IPU/LTB: A Method for Reducing Effective Memory Latency

C. Reid Harmon Jr.
Bill Appelbe
Raja Das

November 24, 1997

Abstract

This paper describes a new hardware approach to data and instruction prefetching for superscalar processors. The key innovation is instruction prefetching by predicting procedural control flow, and decoupling data and instruction prefetching.

Simulation results show this method to recover 72% of unnecessarily lost cache cycles and to yield a great improvement (20-27%) over previous hardware prefetching techniques. The technique has a relatively small cost in hardware, and is intended to come between the processor and a level-1 cache.

1 Introduction

Memory latency is swiftly becoming a serious bottleneck in modern superscalar systems. As the gap between processor speeds and memory access speeds widens, this bottleneck will become even more prominent. The traditional approach to alleviate this problem, hierarchical memories, works well but not optimally; simple hierarchical caches do not capitalize on the fact that the data and instructions which will soon be required are often predictable. Prefetching data and/or instructions into the cache attempts to exploit this opportunity to improve cache hit rates (and thus average access times), but opinions as to the best prefetching approach have varied widely in the literature.

Our approach is to use hardware solely to prefetch both instructions and data, but to do so separately. One hardware unit, the IPU (Instruction Prefetch Unit), prefetches instructions by scanning lines from the I-cache to traverse the program's call graph. It receives a cache line and scans the line for changes in *procedural control flow* (*i.e.* subroutine calls or returns). Then it determines which I-cache line will be needed next, requests it, and repeats this process. In doing this the IPU can keep ahead of the program and minimize I-cache misses.

A separate and independent hardware unit, the LTB (Load Target Buffer), prefetches data by watching sequences of addresses generated per load instruction. It recognizes regular access patterns and requests data which will be loaded in the future. It detects these patterns by analyzing the sequence of address strides. It uses the past three strides to determine whether future strides are predictable and if so, what the sequence of addresses will be. Then it makes a prefetch request to the D-cache so that the data will be available which it is actually requested by the executing program.

The remainder of this paper is divided as follows. Section 2 compares this approach with other prefetching approaches. Section 3 gives more details on implementation

of this method. Section 4 gives experimentation results which answer some design questions. Finally, Section 5 summarizes the results found in this work.

2 Related Work

In software prefetching, the compiler inserts special prefetch instructions into the compiled program, and execution of these instructions triggers prefetching. The difficulty of this approach lies in determining where and when to insert these instructions. Consequently, software prefetching works best for regular dense matrix code[10], where the pattern of data access is known statically. Various compiler algorithms have been studied in [8, 1, 6]. The overhead of additional instructions plus a dependence on sophisticated compile-time analysis limits the effectiveness of this type of prefetching ([12]).

While software prefetching is almost always a form of data prefetching, a form of instruction prefetching in software was proposed by Young, *et. al.* in [14]. In this method, instructions are inserted before branches to advise about prefetching from the branch target. This type of prefetching aids fetching for control flow but does not help the compulsory I-cache misses of straight-line code.

Hardware prefetching approaches are more promising because they do not require the instruction overhead of software prefetching, and because more information about the dynamic nature of the program is known at runtime than at compile time.

The most widely studied hardware prefetching schemes are variations of OBL (one-block lookahead), which use references in cache line n to predict future references in cache line $n + 1$. Such *stream* prefetching approaches can be used for data and/or instruction prefetching, and they have the advantage of being easy and inexpensive to implement in hardware ([11]). However, they make implicit assumptions that regular accesses to memory will be manifested as sequential cache line accesses. If a row-major array is accessed in a column-major order, this will not necessarily be the case. Some software prefetching approaches can prove superior to OBL methods ([9]), although a study in [13] comparing these approaches on two HP machines showed the advantage of a specific OBL approach. By comparison, our data prefetching method is not tied to any particular data access pattern, so long as it is an arithmetic progression rather than a random sequence.

Lopriore ([5]) gives an alternate hardware prefetching strategy to OBL methods. His idea is to reserve cache lines specifically for stack data words. Some optimizations are then possible, as normal program semantics indicate that when the stack shrinks, data previously on the stack are invalid and do not need to be written back to memory. The high locality of stack accesses allow stack data to be prefetched with a high accuracy. The limitation of this approach is that it is limited to stack accesses. Many applications use either large global arrays or dynamically allocated data, and no prefetching is done for these data.

Chen and Baer ([2]) describe a sophisticated alternative to OBL prefetching, referred to as *stride prefetching*. They track load addresses in a *reference prediction table* (RPT) and use a *lookahead program counter* (LA-PC) to detect upcoming loads and thus issue prefetches for these upcoming loads, predicting the next load address. The idea is primarily a data prefetching method, although it is necessary to decode instructions ahead of the normal program counter (PC). Thus, their method performs instruction prefetching as well. Their results were intriguing, although their analysis was limited to what a trace-driven simulation could provide, and then only for a non-superscalar processor. Their simulations were for given values of δ, the difference between LA-PC and PC, but

it appears that the only way LA-PC can get ahead of PC is through fetch stalls caused by execution delays. This problem could be circumvented in a superscalar machine by allowing LA-PC to increment faster than PC (such as by an entire I-cache line), but its ability to get ahead is still hampered by I-cache misses and branch misprediction. Since all data prefetching is done by loads observed in advance via the instruction prefetching, if the LA-PC cannot remain sufficiently far ahead, then the data will not be prefetched sufficiently far in advance. In our method, data prefetching operates independently of instruction prefetching. Therefore, for applications in which instruction prefetching is very difficult (such as xlisp), the data prefetching can still improve performance (see Section 4.4).

Eickemeyer and Vassiliadis ([3]) propose speculative hardware data prefetching; they prefetch data on receipt of a load in the instruction buffer, and provide the value from the predicted address to those computations which use it. When an incorrect address is predicted, the pipeline will need to be flushed and computation resumed from the point at which the load value would have been ready. This scheme depends on the length of the pipeline and instruction buffer to mask the memory latency for cache, and incorrect address prediction will stall the pipeline and cause a flush even if the value from the correct address is in the D-cache. More traditional methods have the advantage that when the predicted address is wrong, only bandwidth has been wasted; the prefetching method supplies data to the cache, but not to the processor. Eickemeyer and Vassiliadis's method uses sophisticated address prediction, which adapts to array accesses well. However, predictions made from irregular accesses will greatly hamper their ability to provide the correct data to the pipeline. This address prediction has been adopted and expanded by our LTB scheme, although like Chen and Baer's method, our approach detects irregular accesses. Eickemeyer and Vassiliadis's method does not because its predicted addresses are always used.

Other cache prefetch techniques are based on markov and correlation-based predictors. Joseph and Grunwald[4] used memory trace simulations to show that a Markov predictor outperformed both stream and stride based prefetching for traces of multiple processes including kernel, user, and shared library activity. Their Markov predictor has a high hardware overhead (over one megabyte for prefetcher data structures), and is intended as an interface between an on-chip L2 cache and an off-chip cache or memory. By contrast, our IPU/LTB prefetcher requires only a small amount of memory for prefetcher data structures (less than one kilobyte), and interfaces directly to the L1 cache and processor. Hence the IPU/LTB data structures could be part of the processor state in a multithreaded processor.

Our simulation studies are based on a micro-architecture simulator and cycle counts rather than just memory reference traces. Miss rates on memory reference traces may be misleading performance measures for superscalar architectures. For example, cycles lost due to data cache misses may be reduced by out-of-order execution.

3 IPU/LTB

3.1 IPU

As stated in Section 1, the IPU is the Instruction Prefetch Unit. It attempts to prefetch instructions by traversing the calling tree of the executing program. It must be able to process more instructions/cycle than the machine's Fetch Unit in order to get ahead, and it has its own bus, the IPUBUS, to the I-cache (see Figure 1).

The IPU relies on the fact that modern superscalar processors invariably can fetch and decode instructions faster than they can be executed. Unlike the IPU, the proces-

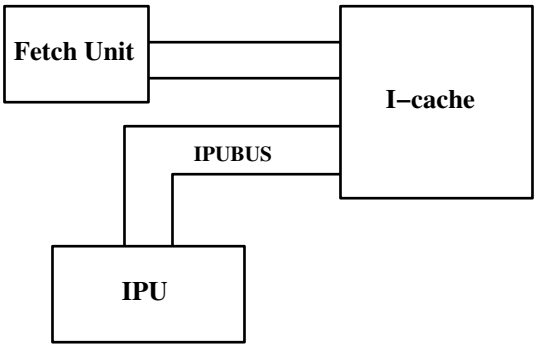

Figure 1: the IPU

sor is limited by data dependences and branch instructions. The IPU ignores branch instructions, except for procedure calls and returns. This means that the IPU can occasionally make a mistake in predicting procedural control flow. However, the IPU has a mechanism to detect and recover from such mistakes.

The I-cache is non-blocking and can handle a request by the Fetch Unit and one by the IPU in a single cycle, although it gives priority to the Fetch request, pulling needed words from memory (or an L2 cache) to satisfy that request before it does so for the IPU's request.

In order to accomplish this, the IPU keeps internally:

- *state* - a 2-bit state (IDLE, RECEIVING, WAIT, STALL)
- *ipPC* - the current prefetch address
- *buffer* - a buffer of prefetched instructions
- *size* - the number of entries in *buffer*
- *stack* - a stack of 3-word entries: *src*, the source of a subroutine call, *dest*, the destination of a subroutine call, and *diff*, the **cumulative** number of words from subroutine starts until calls to the next level. It is likely that an entire stack for *dest* will be ultimately unnecessary, and *diff* is used only to prevent *ipPC* from getting too far ahead (see below).
- *lastagreement* - the highest stack level where the IPU stack agrees with the processors's call stack. This is also used to prevent *ipPC* from getting too far ahead.

Under normal circumstances, the IPU will request enough words from the I-cache to fill *buffer* and then, when it receives part or all of these words, scan them in parallel for a subroutine call or subroutine return. After the request is made, *state* is set to RECEIVING.

If a subroutine call is seen, an entry is pushed on the stack: *src* is the current ipPC, *dest* is the current call destination, and *diff* is the old *diff* plus the current *ipPC* minus the old *dest*. Then *ipPC* becomes the call destination, and *state* becomes IDLE, which triggers a new request at the end of the cycle. The IPU ignores any trailing words from its current request. If more calls are seen in the words received that cycle, only the first is taken.

If a subroutine return is seen, an entry is popped off of the stack, and *ipPC* becomes *src*+1. *state* becomes IDLE, and the IPU will resume prefetching at the end of the cycle.

This is the general approach, although there are a few details which need to be addressed in order to make it feasible:

1. Even with this scheme it is possible for the Fetch Unit to get ahead of the IPU; it may branch over a call to a subroutine which the IPU is exploring. Rather than have the IPU stubbornly continue this way, at the first miss, the I-cache will signal the IPU which will then reset to the current state of the Fetch Unit (set the *ipPC* to *PC*, copy the Fetch Unit's call stack, and adjust *lastagreement*).
2. Since the IPU does not execute instructions, if it were to encounter a recursive function, it would continue to recurse without a terminating condition. Therefore, when the IPU encounters a subroutine call, it checks the destination address against its top 4 destinations in *stack*. Only if there is not a match does it descend into the subroutine.
3. At present the IPU does not attempt to follow indirect function calls, since the target of these calls would not be available. Instead, it ignores such calls. If the Fetch Unit follows an indirect call and subsequently triggers an I-cache miss, the resetting of the IPU will put it back on track again.
4. When, on a function return, popping the return address causes a stack underflow, *state* becomes STALL. The IPU remains in this state until an I-cache miss causes a reset and sets *state* to IDLE.
5. Another potential problem is that of the *ipPC* getting too far ahead of the *PC* and prefetching cache lines which displace those which the Fetch Unit will need. In order to avoid this problem, a value called *ahead* is calculated which represents the distance ahead of the *ipPC*. When the IPU would normally issue a prefetch request, it first compares *ahead* against an internal *threshold*. Only if *ahead* is less than or equal to *threshold* may the prefetch proceed in that cycle.

   The distance *ahead* is calculated by following the subroutine call stack of both the *ipPC* and the *PC*. Figures 2-4 show an example for what *stack*, as well as the Fetch Unit's stack, might look like at one point in execution. In Figure 3, the gray-shaded blocks indicate the instructions that contribute to *ahead*. The reason for not shading the left-end blocks in each range is because the contribution to ahead is the difference between the instructions, not the *inclusive* difference (*i.e.*, $j_2 - j_1 = 3$, so only 3 blocks are shaded). In Figure 4, only the additions to *diff* are shown at each level; the numbers stored are actually cumulative (for simplified calculation). The top *diff* on the IPU stack would therefore be $(ipPC - f_3) + (j_3 - f_2) + (j_2 - f_1)$. The total difference between *PC* and *ipPC* is $(j_2 - j_1) + (PC - f_4) + (j_3 - f_2) + (ipPC - f_3)$. In general, this distance can be calculated as follows:

$$\begin{aligned} ahead \quad = \quad & IPU_{diff}[top] + FU_{diff}[top] - 2IPU_{diff}[lastagreement] - \\ & 2(FU_{src}[lastagreement + 1] - FU_{dest}[lastagreement]). \end{aligned} \tag{1}$$

   A hardware implementation of this metric will be incremental; *ahead* will initially be 0, and events in the machine such as encountering a function call or fetching a number of instructions will cause *ahead* to be updated. Increasing procession of either the *PC* or the *ipPC* inside a function will cause *ahead* to increase, and so will following procedure calls into other functions. Returning from procedure calls (by either the Fetch Unit or the IPU) will decrease *ahead*.

If it is assumed that an entire cache line can be received and dealt with in a single cycle, then *buffer* and *size* do not become necessary, which simplifies the hardware somewhat. Refer to the Appendix for a schematic diagram representing this simpler design.

```
f1() {
   ⋮
   f4();
   ⋮
   f2();
   ⋮
}

f2() {
   ⋮
   f3();
   ⋮
}

f3() {                ipPC
}

f4() {                PC
}
```

Figure 2: Example code

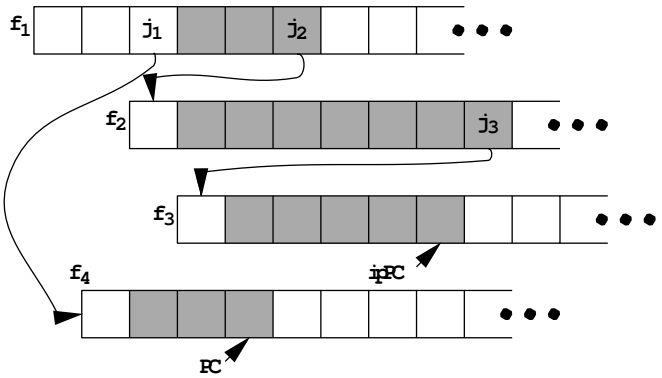

Figure 3: Example execution state

IPU

src	dest	diff
ipPC		+(ipPC-f 3)
j3	f3	+(j 3-f2)
j2	f2	+(j 2-f1)
—	f1	0

Fetch Unit

src	dest	diff
PC		+(PC-f 4)
j1	f4	+(j 1-f1)
—	f1	0

Figure 4: Example stack representations

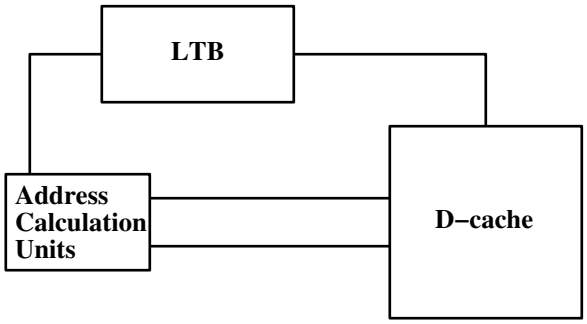

Figure 5: the LTB

3.2 LTB

The LTB is similar in principle to the Branch Target Buffer, tracking a set of recent loads and making predictions about future loads. As addresses are calculated in the address calculation units they are given to the LTB, which creates entries for them if necessary. The LTB may then issue prefetch requests to the D-cache (see Figure 5).

Like branches in the BTB, loads in the LTB are tagged by instruction address. For each load instruction in the LTB, the most recent *load address*, a *count*, and up to three strides are kept. If the sequence of load addresses received for a given instruction are $A_0, A_1, A_2, \ldots, A_n$, then $stride_1$ represents $A_n - A_{n-1}$, $stride_2$ represents $A_{n-1} - A_{n-2}$, and $stride_3$ represents $A_{n-2} - A_{n-3}$.

While *count* is less than a constant K, only the current address is prefetched. (This will actually save a little time, since the load must be checked against pending stores and thus released to the D-cache at a later cycle.) When *count* reaches or exceeds K and if the access pattern is deemed regular, then the Nth address ahead is predicted and prefetched. (N is another constant like K whose specific value is discussed in Section 3.2.3.)

3.2.1 Regular Access Patterns

Regular data accesses produce distinct stride patterns. Take as an example a nest of n loops which are used to access a D-dimensional array, A:

```
for(i1=0;i1<t1;i1++) {
   for(i2=0;i2<t2;i2++) {
      for(i3=0;i3<t3;i3++) {
         (etc.)
            f(A[a1][a2][a3]...);
      }
   }
}
```

The array will be mapped to memory by a scaling the array indexes, as follows:

$$\&A[a_1][a_2][a_3]\cdots[a_D] = A_{base} + a_1 s_1 + a_2 s_2 + a_3 s_3 + \cdots + a_D s_D \tag{2}$$

Or, more succinctly:

$$\&A[a_1][a_2][a_3]\cdots[a_D] = A_{base} + \sum_{j=1}^{D} a_j s_j \tag{3}$$

If the array indexes $\{a_1, a_2, \ldots, a_D\}$ are linear combinations of $\{1, i_1, i_2, \ldots, i_n\}$ as in:

$$a_j = c_{j,0} + c_{j,1} i_1 + c_{j,2} i_2 + \cdots + c_{j,n} i_n, \tag{4}$$

then from Equations 3 and 4:

$$\begin{aligned}
\&A[a_1][a_2][a_3]\cdots[a_D] &= A_{base} + \sum_{j=1}^{D} s_j (c_{j,0} + \sum_{k=1}^{n} c_{j,k} i_k) \qquad (5)\\
&= A_{base} + \sum_{j=1}^{D} s_j c_{j,0} + \sum_{j=1}^{D} s_j \sum_{k=1}^{n} c_{j,k} i_k \\
&= A_{base} + C_0 + \sum_{k=1}^{n} \sum_{j=1}^{D} s_j c_{j,k} i_k \\
&= A_{base} + C_0 + \sum_{k=1}^{n} i_k \sum_{j=1}^{D} s_j c_{j,k} \\
&= A_{base} + C_0 + \sum_{k=1}^{n} i_k C_k \qquad (6)
\end{aligned}$$

where $C_0 = \sum_{j=1}^{D} s_j c_{j,0}$ and $\{C_k = \sum_{j=1}^{D} s_j c_{j,k}\}$ represent new constants.

As the program walks through iterations of the loop nest, the set of iteration variables will change from $\{i_1, i_2, \ldots, i_n\}$ to $\{i'_1, i'_2, \ldots, i'_n\}$, causing the address accessed to change from $A_{base} + C_0 + \sum_{k=1}^{n} i_k C_k$ to $A_{base} + C_0 + \sum_{k=1}^{n} i'_k C_k$, a difference of $\sum_{k=1}^{n} (i'_k - i_k) C_k$.

Usually, i'_n will be $i_n + 1$, and the rest of $\{i'_k\}$ will be $\{i_k\}$. In this case, the difference will be simply C_n. When i_n reaches t_n, it will revert to 0, and usually i_{n-1} will increase by one. This difference, then, is $C_{n-1} - (t_n - 1)C_n$. Less often, both of these indexes will roll over at the same time, and the difference will be $C_{n-2} - (t_{n-1} - 1)C_{n-1} - (t_n - 1)C_n$, *etc.* In short, the stride pattern will be C_n, interrupted every t_n by another value, one which varies.

3.2.2 Predicting the Nth address

Detecting a regular access pattern, then, is nothing more than detecting stride patterns of the form $\ldots, a, a, b, a, a, \ldots$. The Nth address in the future from the current, then, is $A_{current} + aN$. This prediction will be wrong when the next stride is actually b instead of a, but this scheme will stick to predicting a and not be sidetracked by the occasional exception. To make sure that it sees a regular access pattern, it stores the last three strides (in $stride_1$, $stride_2$, and $stride_3$), and requires that two of them consecutively match. The matching value is taken to be a.

This stride prediction works the same on regular array accesses as in [3]. The difference comes when the stride changes for a long stretch (a fairly rare case), or in the case of irregular accesses (an unfortunately common case). This method is able to detect these irregular accesses and not prefetch for them. In addition, if the primary stride does change, this method adapts to it and uses the new value.

Irregular access detection is a feature shared by Chen and Baer's method, although using three strides instead of one stride and two bits of state makes the approach slightly more general and allows the method to recognize the same patterns as Eickemeyer and Vassiliadis's method. Thus, the LTB combines the best features of each of these methods.

3.2.3 Choices for K and N

The best selections for K and N will depend on the nature of the executing code. Selecting too small a value of K may cause prediction without verification of a regular access pattern; selecting too large a value will waste time in prefetching immediately needed lines, when a wiser course would be to look to lines needed in the near future. As the number of total accesses increases much beyond K, however, the specific value chosen for K becomes less important, while the value chosen for N becomes much more important.

The best value of N takes into account the frequency of the load instruction, the approximate IPC being achieved by the machine, and the maximum memory latency. Too small a value of N, and data will not be prefetched early enough; too large a value of N, and the prefetching system will spawn conflict cache misses. Unfortunately, this value will change even as a single application is running. Like *threshold*, a more realistic idea may be to choose a value which tends to work well most of the time.

4 Preliminary Results

4.1 Testing Platform

The results presented in this section were obtained via a machine-level simulation program, SuperDLX, modified from the one developed by McGill University. This simulator is capable of executing compiled DLX assembler files[1] with a variety of machine configurations. It collects cache and prefetch statistics and is able to determine the total latency for instruction fetching and reading and writing data.

The specific configuration used follows the MIPS R10000 processor (refer to [7] for technical details) with the following additions and/or changes:

- combined integer and floating point queues
- separate load and store queues
- a hardware execution stack of 64 entries, allowing the processor to return from functions even before the return address is retrieved from the program stack
- an LTB of 256 entries, organized as 64 4-way associative sets
- no L2 cache, but an L1 cache which is twice as large, requiring a 1 cycle access time to the I-cache and a 2 cycle access time to the D-cache
- a shared 32 byte main memory bus, with a 15 cycle memory access time
- a 32 byte IPUBUS, a 32 byte L1 data bus, and a 16 byte L1 instruction bus

The SPEC95 benchmarks were chosen as representative applications for their variety and availability. Figure 6 lists the programs used with their configurations/inputs.

4.2 Instruction Prefetching

Each of the given applications was executed with the normal configuration above and with instruction prefetching using thresholds of 32 to 2048, in increments of 32. Data prefetching for these experiments was disabled. For each simulation, the *unnecessary fetch cycles total* was calculated. This is the total number of cycles spent fetching instructions *minus* the total minimum access total (the cache access time of 1 cycle times the number of accesses). The prefetching numbers were compared against the

[1] The DLX instruction set is a variation of the MIPS instruction set

Benchmark	Configuration
go	SPEC95 trivial input
compress	SPEC95 trivial input
li	SPEC95 test input (with 4 queens instead of 8)
ijpeg	SPEC95 test input
tomcatv	SPEC95 test input, scaled down
swim	SPEC95 trivial input, scaled up
mgrid	SPEC95 trivial input, scaled up
applu	SPEC95 trivial input, scaled up

Figure 6: Benchmark configurations

original to determine what ratio of unnecessary fetch cycles were recovered through the prefetching. A ratio of 0 indicates no improvement, while a ratio of 1 indicates that the average access time was reduced completely to its minimum.

Figure 7 gives the levels of recovery for the eight applications studied. Five of these perform amazingly well, peaking from 90% to 96%. Out of these, only *applu* begins to suffer from a very high threshold. Both *ijpeg* and *go* peak at a moderately high level, but very high thresholds cause high cache conflict rates, due primarily to the large size of these programs, yielding results worse than for no prefetching scheme. Finally, *li* (xlisp) yields puzzling results. It always performs worse than for no method at all. This can be due to the indirect function calls which xlisp uses, or possibly because the number of levels of recursion used by xlisp is higher than the number checked (4). Studies are underway to determine whether this is the case.

Analysis of the aggregate cycle recovery reveals that the best recovery is yielded by a *threshold* of 448. Even though the maximum is seen at 448, the function is fairly level in this neighborhood, and similar aggregate recoveries are seen in the entire 200-500 domain. See Figure 8 for the fetch cycle recovery rates at a threshold of 448.

4.3 Data Prefetching

In order to determine the best choices for K and N, the eight listed benchmarks were simulated at a *threshold* of 448 for K and N each varying from 0 to 3. The aggregate recovery (across all benchmarks) for combined fetch and read cache losses was calculated for each combination of K and N. This recovery is shown in Figure 9. The best choice for K and N turn out to be 1 and 2, respectively, although for $N > 0$, the specific values have very little effect on the recovery.

The remaining experiments were performed with a *threshold* of 448, a K of 1, and a N of 2. A configuration of this sort will be denoted by a 3-tuple, *e.g.* **(448,1,2)**. Figure 10 shows the separate fetch recovery rates for these applications. The difference between these numbers and those of Figure 8 are that these numbers are the fetch rates with data prefetching turned on as well. Note that this detracts from the effectiveness of the fetch recovery, but only a little.

The data read recovery rates are shown in Figure 11. The data prefetching works above a 50% level, even for *li*.

For perspective, the recovery rates for fetches and data reads are given together in Figure 12. The aggregate recovery across the eight benchmarks is 72%. Since writes do not directly influence the starting time of later instructions, the effect of data prefetching on write completion times is not given here. Preliminary numbers suggest that writes are not affected as much as reads, with two notable exceptions: *li* observes a 50%

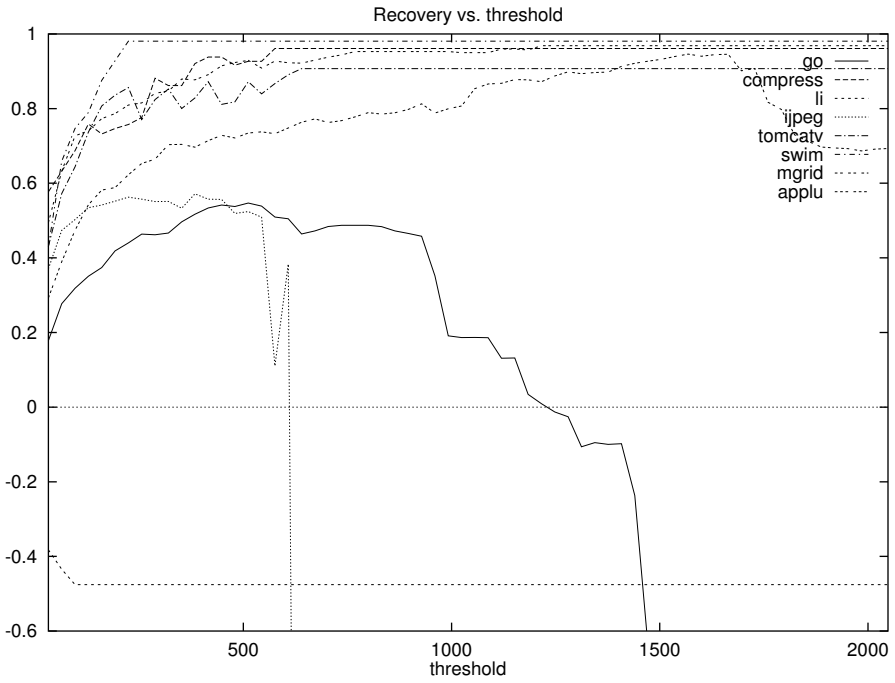

Figure 7: Benchmark fetch recoveries vs. *threshold*

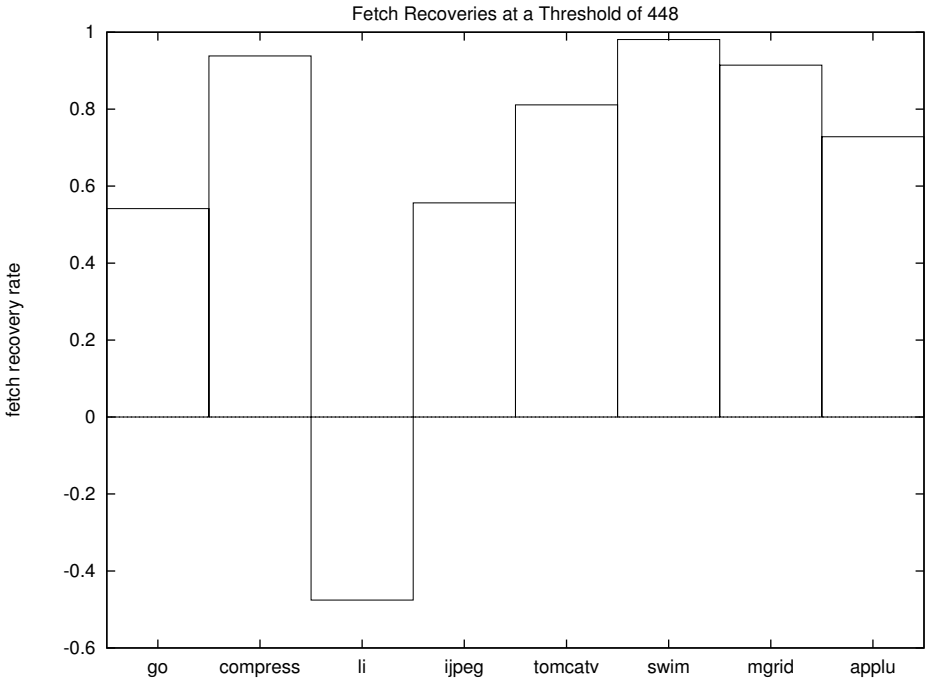

Figure 8: Fetch cycle recovery rates at *threshold*=448

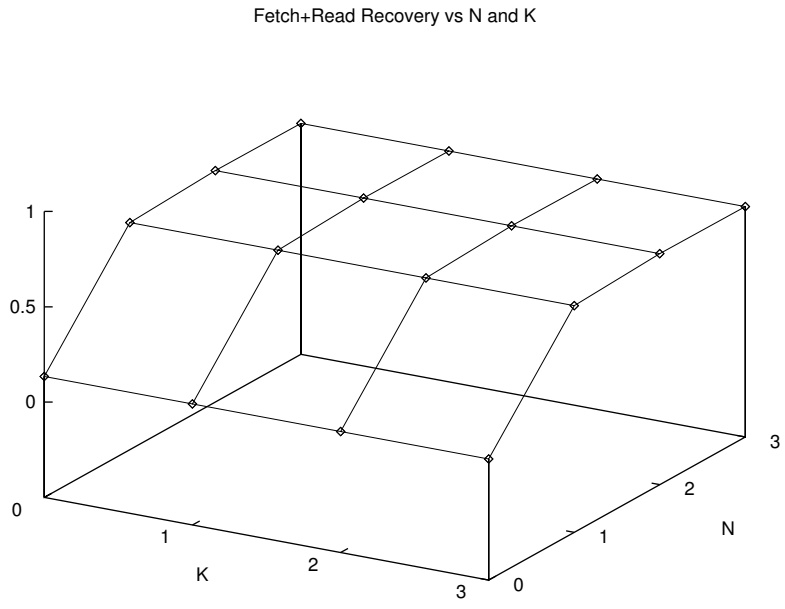

Figure 9: Aggregate recovery versus K and N

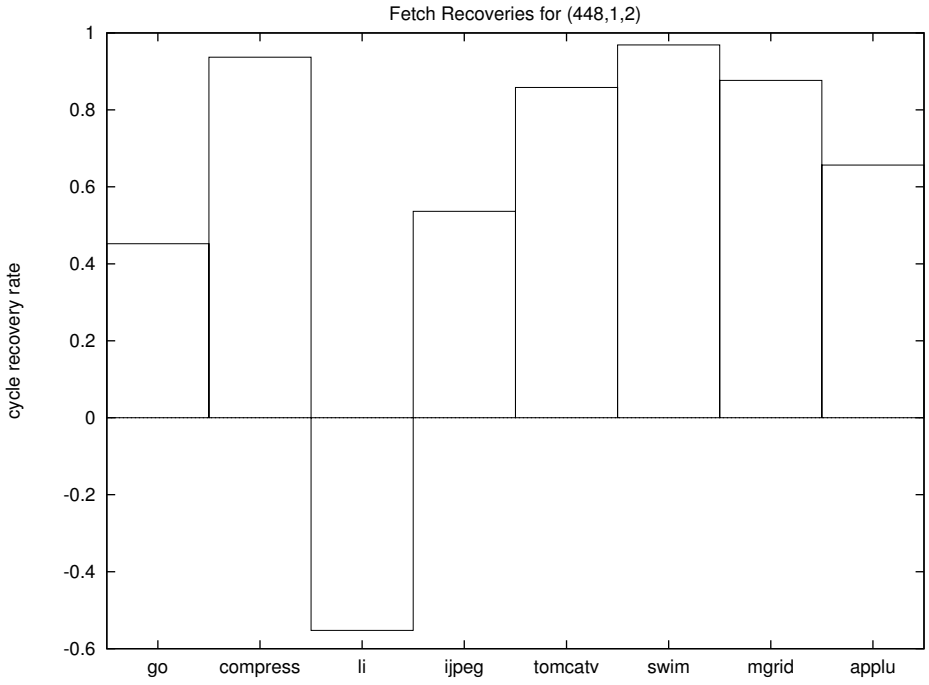

Figure 10: Fetch cycle recovery rates for (448,1,2)

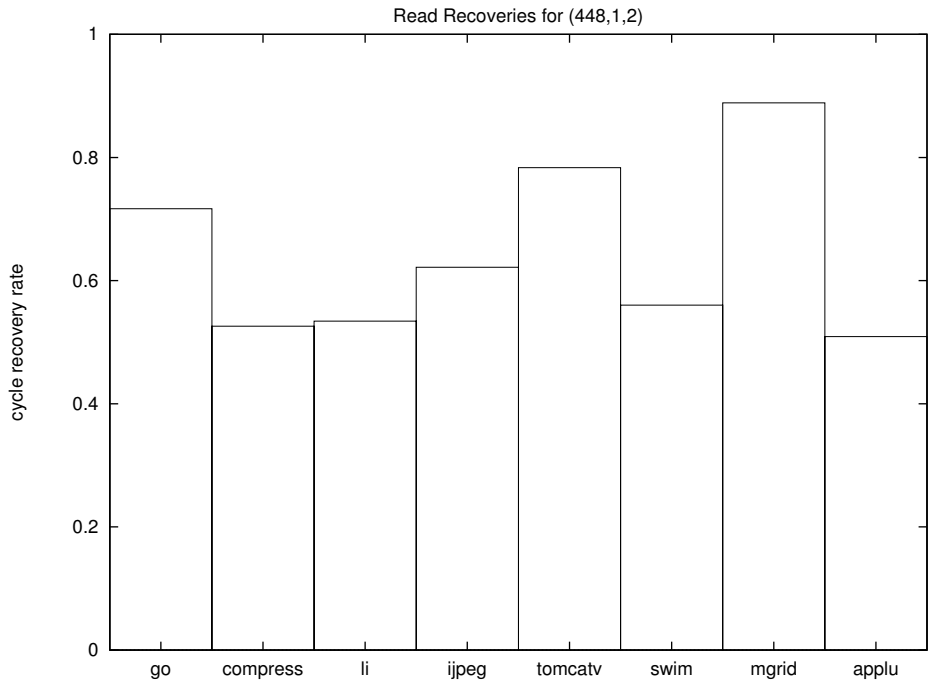

Figure 11: Read cycle recovery rates for (448,1,2)

decrease in unnecessary write completion cycles, while *tomcatv* observes a substantial increase.

In addition, the overall execution time speedups are supplied by Figure 13. Notice that average speedup is about 8%, and that a non-numeric benchmark, *go* shows a 10% speedup. It is important to note that cache losses are partially overlapped with other factors, so even when cache times are greatly improved the gain is not necessarily reflected in the execution time. The benchmark *compress*, for example, has great losses due to cache misses, but these losses tend to overlap a machine dependency on free entries in the load/store queue. Even when the cache times are improved, it takes nearly as long for the program to execute. Also, while we have allowed for a wide path to memory (256 bits), we have also been very conservative in estimating memory latency times (15 cycles). An underestimate of these times will cause an underestimate in initial cache losses. In other words, if *compress* really suffers from much greater losses, then its actual speedup will be much higher.

4.4 Chen and Baer's Method

In order to compare this prefetching method with Chen and Baer's, it was first necessary to extend their method for superscalar architectures. The simulated extended method (which we will refer to as the *EChen* method for simplicity) was capable of receiving an entire 8-word cache line in a single cycle and scan for changes in conditional and procedural control flow. It is able to decide the new target PC and make a request from the I-cache all in a single cycle. The capabilities are the same as for the procedural prefetching method; the difference is that EChen follows branches as well and uses the incoming instructions also to make data prefetches.

Chen and Baer's best performance in [2] utilized a threshold of 30 for non-superscalar

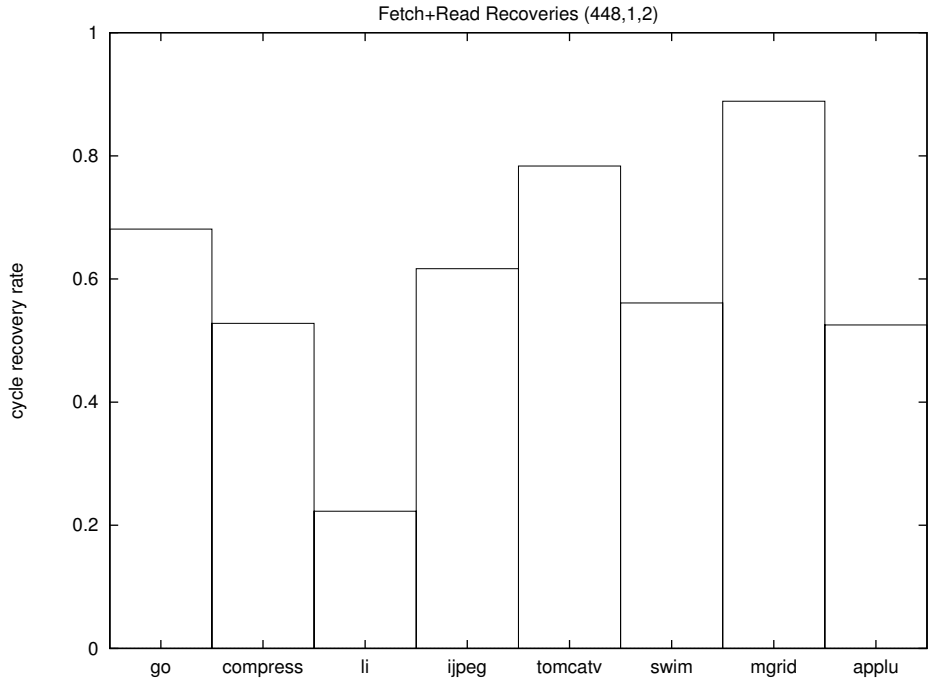

Figure 12: Combined fetch/read cycle recovery rates for (448,1,2)

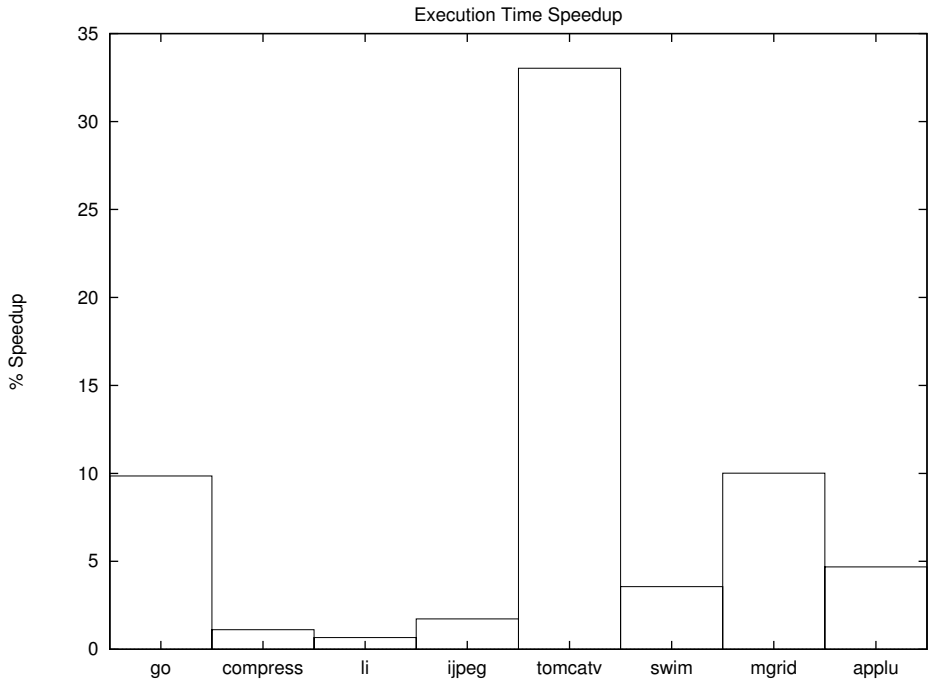

Figure 13: Execution time speedups for (448,1,2)

architectures (this was just a little larger than the memory latency in cycles). To be fair for EChen at a superscalarity of 4, thresholds of 30, 60, 90, and 120 were used. The best value should not scale linearly (*i.e.* all the way to 120), because the average number of instructions processed per cycle will be less than the ideal 4. In fact, since the ILP varies between applications, so will the average number of instructions processed per cycle, and this will cause the best threshold to vary with application.

Also to be fair, comparisons were made against EChen using the same address prediction of the procedural method (this improvement to EChen will be referred to as *EChen+*.) The plots of combined fetch/read cycle recovery rates are shown in Figure 14 compared to our procedural IPU/LTB numbers shown in Figure 12 above. This graph shows that the IPU/LTB approach (represented by the bars with diagonal line patterns) wins over EChen+ in all cases, regardless of the threshold chosen, and the aggregate recovery for IPU/LTB is from 20% to 27% higher than for EChen+. It is clear by these results that even though the scientific benchmarks have simple control flow, the instruction prefetching is limiting the data prefetching. Since the prefetching is separated in IPU/LTB, it does not suffer from this problem.

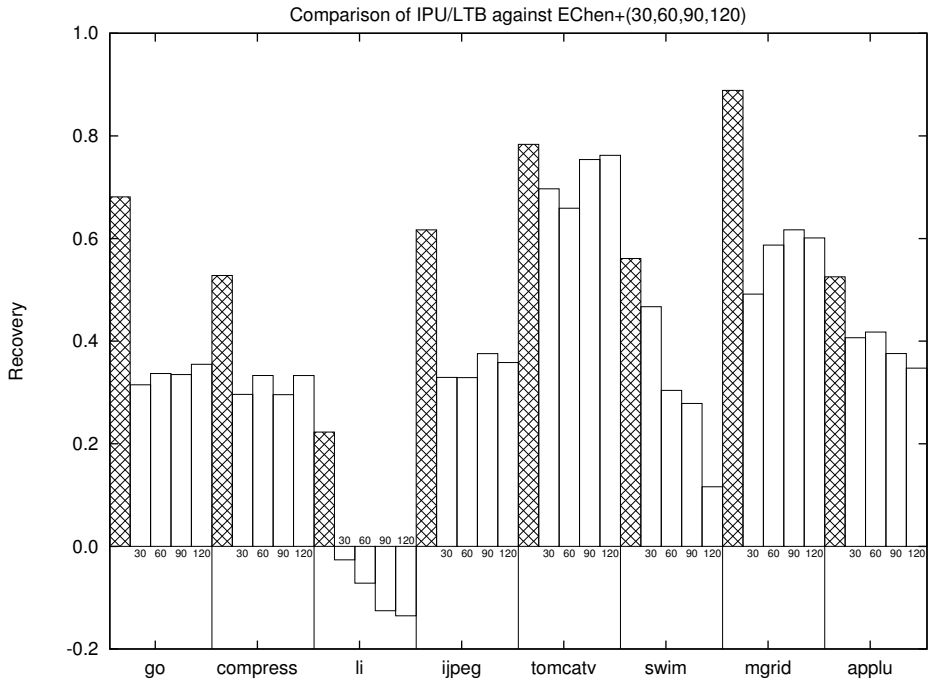

Figure 14: Comparison of EChen+ against IPU/LTB

5 Conclusions

In this paper we have proposed a hardware approach to prefetching which uses a new instruction prefetching method and a data prefetching method which combines the best features of two existing methods. This approach was shown experimentally to yield good results for a set of eight benchmarks, with an aggregate cycle recovery of 72%, a 20-27% improvement over EChen+.

We are currently investigating how crucial the specific system parameters are to the

overall effectiveness of this method. In addition, we are studying how the static and dynamic program size influence this method's behavior, and investigating the results for other benchmarks. Finally, we are looking at ways to reduce the hardware complexity of an IPU/LTB implementation.

Appendix

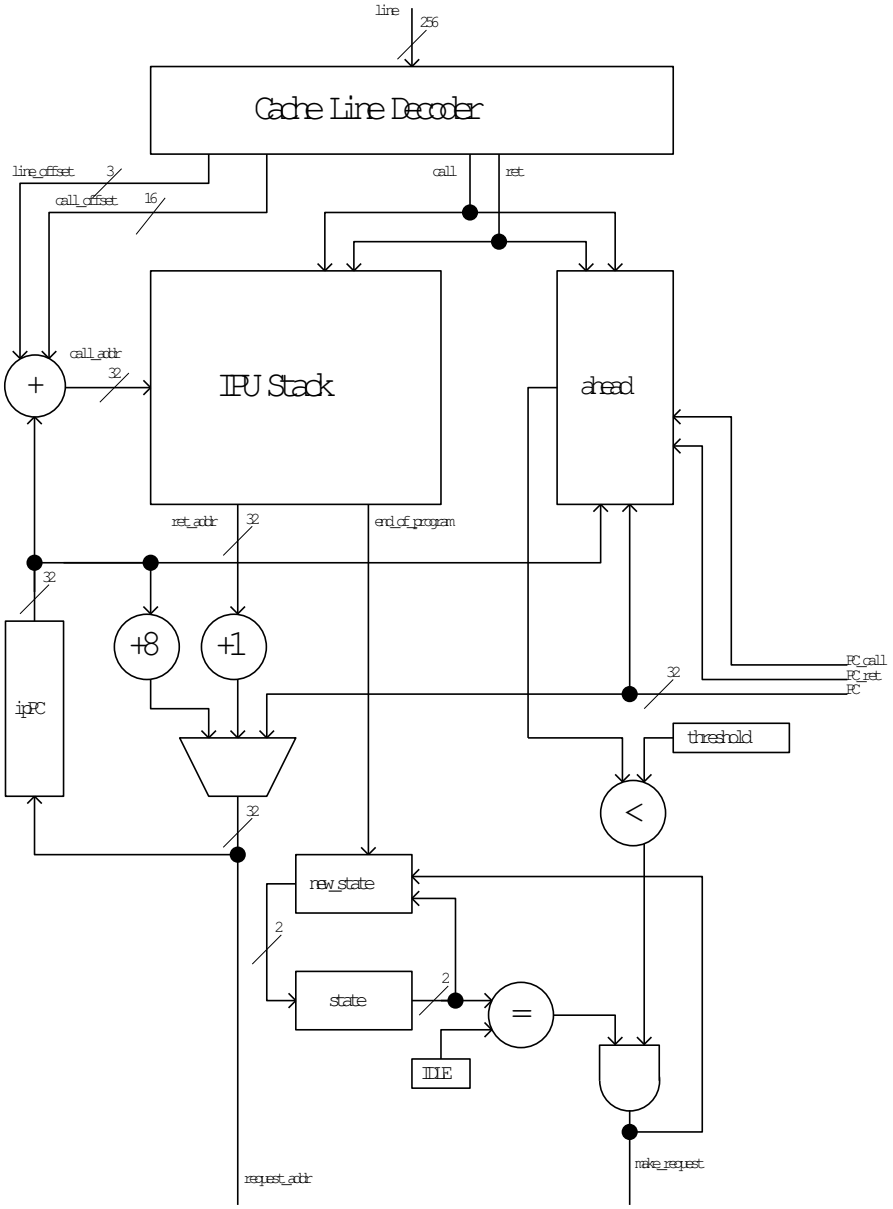

Figure 15: hardware schematic for the IPU

References

[1] D. Bernstein, C. Doron, and A. Freund. Compiler techniques for data prefetching on the PowerPC. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 16–19, 1995.

[2] Tien-Fu Chen and Jean-Loup Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, V44(5), May 1995.

[3] Richard J. Eickemeyer and Stamatis Vassiliadis. A load-instruction unit for pipelined processors. *IBM Journal of Research and Development*, V37:547–564, July 1993.

[4] D. Joseph and D Grunwald. Prefetching using markov predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architectur*, pages 252–263, 1997.

[5] Lanfranco Lopriore. Line fetch/prefetch in a stack cache memory. *Microprocessors and Microsystems*, V17:547–555, November 1993.

[6] C.K. Luk and T.C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, 1996.

[7] MIPS Technologies, Incorporated. *R10000 Microprocessor [Product Overview]*, October 1994.

[8] T.C. Mowry, S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, 1992.

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

[10] E.H. Santhananam, V. Gornish and W-C. Hsu. Data prefetching on the hp pa-8000. In *Proceedings of the 24th Annual International Symposium on Computer Architectur*, pages 264–271, 1997.

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

[12] Dean Tullsen and Susan J. Eggers. Effective cache prefetching on bus-based multiprocessors. *ACM Transactions on Computer Systems*, V13(1):57–88, February 1995.

[13] Steven P. VanderWiel and David J. Lilja. When caches aren't enough: Data prefetching techniques. *IEEE Computer*, pages 23–30, July 1997.

[14] H.C. Young, E.J. Shekita, S. Ong, L. Hu, and W.W. Hsu. On instruction and data prefetch mechanisms. In *International Symposium on VLSI Technology, Systems, and Applications*, pages 239–246, 1995.