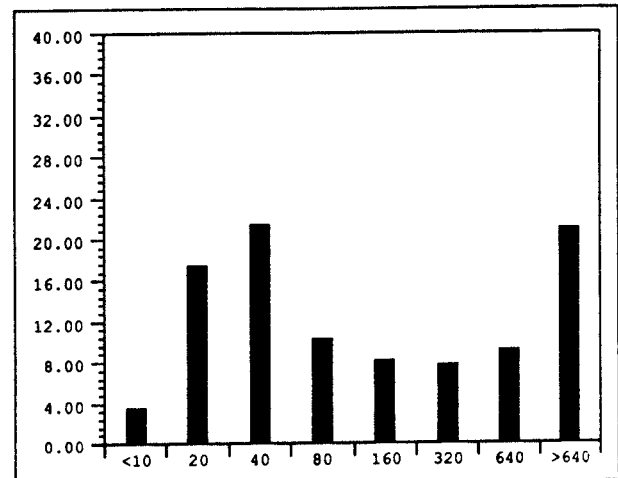


Figure 12: Software Prefetching - Time Between Prefetch and Use (Normalized Totals)

which the cache load preceded the register load for all of the programs in RiCEPS. Figure 11 shows the percentage of all references in RiCEPS. Several programs have significantly more references than the other programs and dominate this graph. Less than 1% of the prefetches preceded the actual reference by less than 10 cycles. About 19% of the prefetches occurred between 10 and 20 cycles before the actual reference. Over 80% of the prefetches occurred over 20 cycles before the actual use. These references should easily return before any use is reached. When longer floating point operation delays were used (such as found on pipelined supercomputers) the prefetches preceded the uses by more than enough to hide any current machines memory delay.

If we normalize each program’s results and weigh

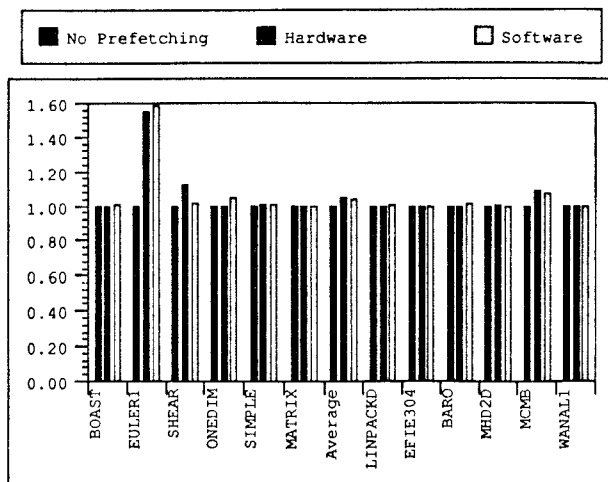


Figure 13: Effect of Software Prefetching — Data Traffic

every program equally, the distribution of time between prefetches and loads is different, as shown in Figure 12. About 3% of the references were less than 10 cycles away from their prefetch. Another 18% were between 10 and 20 cycles away from the first use. These prefetches may not complete on some machines before the actual load is reached. But over 79% of the prefetches occurred 20 cycles ahead of the first use. Loading values one iteration prior to use successfully would hide delays due to cache misses for most of the array accesses in RiCEPS on current memory architectures.

4.4 Data Traffic

One of the reasons for examining the effectiveness of software prefetching was the excellent bandwidth behavior of hardware prefetching. Figure 13 compares the data traffic used in software prefetching with that used normally and that used with hardware prefetching.

For most of the programs, hardware prefetching produced between 0 and 2% more memory traffic than a cache with no prefetching. For these programs, software prefetching required slightly more data (fractions of a percent) than hardware prefetching. When a program fits into the cache and reuses each value a large number of times, the unnecessary data prefetched and the data pushed out (since the cache is not fully associative) can be a significant fraction of the original data transferred. If one looks at the amount of data transferred per floating point operation (or by some other time metric), both of these programs move very little data. The small amount of overhead that is generated looks large because the

original programs required very little data traffic.

Software prefetching maintains the good traffic behavior of hardware prefetching and eliminates miss delays. The major cost of software prefetching will be the amount of time required to issue the requests.

5 Software Prefetching Overhead

Software prefetching would virtually eliminate miss delays for the computationally intensive programs in RiCEPS. If prefetching were free, this could decrease the execution time of programs by up to 50%. For computers with memory approximately 50 cycles away, the average decrease would be over 20%. However, on most computer systems, software prefetching will not be free—it will require that additional instructions to calculate the prefetch address and issue the cache load be executed. In this section we investigate the size of this overhead and present several ways to reduce it.

5.1 Run-time Overhead

Most of the run-time overhead of software prefetching is the additional execution time required to perform the cache loads. Two kinds of computations are added by software prefetching: address generation and the cache loads themselves. Each prefetch requires a single cache load instruction. For each array, there is some amount of address calculation that must occur. This can range from a single addition to a complicated arithmetic expression, possibly including more than one integer multiplication.

If every prefetch brought data into the cache, determining the profitability of any individual prefetch would be relatively simple. The prefetch would be profitable whenever the address generation and prefetch take less time than accessing memory. The prefetching mechanism implemented in this work generates many prefetches for memory locations that are already present in the cache (or are already on their way to the cache) or locations that will not be accessed (at least before the location is bumped from the cache). The benefits of successful prefetching must outweigh all the costs for these unnecessary prefetches.

For the simple prefetching strategy, Table 1 shows that slightly less than one-third of the prefetches cause an actual cache miss and force a value into the cache that is used later. For prefetching to be profitable, memory latency should be greater than 3 times the cost of address generation. Unless memory latencies are very high, the address generation overhead will completely overshadow the overlapping of

| Program | Prefetches that cause Cache Misses |
|----------|--|
| LINPACKD | 24.6% |
| WANAL1 | 58.2% |
| BOAST | 1.7% |
| MCMB | 31.0% |
| MATRIX | 46.4% |
| SIMPLE | 22.2% |
| EFIE304 | 32.6% |
| BARO | 32.3% |
| EULER1 | 2.3% |
| SHEAR | 26.8% |
| MHD2D | 35.1% |
| ONEDIM | 27.4% |
| Average | 28.4% |

Table 1: Cache misses from Prefetches

miss delays with execution. It does not decrease execution time to overlap a memory latency with computation that was added to compute the addresses to be prefetched.

The additional execution time required for address computation was estimated with a minor modification to PFC-Sim. As previously stated, the execution time of the program was always estimated. By changing the estimated time per execution to include an additional statement for every prefetch instruction, the amount of additional time spent by prefetching can be measured. Each prefetch instruction was assumed to require one load (the offset from the previous load), one integer addition, and the prefetch instruction itself. For most programs, the offset is present in registers, reducing the computed overhead values. This estimate is intended to be a reasonable upper bound on the required time.

The average overhead incurred from software prefetching by a program was 28%. MHD2D had a substantial amount of computation for every prefetched value, resulting in a very minimal overhead of 6%. Other programs, like LINPACKD, performed little computation per reference. Adding prefetching to these programs greatly increased the execution time (up to 48%).

With overhead of 28%, software prefetching is unlikely to reduce the execution time of programs. For software prefetching to be effective, methods to reduce the costs will be required. Methods to reduce the overhead of software prefetching are discussed in the next section.

5.2 Reduction of Overhead

Basically, there are three ways to reduce software prefetching overhead: increasing machine parallelism, predicting unnecessary prefetches and saving addresses in registers. The first solution is, in the long run, the best. On a machine that can issue more than one instruction in each cycle, prefetching overhead can almost always be hidden under the costs of the underlying calculation. Since prefetching is so effective in improving the performance of the memory hierarchy, future machines should be designed to take full advantage of it while virtually eliminating the overhead through use of parallelism.

The two remaining methods can be done entirely in software. First, the dependence graph can be used to prevent prefetching of values already present in the cache. Second, the address in a register can be saved between the prefetch and actual load, eliminating the extra address generation. Using these techniques, software prefetching may be made profitable for any processor with a cache and a cache load.

5.2.1 Machine Parallelism

Modern attempts to increase processor performance has led to designs that expose potential machine parallelism to the programmer. The Intel i860 has a mode which allows an instruction to be issued to the floating point and integer unit each cycle. The IBM RS/6000 allows up to 5 operations to be performed in parallel. The Multiflow Trace completely exposed the functional unit pipes to the instruction.

On all of these machines, a cache load instruction could be executed in an otherwise empty instruction (or operation) slot. Since the cache load would have no formal dependence on the other code in the loop, it is very moveable and can probably fill any available hole in the instruction stream.

Without extensive study of the actual instruction streams for one (or more) of the architectures, we can not determine what fraction of the cache load overhead can be overlapped with other instructions. The observed performance of the Intel i860 as compared to the peak performance suggests that on many programs sufficient empty slots may be available to completely hide the cache loads.

5.2.2 Unnecessary Prefetch Elimination

A large percentage of the overhead for the simple software prefetch algorithm is due to generating prefetches for values that are already present in the cache. A better prefetching strategy is to only prefetch references that may be misses (see Figure 14).

```

forall statements S in program
  if S is a DO Statement
    then
      iv = loop induction variable
      s = loop step
    end
    forall array references R in statement S
      if R is a miss and
        iv appears in subscript of R
      then
        replace iv with iv + s and prefetch
      end
    end
  end
end

```

Figure 14: Insert Prefetch Instructions (Estimated Misses Only)

By using the dependence graph to estimate how much memory is accessed by each iteration of a loop, we can determine whether a dependence will reuse a value already present in the cache. The *overflow iteration* [Por89] is the maximum number of iterations of a loop that access less memory than can be contained in the cache at one time. Any dependence carried by a loop with a distance greater than the overflow iteration will result in a cache miss. The following rule can be used to limit prefetching: only prefetch values for which every incoming dependence edge exceeds the overflow iteration.

The overflow iteration was computed by hand for several of the shorter programs in RiCEPS. Table 2 compares the hit ratio and the amount of overhead for the two prefetching strategies. For the tested programs, the overflow iteration did a good job of separating the references that should be prefetched from those that should not.³ The two programs that still had a high percentage of unuseful prefetches were EULER1 and LINPACKD. When the hit ratio was very high, very few prefetches occurred. The overflow iteration version of EULER1 performed about one-third the total number of prefetches, so the total overhead was low, although the percentage of useful prefetches fell only slightly. LINPACKD has triangular loop nests which are hits early and misses after some number of iterations. Prefetching those array references kept the hit ratio for the program high (99.9%).

Using the overflow iteration to limit the number

³When computing the overflow iteration by hand, we have assumed good interprocedural information. If information less exact than regular sections with bounds information is used, the overflow iterations would be less accurate and the number of unnecessary prefetches would be higher.

| Program | Hit Ratio | | |
|----------|-------------|----------------|------------------|
| | No Prefetch | All Prefetched | Limited Prefetch |
| LINPACKD | 75.7% | 99.9% | 99.9% |
| WANAL1 | 41.4% | 99.5% | 99.4% |
| EULER1 | 99.1% | 99.5% | 99.7% |
| BARO | 71.3% | 99.8% | 99.7% |
| MATRIX | 76.7% | 99.9% | 99.7% |
| EFIE304 | 72.4% | 97.6% | 97.1% |
| Average | | 99.4% | 99.3% |

| Program | Useful Prefetches | |
|----------|-------------------|------------------|
| | All Prefetched | Limited Prefetch |
| LINPACKD | 24.6% | 36.9% |
| WANAL1 | 58.2% | 98.7% |
| EULER1 | 2.3% | 2.2% |
| BARO | 32.3% | 56.2% |
| MATRIX | 46.4% | 91.4% |
| EFIE304 | 32.6% | 70.0% |
| Average | 32.7% | 59.2% |

Table 2: Useful Prefetches — After Using Overflow Iteration

of generated prefetches increased the likelihood that any given prefetch actually caused a useful cache load. With every possible value being prefetched, less than one-third of the prefetches caused useful data to be moved into the cache. When the overflow iteration was used to reduce the number of prefetches, almost six out of ten remaining prefetches prevented a later cache miss.

Use of the overflow iteration eliminated over 54% of the prefetches from the six test programs. This reduced the overhead for the prefetching instructions on the six programs from over 31% down to about 14%. If the other programs had shown equivalent reductions, the overhead of software prefetching would be reduced to 12%. This reduction in overhead resulted in a negligible decrease in hit ratio (0.1%). At only 12% overhead, software prefetching is an attractive alternative on computers with a cache miss penalty of 20 or more.

By examining several thousand lines of code to determine which references would be misses, we arrived at several observations about the calculation of the overflow iteration. In general, it is very easy to look at a small to medium loop and roughly determine the overflow iteration by hand, but it is easy to overlook references. In the first modification of WANAL1, prefetches to two references were incor-

rectly eliminated. Those two references lowered the total program hit ratio by 6%.

In summary, by using the overflow iteration in a compiler algorithm to reduce the number of spurious prefetches can reduce the overhead of software prefetching by as much as a factor of three, making the scheme attractive for a number of machines.

5.2.3 Register Allocation

The overhead of prefetching can be further reduced by keeping addresses generated for the prefetch in registers for later use at the corresponding load. As defined in the simple prefetching strategy described above, every prefetch generates its own address, and then the load also generates an address. When the right value is being prefetched, these two addresses will be the same. By keeping the value in a register between the cache load and the actual load, the second address generation can be avoided.

The drawback to this scheme is that saving the address between the two loads greatly increases the lifetime of the address temporary, which in turn increases the register pressure in the program. This will cause more values to be spilled from registers during execution. When the address temporary is spilled, the cost of the cache load is a register store, a register load and the single cache load instruction. To measure the impact on register allocation, several programs were modified and ported to the R^n environment. The R^n compiler does graph coloring register allocation [CAC⁺81, Cha82] and estimates the amount of spilling that will occur during execution.

Every program ported to R^n had every prefetchable reference subscript replaced by a temporary variable. The temporary for the next iteration was then calculated in the statement after its use. This caused the address temporary to have a lifetime of one full iteration of the loop, the amount of time that occurs during software prefetching. The overflow iteration had already been used to reduce the number of prefetches that occurred before porting to R^n .

The six programs consisted of a total of 62 routines. The R^n compiler is still under development, and attempting to compile actual programs uncovered a number of bugs. In particular, only 38 of the routines could successfully pass through register allocation. For these, the original versions of the routines resulted in 223 variables being spilled at an estimated cost of 221,892 cycles. After prefetching address variables were added, the number of spills increased by 58 to 281, and the estimated cost was 708,623 cycles. When all optimizations were activated in the compiler, the number of spills in the original program in-

creased to 387 requiring 353,875 cycles. Prefetching required an additional 74 register spills and 940,813 cycles.

Prefetching increased the number of spills by less than 25% for both the optimized and non-optimized code. The spills tended to occur in more nested loops, increasing the overall cost of spilling by 320% for the non-optimized code and 266% for optimized code. The total cost of spilling is minimal when compared to the number of accesses that actually occur in the programs. WANAL1 alone generates trillions of references.

Unfortunately, the longer routines were less likely pass through the register allocator. For every routine that did pass through the compiler, the estimated spill cost was less than the number of prefetches. Since the cache should maintain any scalar that is used on every iteration, the cost of software prefetching is less than three transfers between processor and cache.

The cost of eliminating duplicate address generation, by saving addresses in registers across iterations is unclear from the experiments. Many routines could not be measured, and the exact correlation between cycles as estimated by R^n and measured by PFC-Sim is unknown. However, an estimate of the likely effect can be derived. The additional spills from software prefetching are likely to be relatively small (on the order of 20 to 50% more spill sites), but are likely to be in more heavily executed sections of the program (spill costs climbed over 200%).

In summary, using registers to maintain addresses between the prefetch and the actual load may or may not substantially reduce the overhead involved in prefetching. This preliminary study is encouraging, but inconclusive.

6 Conclusions

From the results presented here, it should be clear that software prefetching is an attractive strategy for reducing the effects of long memory latencies without notably increasing the bandwidth required to support traffic between main memory and cache. Software prefetching should be particularly useful on high-performance systems that can issue more than one instruction per cycle—if the costs of issuing the prefetch instruction and computing the prefetch address can be completely hidden under other instructions, the reduction in execution time can be substantial.

Even when the cost of software prefetching cannot be eliminated by hardware, software prefetching can be made practical for long-latency machines by a number of optimizations, particularly the elimination

of unnecessary prefetches and assignment of addresses to registers.

Although this study clearly establishes the potential of software prefetching for reducing apparent latency in a system—or reducing the cost of a system by making it possible to build a smaller cache—it remains to be established in practice whether the advanced design of new high-performance microprocessors will reduce the prefetching overhead sufficiently to realize the large potential gains.

References

- [BCK⁺89] M. Berry, D. Chen, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Samah, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, R. Goodrum, and J. Martin. The Perfect Club: Effective Performance Evaluation of Supercomputers. *The International Journal of Supercomputer Applications*, 3(3), Fall 1989.
- [CAC⁺81] Gregory Chaitin, Marc Auslander, Ashok Chandra, John Cocke, Martin Hopkins, and Peter Markstein. Register allocation via coloring. *Computing Languages*, 6, 1981.
- [Cha82] Gregory Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the SIGPLAN 82 Symposium on Compiler Construction*, 1982.
- [CK89] Steve Carr and Ken Kennedy. Blocking linear algebra codes for memory hierarchies. In *Proceedings of the Fourth SIAM Conference on Parallel Processing for Scientific Computing*, Chicago, December 1989.
- [CKC90] David Callahan, Ken Kennedy, and Steve Carr. Improving register allocation for subscripted variables. In *Proceedings of the ACM SIGPLAN 90 Conference on Program Language Design and Implementation*, pages 53–65, White Plains, NY, June 1990.
- [CKP90] David Callahan, Ken Kennedy, and Allan Porterfield. Analyzing and visualizing performance of memory hierarchies. In Margaret Simmons and Rebecca Koskela, editors, *Instrumentation for Visualization*, Frontier Series, pages 1–26. ACM Press, 1990.
- [CP90] David Callahan and Allan Porterfield. Data Cache Performance of Supercomputer Applications. In *Supercomputer 90*, 1990.
- [Don88] Jack Dongarra. Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment. *Computer Architecture News*, 16(1), March 1988.
- [GM87] C. E. Gimaç and V. M. Milutinovic. A survey of RISC processors and computers of the mid-1980's. *Computer*, 20(9), September 1987.
- [Jou90] Norman Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *The 14th Annual International Symposium on Computer Architecture*, Seattle, WA, May 1990.
- [Kro90] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *The 8th Annual International Symposium on Computer Architecture*, Minneapolis, MN, May 1990.
- [Por89] Allan Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Rice University, 1989. Technical Report Number Rice COMP TR89-93.
- [SD88] C. Scheurich and M. Dubois. Concurrent miss resolution in multiprocessor caches. In *1988 International Conference on Parallel Processing*, 1988.