An Effective On-Chip Preloading Scheme To Reduce Data Access Penalty

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

Abstract

Conventional cache prefetching approaches can be either hardware-based, generally by using a one-block-lookahead technique, or compiler-directed, with insertions of non-blocking prefetch instructions. We introduce a new hardware scheme based on the prediction of the execution of the instruction stream and associated operand references. It consists of a reference prediction table and a look-ahead program counter and its associated logic. With this scheme, data with regular access patterns is preloaded, independently of the stride size, and preloading of data with irregular access patterns is prevented. We evaluate our design through trace driven simulation by comparing it with a pure data cache approach under three different memory access models. Our experiments show that this scheme is very effective for reducing the data access penalty for scientific programs and that is has moderate success for other applications.

1 Introduction

The time when peak processor performance will reach several hundred MIPS is not far away. Such instruction execution rates will have to be achieved through technological advances and enhanced architectural features. Superscalar or multifunctional unit CPU's will increase the raw computational speed. Efficient handling of vector data will be necessary to provide adequate performance for scientific programs. Memory latency will be reduced by cache hierarchies. Processors will have to be designed to support the synchronization and coherency effects of multiprocessing. Thus, we can safely envision that the processor chip will include several functional units, first-level instruction and data caches, and additional hardware support functions. In this paper, we propose the design of an on-chip hardware support function whose goal is to reduce the memory latency due to data cache misses. We will show how it can reduce the contribution of the on-chip data cache to the average number of clock cycles per instruction (CPI)[3].

The component of the CPI due to cache misses depends on two factors: miss ratio and memory latency. Its importance as a contributor to the overall CPI has been illustrated in recent papers [1, 5] where it is shown that the CPI contribution of first-level data caches can reach 2.5.

Current, and future, technology dictates that on-chip caches be small and most likely direct-mapped. Therefore, the small capacity and the lack of associativity will result in relatively high miss ratios. Moreover, pure demand fetching cannot prevent compulsory misses. Our goal is to avoid misses by preloading blocks before they are needed. Naturally, we won't always be successful, since we might preload the wrong block, fail to preload it in time, or displace a useful block. The technique that we present will, however, help in reducing the data cache CPI component .

Our notion of preloading is different from the conventional cache prefetching [11, 12] which associates a successor block to the block being currently referenced. Instead, the preloading technique that we propose is based on the prediction of the instruction stream execution and its associated operand references. Since we rely on instruction stream prediction, the target architecture must include a branch prediction table. The additional hardware support that we propose takes the form of a look-ahead program counter (LA-PC) and a reference prediction table and associated control (RPT). With the help of the LA-PC and the RPT, we generate concurrent cache loading instructions sufficiently ahead of the regular load instructions, so that the latter will result in cache hits. Although this design has some similarity with decoupled architectures [13], it is simpler since it requires significantly less control hardware and no compiler support.

The rest of the paper is organized as follows: Section 2 briefly reviews previous studies of cache prefetching. Section 3 introduces the basic idea and the supporting design. Section 4 explains the evaluation methodology. Section 5 reports on experiments. Section 6 contrasts our hardware-only design to a compiler solution. Concluding remarks are given in Section 7.

2 Background and Previous work

2.1 Hardware-based prefetching

Standard caches use a demand fetching policy. As noted by Smith [12], cache prefetching, i.e., the loading of a block before it is going to be referenced, could be used. The pure local hardware management of caches imposes a one block look-ahead (OBL) policy i.e., upon referencing block i, the only potential prefetch is to block $i+1$. Upon referencing block i, the options are: prefetch block $i+1$ unconditionally, only on a miss to block i, or if the prefetch has been successful in the past. Since memory latencies are high, relative to processor speed, prefetching introduces additional risks of a processor stalling, because the memory bus is busy servicing a yet unneeded prefetched block rather than a current miss. This leads Przybylski [11] to argue against complex (pre)fetch strategies because either there is not enough memory bandwidth or because misses are too temporally clustered.

Write buffers, *stream buffers*[5], and *lockup-free* caches [6] are three mechanisms that allow the overlap of processor operation with cache requests. Write buffers allow delaying the writes in favor of more urgent cache loads. Stream buffers are FIFO queues that are filled sequentially starting from the missing block address. They work best for I-caches. Lockup-free operation permits the initiation of cache loads (assuming no hazards) for uncached blocks that will be referenced in subsequent instructions while the fetching of some block is in progress.

If some form of branch prediction mechanism is present, then prefetching based on instruction stream execution can be implemented (see [4] for an extensive study of prefetch instruction buffers, instruction target buffers, and I-caches). A sophisticated data prefetching scheme has been proposed at Illinois [8] in the context of a multiprocessor running a scientific workload. In this case, data prefetching becomes more appealing when memory latency increases and when more pipelining becomes available in the interconnection network. Implicit prefetching is present in decoupled architectures [13]. Two instruction streams operate concurrently, communicate via queues, and drive two execution units: one for data access and one for functional operations. The data access stream can be "ahead" of the functional stream and hence prefetch operands needed in the near future.

2.2 Compiler-directed prefetching

Patterson and Hennessy [3] rightly argue that compiler technology should not be separated from architectural design. Compiler-directed prefetching is a case in point. For example, Porterfield [10] proposes a non-blocking *cache load* instruction. This instruction should be positioned enough in advance of the actual use of its operand, e.g., in iteration i of a loop so that it can be used in iteration $i+1$. Gornish et al. [2] improve on this method by finding the earliest time at which such prefetching can be performed.

3 Data Cache Preloading

Prefetching based on sequentiality (OBL, I-stream buffer) can be successful for the optimization of I-caches, but much less so for D-caches. Therefore, we turn our attention solely to the case of D-caches. The basis of our hardware-based scheme is to predict the instruction execution stream and the data access patterns far enough in advance, so that the required data can be *preloaded* and be in the cache when the "real" memory access instruction is executed.

3.1 Motivation

The design of the hardware support function that we propose is based on the regularity of memory access patterns when they exist and in preventing preloading when the access patterns are unpredictable.

Consider a program segment with m-nested loops indexed by $I_1, I_2, \cdots, I_m$. Let LP_{I_i} be the set of statements with data references in the loop at level i. We denote the subscript expression in an LP_{I_i} as constant, linear, or irregular. Given a data reference r, we can then divide the memory access patterns into four categories:

Pattern	Description	examples
scalar	simple variable reference	index, count
zero stride	$r \in LP_{I_i}$ with subscript expression unchanged w.r.t I_i	A[I_1, I_2] in LP_{I_3} tab[I_1].off in LP_{I_2}
constant stride	$r \in LP_{I_i}$ with subscript expression linear w.r.t I_i	A[I_1] in LP_{I_1} A[I_1,I_2],A[I_2,I_1] in LP_{I_2}
irregular	none of the above	A[B[I]] in LP_I A[I,I] in LP_I Linked List

The difference between *scalar* and *zero stride* is that the latter is a reference to a subscripted array element with the subscript being an invariant at that loop level but modifiable at an outer level. Obviously, caches work well for *scalar* and *zero stride* references. Caches with large block sizes and most of the prefetch strategies discussed previously can improve the performance for the *constant stride* category if the stride is small but will be of no help if the stride is large. Our goal is to generate preloads in advance for uncached blocks in the *scalar*, *zero stride*, and *constant stride* access categories independently of the size of the stride. At the same time, we will avoid unnecessary preloading for the *irregular* accesses. Our scheme would be most appropriate for high-performance processors with relatively small first-level caches with a small block size running programs where the data access patterns are regular but not necessarily of stride 1.

Data access patterns of load/store instructions will be kept in a Reference Prediction Table (RPT) which will be accessed ahead of time by a Look-Ahead Program Counter (LA-PC). The LA-PC will be incremented as a regular PC and modified appropriately with the help of a Branch Prediction Table (BPT). In an ideal situation, the LA-PC would run δ cycles ahead of the PC, where δ is the latency to access the next level in the memory hierarchy.

For example, consider the usual matrix multiplication loop (for more detail, see Section 3.3):

```
int A[100,100],B[100,100],C[100,100]
for i = 1 to 100
    for j = 1 to 100
        for k = 1 to 100
            A[i,j] += B[i,k] × C[k,j]
```

and the pseudo-assembly RISC-like code version of the computational part of the inner loop, assuming that the subscripts are kept in registers:

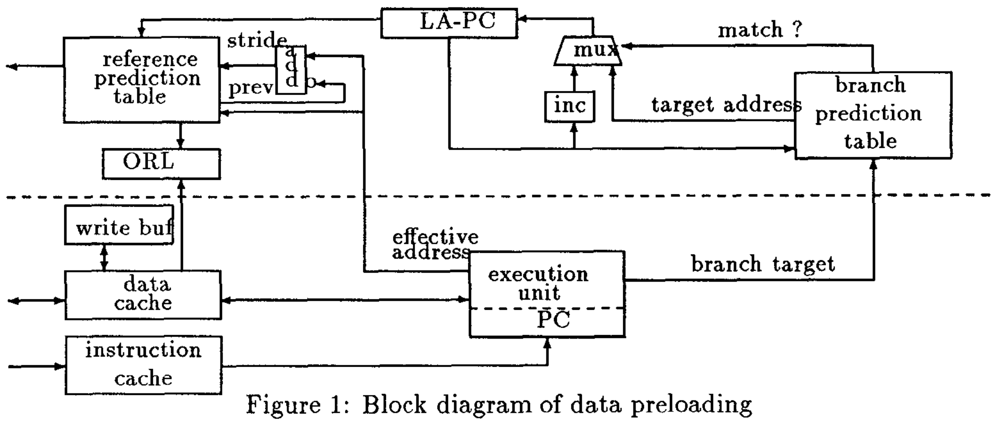

Figure 1: Block diagram of data preloading

```
addr  instruction          comment
500   lw    r4, 0(r2)      ; load B[i,k]    stride 4 B
504   lw    r5, 0(r3)      ; load C[k,j]    stride 400 B
508   mul   r6, r5, r4     ; B[i,k] × C[k,j]
512   lw    r7, 0(rl)      ; load A[i,j]    stride 0
516   addu  r7, r7, r6     ; +=
520   sw    r7, 0(rl)      ; store A[i,j]   stride 0
524   addu  r2, r2, 4      ; ref B[i,k]
528   addu  r3, r3, 400    ; ref C[k,j]
532   addu  rll, rll, 1    ; increase k
536   bne   rll, rl3, 500  ; loop
```

At steady state, the RPT will contain entries for the three load *lw* and the *sw* instructions. Since each iteration of the inner loop accesses the same location of A[i,j] (*zero stride*), no preload will be requested for it. Depending on the block size, references to B[i,k] (*constant stride*) will either be preloaded at every iteration (block size = 4), or every other iteration (block size = 8), and so on. Load references to C[k,j] (*constant stride* with a stride larger than the block size) will generate a preload instruction every iteration. If the LA-PC is sufficiently ahead of PC (cf. Section 3.4), the data will be present at the next iteration.

3.2 Basic Block Diagram

An overall block diagram of the target processor is shown in Figure 1. The bottom part of the figure abstracts a standard RISC processor with on-chip data and instruction caches. The additional top part contains:

- The Branch Prediction Table (BPT) necessary to predict the value of LA-PC (naturally, it will be used also for PC). BPT designs have been thoroughly investigated [7, 9] and we will not repeat these studies here. In our experiments we use the Branch Target Buffer with two-bit state transition design described in [7].

- The Look-Ahead Program Counter (LA-PC), a secondary PC used to predict the execution stream. The LA-PC is incremented or modified via the BPT.

- The Reference Prediction Table (RPT) used for preloading blocks in the data cache. An adder is also needed for stride computations.

- The Outstanding Request List (ORL) that holds the addresses of in progress or outstanding requests.

Each RPT entry (cf. Section 3.3) corresponds to a load/store instruction and contains, hopefully, the address of the operand to be loaded. When the LA-PC hits a load/store instruction that has already been stored in the RPT, a check is made to see whether (1) the state of the entry (cf. Section 3.3) is for *no prediction*, or (2) the data is already in the cache, or (3) the preloading of the block is in progress (by checking ORL). If none of these three conditions is true, a request to load the operand is performed and its address is stored in ORL. If the LA-PC hits a branch instruction that is already in the Branch Prediction Table, a predicted value is given to the LA-PC. It will be corrected later if the prediction was wrong. When the PC encounters a load/store instruction, it modifies (or enters) the corresponding entry in the RPT. The same process occurs for a branch instruction in the BPT.

On a store, a write-allocate, copy-back mechanism is employed. On a replacement, the replaced dirty line is put in the write buffer and handled in the usual way. However, flow dependencies require that each preload or data miss be checked with the contents of the write buffer.

Note that, unlike the instruction prefetch buffer structure in [8] or decoupled architectures, the system does not need to decode the predicted instruction stream. Instead, the prediction mechanism is based on the history information of the execution stream.

3.3 Reference Prediction Table – RPT

The Reference Prediction Table (RPT) is used to keep track of previous reference addresses and associated strides for load and store instructions. In the following we will give examples using load instructions (cf. Section 3.5 for the handling of stores). In our current design, RPT is organized as a cache. Each RPT entry has the following format (see Figure 2).

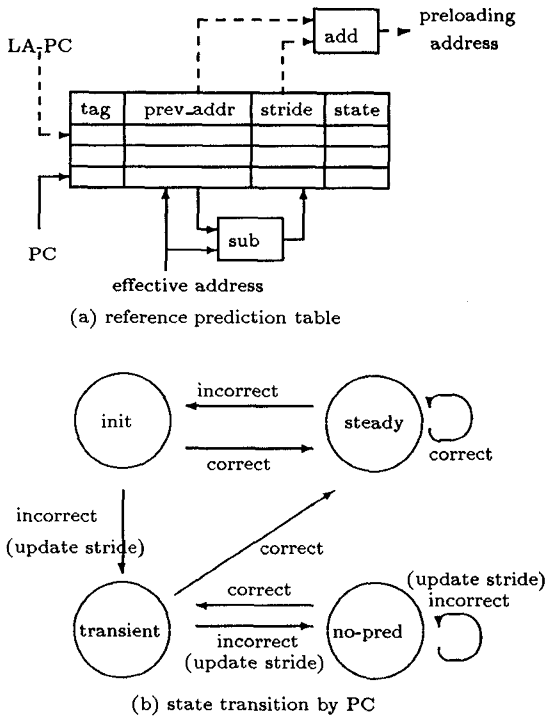

Figure 2: Reference Prediction

- *tag:* corresponds to the address of the Load/Store instruction
- *prev_addr:* the last (operand) address that was referenced when the PC reached that instruction.
- *stride:* the difference between the last two addresses that were generated when a state transition (see below) occurred from *init* to *transient.*
- *state:* a two-bit encoding (4 states) of the past history; it indicates how further preloading should be generated. The four states are:
  - *initial:* set at first entry in the RPT or after the entry experienced an incorrect prediction from *steady* state.
  - *transient:* corresponds to the case when the system is not sure whether the previous prediction was good or not. The new stride will be obtained by subtracting the previous address from the currently referenced address.
  - *steady:* indicates that the prediction should be stable for a while.
  - *no prediction:* disables the preloading for this entry for the time being.

When the LA-PC encounters a load instruction, there are two mutually exclusive possibilities:

- A.1. No action.
  There is no corresponding entry in the RPT, or there is an entry in state *no prediction.*
- A.2. Potential preload.
  There is a corresponding entry. A block address (*prev_addr* + *stride*) is generated. If the block is uncached and the address is not found in the Outstanding Request List (ORL), a preload is initiated. This implies sending a request to the next level of the memory hierarchy, or buffering it if the communication channel is busy. The address of the request is entered in the ORL.

When the PC encounters a load/store instruction with effective operand address *addr*, the RPT is updated as follows: (To make it clear, we denote *correct* by the condition: $addr = (prev\_addr + stride)$ and *incorrect* by the condition: $addr \neq (prev\_addr + stride)$.)

- B.1. There is no corresponding entry. The instruction is entered in RPT, the *prev_addr* field is set to *addr*, the *stride* to 0, and the *state* to *initial.*
- B.2. There is a corresponding entry. Then:
  - (a) Transition –
    When *incorrect* and *state* = *initial*:
    Set *prev_addr* to *addr*, *stride* to (*addr* − *prev_addr*), and *state* to *transient.*
  - (b) Moving to/being in steady state –
    When *correct* and
    (*state* = *initial*, *transient*, or *steady*):
    Set *prev_addr* to *addr*, leave *stride* unchanged, and set *state* to *steady.*
  - (c) Steady state is over; back to initialization –
    When *incorrect* and *state* = *steady*:
    Set *prev_addr* to *addr*, leave *stride* unchanged, and set *state* to *initial.*
  - (d) Detection of irregular pattern –
    When *incorrect* and *state* = *transient*:
    Set *prev_addr* to *addr*, *stride* to (*addr* − *prev_addr*), and *state* to *no prediction.*
  - (e) No prediction state is over; back to transient
    When *correct* and *state* = *no prediction*:
    Set *prev_addr* to *addr*, leave *stride* unchanged, and set *state* to *transient.*
  - (f) Irregular pattern –
    When *incorrect* and *state* = *no prediction*:
    Set *prev_addr* to *addr*, *stride* to (*addr* − *prev_addr*), and leave *state* unchanged.

Figure 3 illustrates how the Reference Prediction Table is filled and used when the inner loop of the matrix multiplication code is executed. We restrict our example to the handling of the 3 load instructions. Let us assume that the loop code starts at address 500, i.e., the load instructions are at addresses 500, 504, and 512, and that the base addresses of matrices A, B and C are respectively at locations 10,000, 50,000, and 90,000.

Before the start of the first iteration, the RPT can be considered empty since there won't be any entry corresponding to addresses 500, 504, and 512. Let us assume also that no element of A, B, or C has been cached.

When the LA-PC gets the value 500, a check is made in the RPT to see if there is a corresponding entry.

tag	prev_addr	stride	state
500	50,000	0	*init*
504	90,000	0	*init*
512	10,000	0	*init*

After iteration 1
(a)

tag	prev_addr	stride	state
500	50,004	4	*transient*
504	90,400	400	*transient*
512	10,000	0	*steady*

After iteration 2
(b)

tag	prev_addr	stride	state
500	50,008	4	*steady*
504	90,800	400	*steady*
512	10,000	0	*steady*

After iteration 3
(c)

Figure 3: Example: Filling RPT entries

Since there is none, no action is taken (cf. A.1 above). The same "no action" occurs when the LA-PC takes the values 504 and 512.

When the PC executes for the first time the load instruction at address 500, there is no corresponding entry. Therefore, the instruction is entered in RPT with its *tag* (500), the *prev_addr* field set to the address of the operand, i.e., 50,000, the *stride* set to 0, and the *state* to *initial* (cf. B.1 above). Similar actions are taken for the other two load instructions (cf. Figure 3.a). In all three cases, there will be cache misses.

Assume now that the LA-PC is 4 instructions ahead of PC and that a branch taken has been predicted at the end of the loop. While the PC controls the end of the normal execution of the first iteration, the LA-PC will be reset at 500. Since there is a corresponding entry in RPT, we are in a preload situation (cf. A.2). The block address to be preloaded is $(50,000 + 0) = 50,000$. The block is cached and therefore no action is taken. The same holds true when the LA-PC hits the other two load instructions.

When the PC executes the load instruction at address 500 at the beginning of the second iteration, we are in the situation described as "transition" (cf. B.2.a). Therefore, the *prev_addr* field is set to 50,004, the stride to 4, and the *state* to *transient*. Depending on the block size, we could have a cache miss (block size = 4 bytes) or a cache hit (block size > 4 bytes). A similar action is taken for the load at address 504 (cf. Figure 3.b). However, for the load at instruction 512, we are in the situation "moving to steady state". The *prev_addr* and *stride* fields are unchanged and the *state* becomes *steady*. Of course, we had a cache hit.

When LA-PC hits instruction 500 for the third time, we are again in situation A.2. The block address $50,004 + 4 = 50,008$ is generated. If there is a cache miss (block size $\leq$ 8 bytes) a preload request is generated. A preload request will be generated for address 90,800 (instruction 504) and the preload for address 10,000 at address 512 will be squashed since the block is still in the cache.

During the third iteration, when PC hits instructions 500 and 504 the changes shown in Figure 3.c will occur in the RPT table. No change will occur for the load at address 512. The same process will be repeated for all further iterations.

In summary, *scalar* and *zero stride* references will pass from the *initial* to the *steady* state in one transition (instruction 512). The *constant stride* references will pass through the *transient* state to "obtain" the stride and then stay in *steady* state (instructions 500 and 504). References with two wrong predictions in a row (not shown in the example) will be prevented from being preloaded by passing to the *no prediction* state; they could re-enter the *transient* state, provided that the reference addresses become predictable. For instance, accesses to elements of a triangular matrix may follow such a pattern. Note that the *stride* field is not updated in the transition from *steady* to *initial* when there is an incorrect prediction. In the case of nested loops, after finishing executing the inner loop, the *stride* will most likely be correct for the next iteration of the outer loop. When the execution revisits the inner loop, the preloading scheme will suffer a penalty of one or two iterations depending on the addressing function.

3.4 Look Ahead Distance (LA-distance)

Ideally, we would like to keep a Look Ahead distance between the current PC and the LA-PC equal to the latency δ of the next level in the memory hierarchy. Clearly this can only be approximated, since the LA-distance is variable. Initially, and after each wrong branch prediction, the LA-distance will be set to one, i.e., the LA-PC points to the instruction following the current PC. When a real cache miss occurs or when a preload is not completed by the time the data is actually needed, the current execution is stalled, i.e., the value of PC does not change, while the LA-PC can still move ahead and generate new requests (recall the role of the ORL).

As the LA-distance increases, the data preload can be issued earlier than its actual need so that the average memory access time is reduced. However, the further PC and LA-PC are apart, the more likely the prediction of the execution stream will be incorrect because the LA-distance is likely to cross over more than one basic block. Moreover, we don't want some of the prefetched data to be cached too early and displace other needed data. Therefore, we introduce a system parameter called Look Ahead Limit (LA-limit d) to specify the maximum distance between PC and LA-PC. Thus, the LA-PC is stalled (until the normal execution is resumed) in the following situations: (cf. Section 5) (1) The LA-distance reaches the specific limit d, or (2) the ORL is full.

3.5 Cache misses

On a cache read miss, the cache controller checks the ORL. If the block has already been requested, a "normal" (but less lengthy) stall occurs. We refer to the cycles that the CPU waits for the preloaded block to be in the cache as *hit-wait cycles*. Otherwise, a regular load is issued with priority over the buffered preload requests.

A write miss in the data cache will cause the system to fetch the data block and then update the desired word. If the block size is larger than a single word, we can initiate preloading as for a read miss. When the block size is one word, no preload needs to be issued but a check of the ORL is needed for consistency purposes. In case of a match, the entry in the ORL must be tagged with a discard status so that the data will be ignored when it arrives.

When the LA-PC has to be reset because of an incorrect branch prediction, the buffered preload requests are flushed. Finally, when a preload raises an exception (e.g., page fault, out-of-range violation) we ignore the preload. The drawbacks of a wrong page fault prediction would far outweigh the small benefits of a correct preload.

4 Methodology

4.1 Trace-driven simulation

We evaluated our design through trace driven simulation. Several scientific programs and general applications (see Table 1) were traced on a DEC Station 5000 (R3000 MIPS CPU) using the *pixie* facility. Data references and intervals between two references are recorded in each trace. Two of the applications (Matrix and Spice) are from the SPEC[1] Benchmarks; Matrix is a vectorizable floating-point benchmark and Spice is less likely to benefit from vectorization. Four programs (QCD, MDG, MG3D, and TRFD) are from the PERFECT CLUB benchmarks[14]. Those programs are examples of scientific computation workloads. Pverify is a VLSI CAD tool, written in C, determining whether two boolean circuits are functionally identical. The other programs are Unix utilities. The compilers we used were the RISC C compiler and the MIPS F77 compiler, both with default options.

The traces captured at the beginning of the execution of the SPEC and PERFECT benchmarks were discarded because they are traces of initial routines that generate the test data for the benchmarks. No statistical data was recorded while the system simulated the first 500,000 data accesses. These references were used however to fill up the cache, the BPT, and the RPT in order to simulate a warm start.

Table 2 shows the dynamic characteristics of the workload. The columns below *data references* show the proportions of data references (weighted by their frequency) that belong to the categories mentioned previously. They are one indication of the reference predictability of the program under study. The column *branch prediction miss ratio* shows the outcome of branch predictions with a 256-entry BPT, which functions like a 2-bit Branch Target Buffer[7]. This is a second indication of the reference predictability. The last column *reference prediction miss ratio* denotes the fraction of total incorrect predictions when preloading requests were generated using a direct-mapped RPT of 256 entries. Incorrect reference predictions could be caused by incorrect branch predictions, RPT entry conflicts, and incorrect states in the RPT table. As can be seen, the scientific programs have better predictability.

[1]SPEC is a trademark of the Standard Performance Evaluation Corporation.

Name	Description (problem size)	total # in M instr	refer.	wrt
Matrix	Matrix multiplication (300x300)	48.2	22.2	7.4
Spice	Analog circuit simulation and analysis	42.3	15.6	2.9
QCD	Quantum Chromodynamics (12x12x12x12)	39.7	17.4	4.1
MDG	Liquid water simulation (343 molecules)	38.6	19.2	4.2
MG3D	Seismic migration (125x120)	47.5	18.1	2.5
TRFD	Quantum mechanics	40.0	20.8	4.3
Pverify	Logic verification	38.2	12.9	2.5
Compress	Data compression	66.7	16.1	5.1
Nroff	Text formatter	7.6	1.9	0.5
Asmbler	MIPS assembler	9.2	2.7	0.8

Table 1: Description of benchmarks

We view our scheme mostly as an enhancement to an on-chip D-cache. Nonetheless, we will evaluate the performance of the scheme with various RPT, BPT, and cache sizes and three memory models. All of the data caches, RPT, and BPT in the experiment are direct-mapped and the block size is 16 bytes.

The system is simulated subject to restrictive models for the communication between the various levels of the memory hierarchy (cf. next section). We consider

Name	data references			branch	ref.
	scalar, zero stride	constant stride	irregular	pred. MR	pred. MR
Matrix	0.028	0.957	0.015	0.073	0.050
Spice	0.784	0.094	0.122	0.064	0.119
QCD	0.681	0.117	0.202	0.128	0.121
MDG	0.616	0.366	0.018	0.102	0.220
MG3D	0.676	0.323	0.001	0.038	0.010
TRFD	0.656	0.340	0.004	0.087	0.072
Pverify	0.224	0.354	0.422	0.190	0.444
Compress	0.657	0.169	0.173	0.108	0.132
Nroff	0.805	0.098	0.097	0.149	0.176
Asmbler	0.637	0.138	0.225	0.314	0.299

Table 2: Test program characteristics

the following three types of architectures:

1. Pure Data Cache of size N KByte.
   This is for providing a baseline comparison.
2. Data cache of N KByte; 256-entry RPT and 256-entry BPT
   This design is an "add-cost" design albeit a realistic one.
3. Data cache of $N/2$ KByte; $32N$-entry RPT and $32N$-entry BPT (but no more than 256)
   This "no-cost" model assumes that the RPT and BPT will take as much real estate on the chip as half the D-cache. That accounts approximately for the RPT and BPT entries and the additional logic.

We let N vary from 4 to 64.

4.2 Memory models

Since there are several sources of data requests from the cache/RPT combination (e.g., cache miss, preloading requests) to the next level in the memory hierarchy, we need to decide how to handle contention and concurrency. Contention occurs when either (1) a cache miss is blocked by a previous preload request, or (2) a preload is blocked by a previous preload or a demand cache miss. We will assume, conservatively, that a fetch in progress cannot be aborted. However, a real cache miss will be given priority over outstanding preload requests.

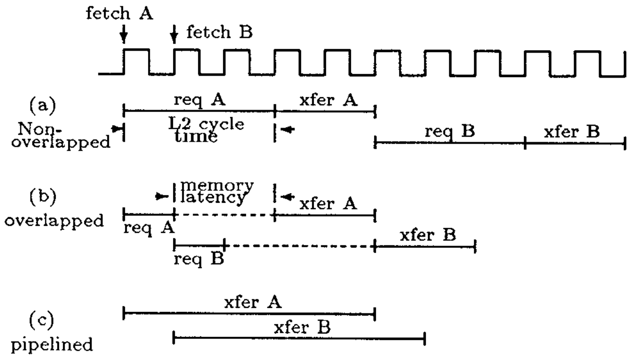

Figure 4: Timing of data access for memory models

We consider three models of access to the next level of the memory hierarchy (cf. Figure 4):

1. *Nonoverlapped.*
   As soon as a request is sent to the next level, no other request can be initiated until the (sole) request in progress is completed. This model is typical of an on-chip cache backed up by a second level cache.
2. *Overlapped*
   The access time for the memory request can be decomposed into three parts: request issuing cycle, memory latency, and transfer cycles. During the period of memory latency other data requests can be in their request issuing or transferring phases. However, no more than one request issuing or transfer can take place at the same time. This model represents split busses and a bank of interleaved memory modules or secondary caches.
3. *Pipelined*
   A request can be issued at every cycle. This model is representative of processor-cache pairs being linked to memory modules through a pipelined packet-switched interconnection network. We assume a *fetch bypass* mechanism, i.e., the desired word is available as soon as the first data response arrives.

The parameters of each model used in our simulation for a 16-byte line are summarized in Table 3.

Model	Component	cycle time	cycles /line
Non-overlapped	secondary level cycle time	6	10
	transfer rate	1/word	
Overlapped	sending request	2	20
	memory access	14	
	transfer rate	1/word	
Pipelined	network latency +memory access	30/line	30

Table 3: Fetch latency of each model

4.3 Other parameters

The traces were simulated assuming a RISC CPU model that executes one instruction per cycle. Each cache write hit takes one extra cycle to update the data in the cache. A 4-entry ORL is used for buffering preload requests. There is a 4-entry write back buffer used for write-backs. Unless otherwise specified, the LA-limit d is equal to 1.5δ, where δ is the latency of the next level in the memory hierarchy.

5 Performance Evaluation

Since our main interest is in the influence of preloading on data cache access, we assume: (1) no I-cache miss (reasonable since preloading is active only during loops; the loop code should be resident in the I-cache), (2) all operations take one cycle (i.e., perfect RISC pipelining), (3) no wait on a cache hit, (4) the processor stalls on a cache miss until the data is in the cache (i.e., no nonblocking loads), (5) there is no miss on the next level of the memory hierarchy.

We simulate the memory system on a cycle-per-cycle basis so that we can have an accurate count of the delays incurred by cache misses and by *hit-wait* cycles.

Assumptions 2 and 4 are related. Preloading could become more advantageous if some instructions (e.g., floating-point instructions) were multi-cycles and were executed after a preload was initiated but before the preloaded data was needed. This would have a tendency to reduce *hit-wait* cycles. Similarly, these multi-cycle instructions would be advantageous for a nonblocking-load scheme, and therefore would reduce the penalty due to cache misses. Measuring exactly these effects would require a cycle-per-cycle simulation

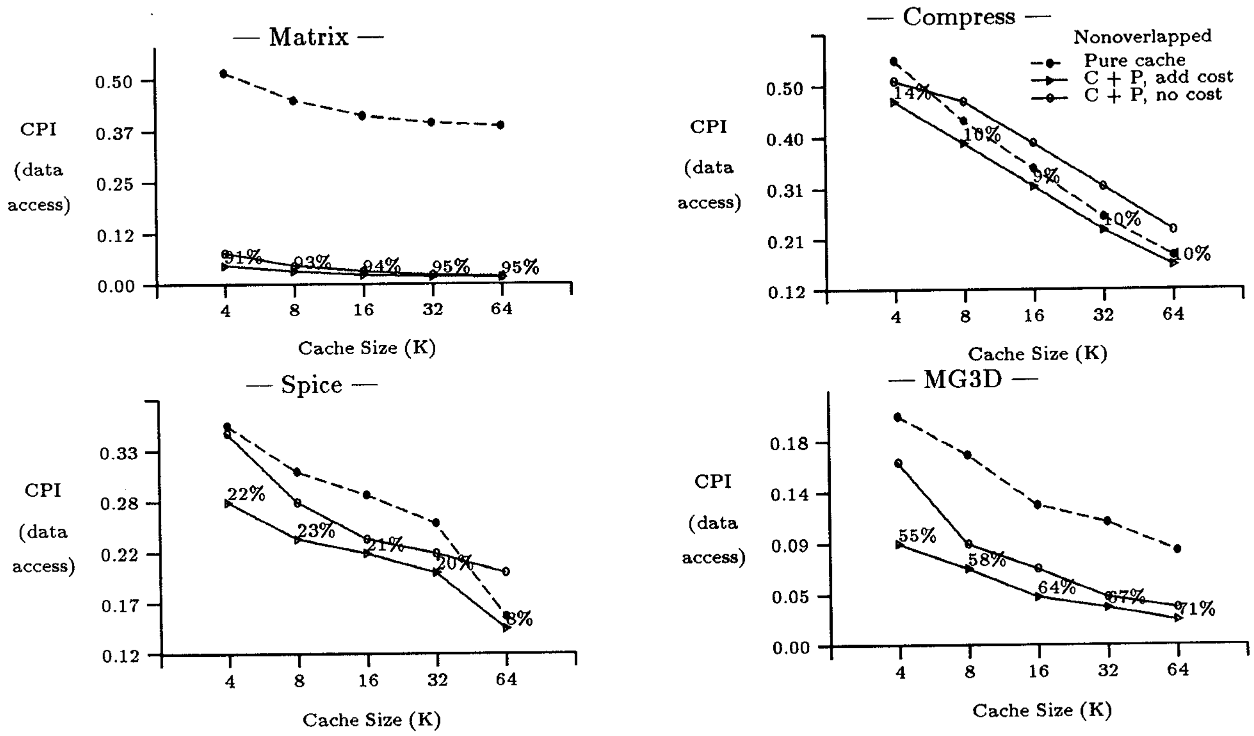

Figure 5: $CPI_{data\ access}$ vs. Cache size on *Nonoverlapped* model

of all instructions and is beyond the scope of this paper. In a first approximation, we feel that assumptions 2 and 4 balance each other.

5.1 Metrics

We present the results of our experiments by using the contribution to the CPI as the main metric. The contribution to the CPI due to data access penalty is:

$$CPI_{data\ access} = \frac{total\ data\ access\ time}{number\ of\ instructions\ executed}$$

In the figures we show the percentage of data access penalty reduced by the preloading scheme, i,e.:

$$\%\text{ of penalty reduced} = \frac{CPI_{cache} - CPI_{preload}}{CPI_{cache}} \times 100$$

where CPI_{cache} corresponds to the pure cache experiment and $CPI_{preload}$ to the "add-cost" model.

We compare the preloading scheme with the equivalent pure data cache design while varying the cache size. To be concise we often contrast only two extreme cases of the benchmarks.

5.2 Preloading performance for the *Nonoverlapped* model

Figure 5 presents the performance of the "no-cost" and "add-cost" organizations with varying cache sizes for the *Nonoverlapped* memory model. The "add-cost" organization always performs better than the pure cache scheme since it has the same amount of cache and, in addition, the preloading component. The "add-cost" organization will always perform better than the "no-cost" since it has more cache with at least the same amount of preloading hardware (the "no-cost" at cache size N Kbyte has the same performance as the "add cost" at cache size $N/2$ Kbyte).

The results show that the "add-cost" preloading scheme can reduce the data access penalty from 10% up to 95% compared to a pure data cache. The additional cost paid for preloading is justified by the significant performance improvement. The relative advantage of an "add-cost" feature increases slightly with cache size for programs with good predictability. As cache size increases, the higher proportion of compulsory misses is compensated by a reduction in the proportion of conflict and capacity misses. The program Spice does not have the above characteristic. From the fact that Spice has a larger CPI contribution and shows an obvious drop in the CPI from 32K to 64K, we conjecture that it has a significant proportion of conflict and capacity misses.

In the case of the "no-cost" organization, the results are mixed. Programs such as Matrix that have very long inter-reference intervals for each individual data element, very large data sets (i.e., non-cache resident), and good predictability, benefit almost as much as in the "add-cost" case. In programs like Spice and MG3D where the data sets are such that there is a need for a cache of minimal size before the hit ratios are high enough, the advantage of the preloading feature is offset by the penalty of halving the cache size until the data locality can be captured. Once this size is reached, the "no-cost" organization is almost near the

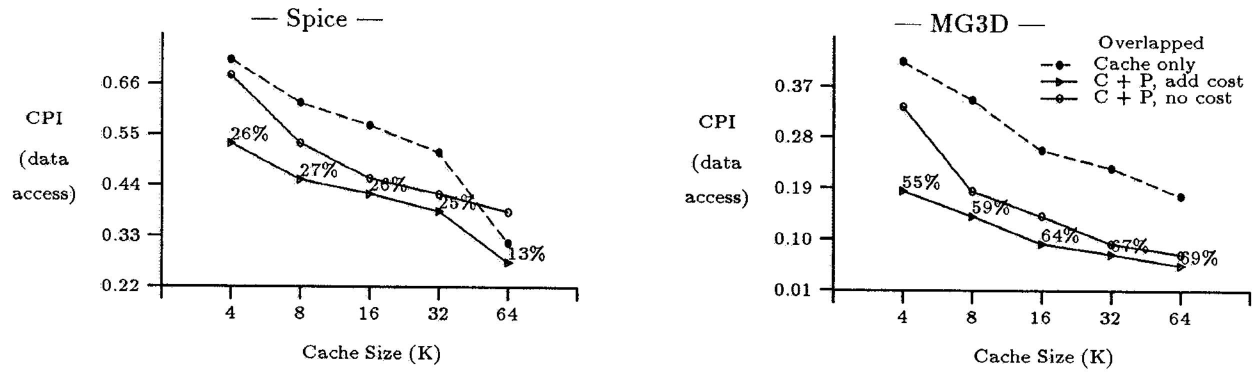

Figure 6: $CPI_{data\ access}$ vs. Cache size for the *Overlapped* model

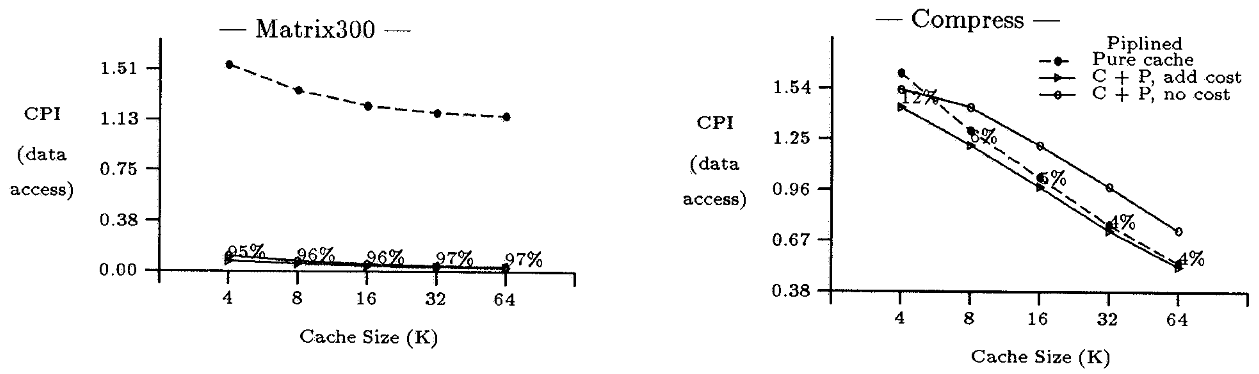

Figure 7: $CPI_{data\ access}$ vs. Cache size for the *Pipelined* model

"add-cost" since most of the predictable cache misses have been removed.

The programs Compress,and Pverify not shown in Figure 5, do not benefit much from preloading. This is not surprising considering their characteristics (see Table 2). Compress relies on a data-dependent table and Pverify contains linked lists data structures and a large number of pointer-based indirect accesses.

5.3 Preloading performance for the *Overlapped* and *Pipelined* models

Figure 6 and Figure 7 show respectively the effects of preloading for the *Overlapped* and *Pipelined* models. Since these models are less bandwidth restrictive than the *Nonoverlapped*, the preloading could take advantage of the additional degree of freedom to preload data blocks. On the other hand, the latency is longer and any incorrect preload will result in a larger penalty.

In Figure 6, we see that both "add-cost" and "no-cost" preloading of the program Spice are slightly more advantageous than in the *Nonoverlapped* case. In the case of MG3D, the effectiveness of the preloading is comparable to that of *Nonoverlapped.*

The *Pipelined* experiments shown in Figure 7 indicate that the relative improvements in program Matrix are similar to those in the *Nonoverlapped* model. However, the absolute reduction in CPI is more than 1 cycle which is quite significant (the absolute reduction is larger since the memory latency is larger). The results show that the preloading can bring into the cache almost all data blocks which could miss and thus eliminate the miss penalty, provided that the communication channel allows it to do so.

The effectiveness (percentage of penalty reduction) of the preloading scheme in the *Pipelined* model for the program Compress is slightly less than that of the *Nonoverlapped* model. This is due to the fact that the longer latency will cause more penalty whenever the preload is incorrect. Furthermore, the larger LA-distance may cause more incorrect branch predictions.

5.4 Preloading performance vs. Look-ahead Limit

Setting the LA-limit d is constrained by two opposite effects. On one hand, when the data misses are clustered and the memory model is restrictive like in the *Nonoverlapped* and *Overlapped* models, a larger d allows earlier issue of preload requests and thus can better take advantage of the limited memory bandwidth to reduce the *hit-wait* cycles. On the other hand, d should not span over too many basic blocks so that the value of LA-PC is not based on many contingent predictions.

Our experiments show that a value of d such as $\delta \leq d \leq 2\delta$ seems correct for the *Nonoverlapped* and *Overlapped* models. As illustrated in Figure 8, when $d < \delta$ (20 cycles in the figure), each access to the preloaded block in progress will be a *hit-wait* access. The contributed *hit-wait* cycles are decreasing as d approaches δ. In the program MG3D and other pro-

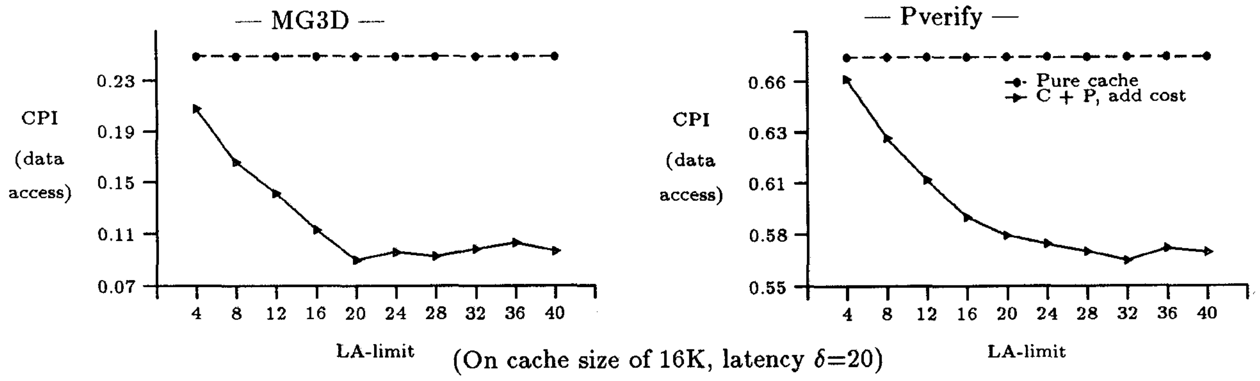

Figure 8: $CPI_{data\ access}$ vs. LA-limit for the *Overlapped* model

grams with low $CPI_{data\ access}$, a local minimum for $CPI_{data\ access}$ happens around $d = \delta = 20$, since the cache misses do not saturate the communication channel. It is not necessary to issue the preload requests earlier. An increase in d will result in a slight $CPI_{data\ access}$ increase because incorrect predictions bring more negative effects. However, for programs such as Pverify with low predictability and high (clustered) miss ratios, the $CPI_{data\ access}$ is not optimal at $d = \delta$, but for a larger value of d.

In general, the more contentious the memory model is, the more sensitive the preloading will be to the setting of the LA-limit d. We conjecture that it would be less interesting to set $d > \delta$ for the *Pipelined* model. From the above discussion, it appears that a good setting of the LA-limit is also a characteristic of the workload. In future work, we will examine the setting of a dynamic LA-limit based on the length of the current basic block.

6 Comparisons with other schemes

6.1 Hardware preloading vs. software prefetching

A potential criticism of our approach is that a compiler could, at no extra hardware cost, capture the *regular* access patterns as well as our hardware scheme, could issue prefetch requests as early as needed and, better yet, would not need to consider the *irregular* patterns at all. However, a compiler approach may have the following disadvantages:

1. Extra *prefetch* or *cache load* instructions may increase the number of instructions to be executed by 20%, since there are between 25% and 35% of instructions referencing memory.

2. The number of *prefetch* instructions can be decreased by clustering them (e.g., using vector load/store instructions[2]). This has two drawbacks when small on-chip caches are the targets. First, there is a higher probability of displacing useful data, and, second, the communication channel is dedicated to a single transfer for a longer period of time.

3. A compiler can move instructions ahead (e.g., load for the $(i + d)^{st}$ iteration while executing the i^{th} iteration) thus removing the drawbacks listed in item 1 but with the dangers of cache pollution if prefetching occurs too early or of contribution to *hit-wait* cycles if it occurs too late.

4. A compiler must perform conservative array subscript analysis for the prefetching instructions. Therefore it will not uncover some of the run-time constant stride or data-dependent patterns that will be detected by the hardware scheme.

It is clear that compiler interaction can help the hardware scheme. Loop unrolling and prevention of preloading of obvious *irregular* access patterns are primary examples.

6.2 Hardware preloading vs Lockup-free Cache/Decoupled Architecture

Conceptually, lockup-free caches, decoupled architectures [13], and our scheme bear the same idea: allow the processor to continue executing while a memory request is being serviced. What make our scheme different is that the preload is a *non-binding* request (it is a "hint" and can be issued earlier than when the data block is really loaded) whereas the load in the lockup-free cache and other architectures is *binding* (a load is generated only when the address of the required data is known).

The LA-distance in our scheme is set so that the data is (optimally) made available "just in time" while the overlap of access time with computation time in the other architectures is subject to the data dependence distance. The latter is usually small when compared with the memory latency. However, our scheme needs to pay the price of the waste of memory bandwidth due to incorrect predictions.

As a matter of fact, the preloading scheme is orthogonal to the lockup-free concept. We could have the *non-binding* preload be issued earlier, use the *binding* load to ensure the correctness without blocking the execution of instructions after the load, and require the machine to block only when an instruction needs to use the data.

7 Conclusions

In this paper we have proposed and evaluated a design for a hardware-based preloading scheme. The goal of this support unit is to reduce the CPI contribution associated with data cache misses. The basic idea is to predict the instruction stream with a look-ahead program counter, LA-PC. When the LA-PC encounters a load/store instruction it predicts the address of the block to be preloaded. Predicted addresses are stored and updated in a cache-like Reference Prediction Table and associated finite-state transition table. Our scheme solves two aspects of a prefetching policy: (1) when to prefetch - at most d (Look Ahead limit) cycles ahead of current execution, (2) which block to prefetch - the most one likely to be referenced - or not to prefetch at all.

We have compared a pure cache baseline architecture, an "add-cost" unit consisting of the original cache and the hardware support unit, and a "no-cost" unit that has part of the cache being replaced by the preloading unit. We have evaluated these organizations using three different memory models with increasing possibilities of overlap. The "add-cost" design works very well in all programs. The "no-cost" design is quite beneficial for scientific applications that exhibit regular data access patterns. Its benefit is moderate on applications that have more irregular data access patterns. Memory bandwidth may slightly limit the advantages of the preloading scheme, but the problem can be alleviated by properly setting the maximum number of cycles allowed between the LA-PC and the regular PC. This scheme could certainly be helped by compiler interaction through mechanisms such as loop unrolling and better prediction.

The introduction of a preloading hardware function in a multiprocessor system poses several questions. Cache coherence is complicated by the preload requests. The way to capture the regular patterns in a uniprocessor system is not necessarily identical to that of the multiprocessor case, since the loop iterations will be spread over several processors. Memory traffic could increase because of incorrect predictions and more invalidations on preloaded data blocks. Our future study on multiprocessor issues will include: (1) the design of a hierarchical memory system in which a *non-binding* preload is used to "hint" at the coherence operations and a *binding* load is used to ensure correctness, (2) the investigation of some scheduling and data allocation techniques to retain the regular data access pattern in each processor, (3) the examination of a more accurate approach to generate preloads and the assessment of the impact of preloading on the memory traffic.

Acknowledgments

We would like to thank Susan Eggers for her helpful comments on an earlier version of this paper and Mike Smith at Stanford for his help with the *pixie* utility. This work was supported by NSF Grants CCR-8904190 and CCR-8702915.

References

[1] A. Borg, R. E. Kessler, and D. W. Wall. Generation and analysis of very long address traces. In *Proc. of the 17th Annual Int. Symp. on Computer Architecture*, pages 270–281, May 1990.

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

[3] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, San Mateo, CA, 1990.

[4] M. D. Hill. *Aspects of Cache Memory and Instruction Buffer Performance.* PhD thesis, University of California, Berkeley, 1987.

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

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

[7] J. K. F. Lee and A. J. Smith. Branch prediction strategies and branch target buffer design. *Computer*, pages 6–22, January 1984.

[8] R. L. Lee, P-C. Yew, and D. H. Lawrie. Data prefetching in shared memory multiprocessors. In *Proc. of the Int. Conf. on Parallel Processing*, pages 28–31, 1987.

[9] C. H. Perleberg and A. J. Smith. Branch target buffer design and optimization. Technical Report UCB/CSD 89/552, University of California, Berkeley, December 1989.

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

[11] S. Przybylski. The performance impact of block sizes and fetch strategies. In *Proc. of the 17th Annual Int. Symp. on Computer Architecture*, pages 160–169, May 1990.

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

[13] J. E. Smith. Decoupled access/execute computer architecture. In *Proc. of the 9th Annual Int. Symp. on Computer Architecture*, pages 112–119, 1982.

[14] The Perfect Club, et al. The Perfect Club benchmarks: Effective performance evaluation of supercomputers. *Int. J. of Supercomputer Applications*, 23(3):5–40, Fall 1989.