Wrong-Path Instruction Prefetching

Jim Pierce[1] and Trevor Mudge

Department of Electrical Engineering and Computer Science
The University of Michigan, Ann Arbor

1. Jim Pierce was supported in this work by a grant from the Intel Corp.

Abstract

Instruction cache misses can severely limit the performance of both superscalar processors and high speed sequential machines. Instruction cache prefetching attempts to prevent misses, or at least reduce their cost, by bringing lines into the instruction cache before they are accessed by the CPU fetch unit. There have been several algorithms proposed to do this, most notably next-line prefetching and table-based target prefetching schemes. A new scheme called wrong-path prefetching is proposed which combines next-line prefetching and target-always prefetching. Surprisingly, a large part of its performance is based upon prefetching the not-taken path of conditional branches. Not only does wrong-path prefetching achieve higher performance than next-line or table-based prefetching schemes, the amount of additional hardware required is roughly the same as next-line and considerable less than table-based implementations. When compared with no prefetching, wrong-path prefetching can reduce the cache miss penalty by as much as 70%. As with all prefetching methods, performance comes at the cost of additional memory traffic. The amount of traffic generated by wrong-path prefetching is similar to that of the other schemes.

1 Introduction

Instruction cache misses are detrimental to the performance of high-speed microprocessors. As the differential between processor cycle time and memory access time grows and the degree of instruction-level parallelism increases in superscalar architectures, the performance degradation of cache misses will become even more apparent. Conventional cache designs will not satisfy the processor's growing demand for instructions. There are several strategies for improving cache performance. The most common method, increasing the cache size and/or its associativity, consumes additional chip area and becomes less effective as caches get larger. Compulsory misses will not be reduced by the even the largest cache. In addition, increasing the cache associativity lengthens the cycle time and could adversely affect the chip's overall cycle time. To improve performance while retaining the small size and speed of a direct-mapped cache, Jouppi proposed adding a small buffer called a victim cache to a conventional direct-mapped cache design to improve performance [4]. Victim caching reduces the number of conflict cache misses by holding onto recently displaced lines. When a conflict miss occurs, the displaced line is stored in the small (1 to 5 entry) fully-associative victim cache. A cache lookup then involves a parallel check in the main cache and the victim cache. If the access hits the victim cache, the lines in the main victim caches are swapped and a conflict miss is avoided.

Cache prefetching is another method to increase cache performance and has been widely studied. Prefetching is an attempt to fetch lines from memory into the cache before their instructions are required by the execution unit. To be effective, the prefetch strategy must accomplish two things. It must be able to guess which cache lines will soon be needed by the fetch unit and it must initiate the prefetch requests long before the instructions are needed so that miss latencies can be reduced or eliminated completely. Theoretically, an optimal prefetch algorithm could remove all cache misses by prefetching all instructions right before they are needed. Unfortunately, non-sequential program flow makes it impossible for the prefetcher to always predict the correct execution direction. Much work has been done to develop methods which anticipate the direction of program flow and to prefetch instructions in this direction. This paper proposes a new prefetching algorithm which makes no attempt to predict the correct direction. In fact, it relies heavily on prefetching the wrong direction. Not only does this method outperform previously proposed prefetching schemes, but it does so at a lower hardware cost.

The word prefetch can be found in several different contexts in current computer literature and it is important we clarify exactly what we mean by cache prefetching and what problem this paper intends to address. We are studying cache prefetch algorithms which reduce instruction cache misses by prefetching instruction lines from memory into the cache. A source of confusion is that the term prefetch is also used to denote the act of fetching multiple words from the cache into the fetch unit of the execution pipeline. The goal of cache prefetching is to reduce cache misses. The goal of what we will call instruction prefetching is to assist in instruction decode or to increase the instruction issue rate. The Intel Architecture processors (i486 and Pentium) utilizes cache-to-buffer prefetching to alleviate the decode problems associated with variable instruction size and complex encodings [1]. Superscalar processors like the Alpha or Power2 architectures prefetch multiple lines from the cache so that multiple instructions can be issued per cycle even during branch conditions [7][13]. The PowerPC also prefetches multiple instructions from the cache into prefetch buffers. It does this primarily because the instruction fetch must share a single

port to the unified cache with data memory requests and thus it cannot fetch an instruction from the cache every cycle. What is important to note here is that these four processors (and others like them) perform instruction prefetching and not cache prefetching. In all the above examples, instruction prefetching never initiates requests to memory. Instruction prefetching stops if the correct lines are not found in the cache. Finally, another use of the term prefetching applies to data prefetching. Data prefetching attempts to reduce data cache misses by exploiting a program's data access patterns in order to prefetch data from memory [2][5]. This paper does not address data prefetching issues.

2 Instruction Prefetching

Instruction prefetching can be done passively by modifying the cache organization to promote prefetching or by including additional hardware mechanisms to execute an explicit prefetching algorithm.

Long Cache Lines

The simplest form of prefetching is the use of long cache lines [11]. When a line is replaced, new instructions are brought into the cache in advance of their use by the CPU, thereby reducing or eliminating miss delays. The disadvantages are that longer lines take longer to fill, they increase memory traffic, and they contribute to cache pollution due to the larger replacement granularity. A long instruction line which is only partially accessed will displace many existing instruction words which may be needed in the future.

Next-Line Prefetching

Another approach to instruction prefetching is next-line prefetching. It tries to prefetch sequential cache lines before they are needed by the CPU's fetch unit. In this scheme, the current cache line is defined as the line containing the instruction currently being fetched by the CPU. The next line is the cache line located sequentially after the current line. Next-line prefetching works in the following manner. If the next line is not resident in the cache, it will be prefetched when an instruction located some distance into the current line is accessed. This specified distance is measured from the end of the cache line and is called the fetchahead distance, see Figure 1. Next-line prefetching predicts that execution will "fall-through" any conditional branches in the current line and continue along the sequential path. The scheme requires little additional hardware since the next line address is easily found. Unfortunately, next-line prefetching is unlikely to reduce misses when execution proceeds down non-sequential execution paths caused by conditional branches, jumps, and subroutine calls. In these cases, the next line guess will be incorrect and the correct execution path will not be prefetched. Performance of the scheme is dependent upon the choice of fetchahead distance. If the fetchahead distance is large, the prefetch is initiated early and the next line is likely to have been received from memory in time for CPU fetch. However, increasing the fetchahead distance increases the probability that a branch will be encountered and execution will continue in a non-sequential direction rendering the next-line prefetch ineffectual. This useless prefetch increases both memory traffic and cache pollution. In spite of these shortcomings, next-

line prefetching has been shown to be an effective strategy, sometimes reducing cache misses by 20-50% [3].

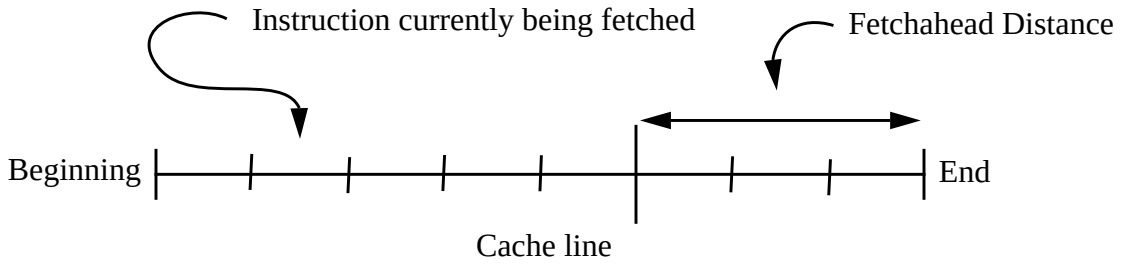

FIGURE 1. In next-line prefetching, once instruction fetch occurs within the fetchahead distance. the next consecutive cache line will be prefetched.

Target-Line Prefetching

Target-line prefetching addresses next-line prefetching's inability to correctly prefetch non-sequential cache lines. When instructions in the current line are being executed, the next cache line accessed might be the next sequential cache line or it might be a line containing the target of a control instruction found in the current line. Since unconditional jump and subroutine call instructions have a fixed target and conditional branch instructions are often resolved in the same direction as they were when last executed, a good heuristic is to base the prefetch on the previous behavior of the current line. Target-line prefetching uses a target prefetch table maintained in hardware to supply the address of the next line to prefetch when the current line is accessed. The table contains current line and successor line pairs. When instruction execution transfers from one cache line to another line, two things happen in the prefetch table. The successor entry of the previous line is updated to be the address of new current line. Also, a lookup is done in the table to find the successor line of the new line. If a successor line entry exists in the table and that line does not currently reside in the cache, the line is prefetched from memory. By using this scheme, instruction cache misses will be avoided or at least their miss penalty will be reduced, if the execution flow follows the path of the previous execution.

Hybrid Schemes

A hybrid scheme which combines both next-line and target prefetching was proposed in [3]. In this scheme, both a target line and next line can be prefetched, offering double protection against a cache line miss. Next-line prefetching works as previously described. Target-line prefetching is similar to that above except that if the successor line is the next sequential line, it is not added to the target table. This saves table space thus enabling the table to hold more non-sequential successor lines. The results are impressive: miss rates are reduced by a factor of 2 or 3. In addition, the results in [3] show that the performance gain of the hybrid method is roughly the sum of the gains achieved by implementing next-line and target prefetching separately.

Performing target prefetching with the help of a prefetch target table is not without disadvantages, however. First, significant hardware is required for the table and the associated logic which performs the table lookups and updates. This uses additional chip area and could increase cycle time. Second, the extra hardware has only limited benefit. Table-based target prefetching does not help first-time accessed code since the table first needs to be set up with the proper links or current-successor pairs. Thus compulsory misses are unaffected by target prefetching. Furthermore, unlike a branch prediction table, even when the correct information does exist in the table it cannot always be utilized. Upon re-execution of the code when the links are properly set, prefetching will only occur if the target line has been previously displaced from the cache. In the likely event that the line is still in the cache, the table entry space and lookup are wasted because prefetching is not needed. This suggests that target prefetching using a table is best suited for small caches with low associativity where lines are often displaced and then rereferenced. This was the proposed application environment in [3].

It is interesting to note several points common to the above schemes. One is that prefetch decisions are made at the cache line level. No instruction-specific information is used. This makes sense because a prefetch decision must be made early and several cycles may pass before instruction recognition can take place in the decode stage of the pipeline. Another point is that the above schemes try to predict the correct execution path and then prefetch only down the predicted path. For instance, using a small fetchahead distance will bias the next-line prefetching scheme toward the correct path by lowering the probability of a control instruction being within the fetchahead distance. Target prefetching predicts that the correct direction in which to prefetch is the direction of the previous execution. Even though the hybrid algorithm may prefetch lines down the wrong path since it sometimes prefetches both a next line and a target line for the current line, such actions are unintentional and rarely occur. Prefetching the correct path satisfies intuition since only lines soon to be executed should be prefetched. The alternative, fetching wrong path lines into the cache, will likely increase memory traffic and cause cache pollution.

3 Wrong-Path Prefetching

A recent study of ours showed that the intuition expressed above is partially false [9]. The work primarily studied the effects of speculative execution on data cache performance. In particular, it focused on how data brought into the cache during speculative execution down what later turned out to be a mispredicted path affected the cache miss rate and memory traffic. The results showed that memory traffic increased due to additional speculatively executed instructions, as might be expected. However, the data cache miss ratio did not increase significantly. In fact, the number of data misses generated during the execution of correctly predicted paths decreased. In other words, the mispredicted or wrong path data references acted as data prefetches for later correctly predicted path data accesses. In addition to data cache studies, the work also revealed that on the benchmarks tested, over 50% of instructions accessed on mispredicted paths were later accessed during correct path execution. This suggests that prefetching down wrong paths may have some advantages and led to the development of a new prefetching strategy.

We propose a new algorithm called *wrong-path prefetching* which is similar to the hybrid scheme in the sense that it combines both target and next-line prefetching. The next line is prefetched whenever instructions are accessed inside the fetchahead distance as described earlier.

The major difference is in target prefetching. No target line addresses are saved and no attempt is made to prefetch only the correct execution path. Instead, in the simplest wrong path scheme, the line containing the target of a conditional branch is prefetched immediately after the branch instruction is recognized in the decode stage. Thus, both paths of conditional branches are always prefetched: the fall-through direction with next-line prefetching, and the target path with target prefetching. Unfortunately, because the target is computed at such a late stage, prefetching the target line when the branch is taken is unproductive. A cache miss and a prefetch request would be generated at the same time. Similarly, unconditional jump and subroutine call targets are not prefetched since the target is always taken and the target address is produced too late. The target prefetching part of the algorithm can only perform a potentially useful prefetch for a branch which is not taken. If execution returns to the branch in the near future, and the branch is then taken, because of the prefetch the target line will probably reside in the cache.

The obvious advantage of wrong-path prefetching over the hybrid algorithm is that no extra hardware is required above that needed by next-line prefetching. All branch targets are prefetched without regard to predicted direction and the existing instruction decoder computes the address of the target. There are several reasons to believe that the performance of wrong path prefetching might also compare favorably with other schemes. Wrong-path prefetching can prefetch target paths which have yet to be executed unlike the table-based schemes which require a first execution pass to create the cache line links. In addition, wrong-path prefetching should perform better than correct-path only schemes when there exists a large disparity between the CPU cycle time and the memory speed. This is because other algorithms try to prefetch down target paths which will be executed almost immediately, and if memory is slow the prefetch may not be initiated soon enough. On the other hand, wrong-path prefetching prefetches lines down a path which is not immediately taken thus it has more time to prefetch the line from a slow memory before the path is executed. However, the performance of wrong-path prefetching does not come without cost. Unavoidably, prefetching down not-taken paths will put lines into the cache that are never accessed. This will increase both memory traffic and cache pollution. For the algorithm to be successful, the benefits of prefetching must overcome the added pollution misses. The extra traffic cannot be reduced, but memory bandwidth can be viewed as a hardware resource to be utilized to reduce the performance degradation caused by instruction cache misses.

Again, it should be emphasized that wrong-path prefetching is fundamentally different from the both path instruction prefetching done in some current superscalar processor designs. In these architectures, words from both paths are copied from the cache to the prefetch buffer. After one of the paths is executed, the wrong path words are removed from the buffer. Instruction prefetching never causes a memory-to-cache transfer so the number of cache misses will not be affected. In the proposed wrong-path prefetching scheme, lines containing instructions from not-taken paths are routinely fetched from memory and stay resident in the cache.

One way to improve the performance of the wrong-path prefetching scheme would be to initiate target prefetches earlier. This would then remove misses or reduce miss latencies on target paths which are taken. One way to do this would be to add a prefetch buffer which has the ability to partially decode the instructions as they are stored there. When the buffer is filled, all control instructions would immediately be detected and the target addresses would be queued for possible prefetching. This would be beneficial in several ways. First, prefetches of taken branch paths

would be initiated several cycles before the target instruction is actually fetched by the CPU. Second, prefetching the targets of jump and subroutine call target would beneficial for the same reason as taken branches. Thus, the earlier prefetch initiation will at least reduce miss latencies caused by misses on taken target paths. Adding the hardware to detect control instructions in the prefetch buffer allows wrong path prefetching to begin at most one cycle later than the table-based target prefetching's optimal case. Furthermore, wrong-path prefetching can prefetch multiple targets found in the same cache line which almost guarantees that the successor line will be cache resident or prefetched. Adding a prefetch buffer with partial decode would only be practical with fixed-length instructions using simple encoding and target address generation schemes. To reduce the hardware complexity, the number of instructions decoded in the buffer could be limited. For instance, only instructions in the last half of the buffer might be examined for control types.

4 Preliminary Experiments

Since the wrong-path prefetching algorithm relies on prefetching target lines that are not taken, we first performed experiments which isolated the effects of prefetching lines only down not-taken paths. The experiments were done on an i486 SysVR4 Unix platform using the SPEC benchmark `gcc` as the workload. Traces were generated using `IDtrace` and then fed into a prefetch/multi-cache simulator. `IDtrace` is a binary instrumentation tool for the Intel architecture [8]. The prefetch simulator was programmed to prefetch lines from only not-taken paths of conditional branches. Therefore, a not-taken branch causes the cache line containing the fall-through address to be prefetched. A taken branch causes the target line to be prefetched. No other lines are prefetched. Figure 2 compares the number of cache misses when no prefetching is performed with the miss performance of 4 variations of the described wrong-path-only prefetching algorithm. The WPO-1, 2, and 3 algorithms represent prefetching 1, 2, and 3 consecutive wrong path lines respectively. For instance, the WPO-2 algorithm would prefetch two cache lines down the not-taken direction of every conditional branch. The profile algorithm, PROF-1, will be explained later. It can be seen that prefetching only the lines from paths not immediately executed exhibits surprisingly good results. The prefetching effect far outweighs the extra pollution generated by sometimes prefetching unused cache lines.

One problem with this prefetching approach is the large amount of extra traffic generated, as shown in Figure 3. A way to reduce this traffic would be to eliminate prefetches of paths which are never taken. Prefetching these paths cause extra traffic and contribute to cache pollution. Thus wrong-path prefetching should not occur for conditional branches which are always taken or always not-taken. To examine the effects of removing these wasted prefetches, `gcc` was profiled to create a list of the conditional branches in which both the target and fall-through paths are taken sometime during execution. The program was reexecuted using the Prof-1 prefetch algorithm. Prof-1 uses the profile data to decide when to prefetch. It prefetches a cache line from the not-taken path of a conditional branch only if the branch is in the profile list, i.e., if both directions of the branch are taken sometime during program execution. This reduces the number of prefetches since some branch paths are never taken but should not degrade the overall performance since the removed prefetches prefetched only non-taken paths. In fact, Prof-1 should have better performance than WPO-1 because of a reduction in cache pollution. Surprisingly, comparing the WPO-1 and Prof-1 results in Figure 2 and Figure 3 shows that the expected traffic reduction was accompanied by an unexpected increase is cache misses. The unanticipated poor performance has two

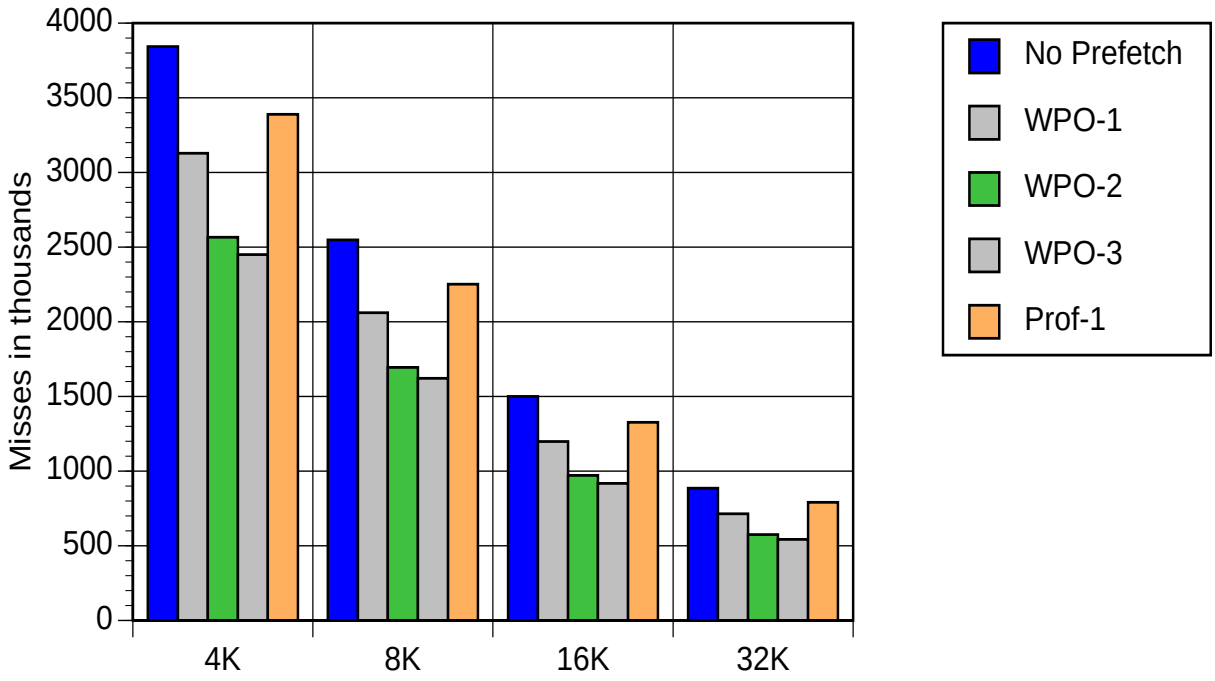

FIGURE 2. Instruction misses resulting from prefetching lines down only the mispredicted direction of conditional branches. The benchmark was gcc run to completion on an i486 Unix machine. The cache was direct mapped with a line size of 32 bytes.

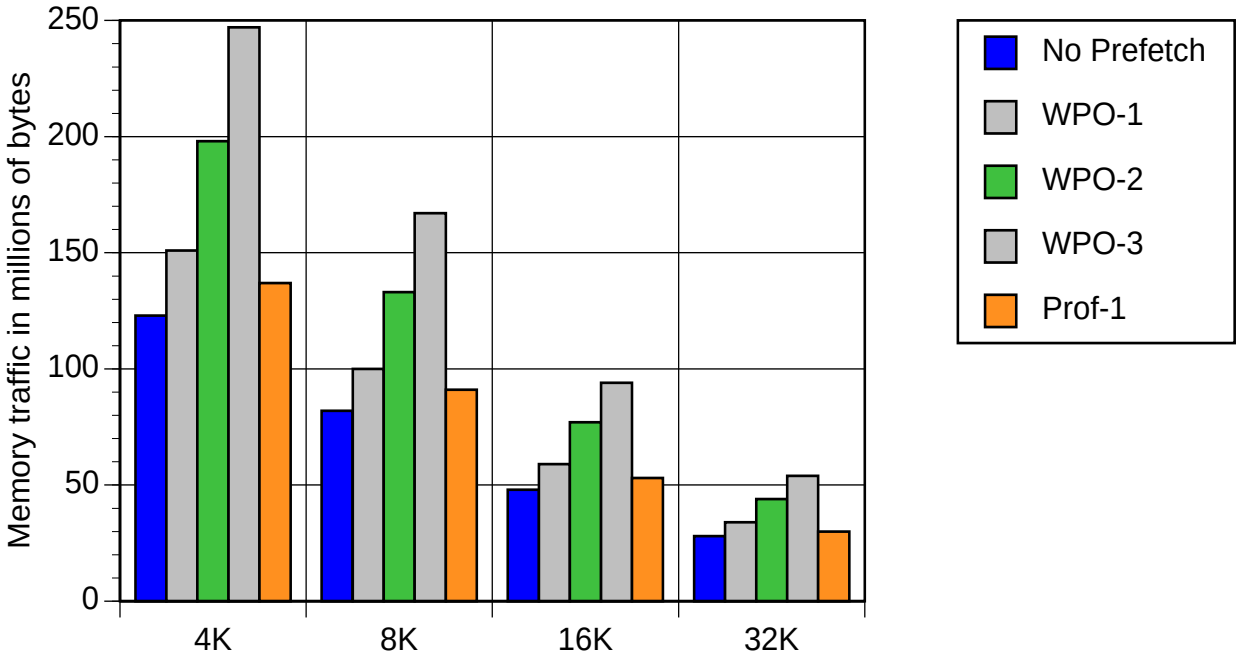

FIGURE 3. Traffic generated by prefetching lines down only the mispredicted direction of conditional branches. The benchmark was gcc run to completion on an i486 Unix machine.

possible explanations. One is that the not-taken cache line contains more than just the not-taken path instructions. The relatively long 32-byte line contains other paths which are soon to be accessed. The other reason is that multiple branches can have the same target and different branches can prefetch for each other. For instance, say that branch A and branch B have the same target address but branch A never executes the target path and branch B does. The situation could occur in which prefetching the wrong-path of branch A, the target path, prevents a miss from occurring during the later execution of branch B's target path. In these ways, prefetching never executed branch directions has a positive effect on cache performance.

These preliminary experiments show that prefetching down not-taken paths can significantly reduce cache misses at the cost of higher memory traffic. Methods which attempt to select which branches to prefetch reduce the memory traffic but can also impair the algorithm's ability to reduce cache misses.

5 Algorithm Comparisons

5.1 Benchmarks and Tools

Experiments to evaluate prefetching algorithms were done using two different architectures: an Intel i486 SysVR4 Unix system and a DECstation 5000 with a MIPS R3000 processor. The relative results were similar on each architecture and for brevity, only the MIPS results are shown here. Traces were gathered using `IDtrace` on the i486 and `pixie` on the DECstation [8][12]. The benchmarks used are listed in Table 1. They currently consist of several C integer SPEC benchmarks. The benchmarks were all run to completion.

The cache simulator allows variation of many cache parameters. We studied a range of cache configurations including line sizes of 8, 16, 32, and 64 bytes, set associativities of 1, 2, 4, and 8 way, set sizes of 128, 256, 512, and 1024, and fetchahead distances of 1/4, 1/2, and 3/4 of the line size. A representative subset of these results are used in the illustrations in this paper.

5.2 Hardware Considerations

A trace driven prefetch simulator models the performance of various algorithms observing somewhat conservative hardware restrictions. An attempt was to make an accurate comparison by basing the simulator on realistic and achievable hardware configurations. The simulator limits the number of memory references per cycle, the number of tag lookups possible per cycle, and con-

strains the ordering of prefetch requests. One hardware feature utilized in this study but not yet

Program	Description	Number of MIPS Instructions (million)	Miss Index (misses per thous. instr.)	Traffic Index (bytes per instr.)
gcc	Gnu C compiler	83	74	2.4
sc	spreadsheet program	1438	19	0.6
xlisp	XLISP interpreter solving 8 queens problem	1028	35	1.2

TABLE 1. The benchmark set used our experiments. The last two columns give a measure of the instruction cache load caused by the benchmark. The number are found using a 8K, 32-byte line, direct-mapped cache.

found in general purpose microprocessors is a non-blocking instruction cache [2][6]. This allows a memory request to be issued (but not completed) in each clock cycle. While this is not a necessity for instruction prefetching, it allows a great deal of independence between the prefetch and the fetch units. Other hardware and algorithm assumptions include:

- Two sets of cache tags. One is for regular instruction fetches and the other is for prefetch checking. This enables instruction fetching and cache line prefetching to occur simultaneously.
- A limit of only one memory access per cycle. Depending upon the algorithm, multiple prefetch requests might be generated per cycle and the CPU's fetch unit might also issue a request due to a cache miss. Requests not granted permission during a cycle are queued for subsequent cycles. A cache miss takes precedence over any other request. Currently, the order of precedence of the remaining requests is target prefetch, queued prefetch, and then next-line prefetch.
- If multiple prefetch requests are generated, only one request can check the tags for cache residency. Using the same precedence order, one (target or queued request) can check the tags, the other (next-line request) is queued. Thus the queue may contain prefetch requests for lines already resident in the cache.
- A queued prefetch ready to be initiated checks the cache tags before it initiates a memory request since the cache may have changed since the request was queued and not all queued prefetches are valid.

We expect that the performance of the prefetch algorithms could be improved by issuing multiple memory requests through memory interleaving or by performing multiple cache tag checks with higher tag replication. However, our conservative study puts all algorithms on an equal footing for comparison and generates results which are achievable with today's processor technologies.

5.3 Performance Measurement

Most cache studies use the number of cache misses as their performance measure. Miss reduction alone is not a sufficient measure of the performance of cache prefetching algorithms because it does not account for differences in prefetch initiation times. For instance, suppose the

miss latency for a processor is 5 cycles and one algorithm initiates a prefetch 3 cycles before the fetch while the other only 1 cycle before the fetch. An instruction miss will occur in both cases but the first prefetch algorithm will have reduced the miss penalty by more cycles thus giving better performance. To account for both misses and timing differences in prefetch algorithms we define a new measure called the *penalty index*. It can be thought of as an approximation of the number of cycles wasted due to cache misses assuming that no instruction reordering or other work is done during these cycles to hide the delay cost. If an instruction miss occurs, we assume the memory system has the ability to return the missed instruction word first so that *ML*, the miss latency, is the miss delay. If, however, the first word in the cache line is the first word returned from memory, our measurements will be a lower bound for the delay caused by instruction misses. With no prefetching, the penalty index for some program and cache configuration is just

$$\text{Penalty Index } (ML) = (\#\text{ of misses}) \times ML$$

Thus, the penalty index is just the number of delay cycles caused by instruction cache misses. To measure the wasted cycles when using a prefetching algorithm, the time between a line prefetch and the first access of the line must be calculated. To do this, the simulator computes the prefetch distance, which we define as the number of cycles between the prefetch and the first access of the prefetched line. If the prefetch distance is greater than the memory latency, the prefetch is perfect and there are no wasted cycles. On the other hand, if the prefetch distance is *n*, where *n* is less than the latency, then (*ML - n)* cycles are wasted. The following equation computes the penalty index using a prefetch algorithm where PD_i is the number of prefetches with prefetch distance *i*.

$$\text{Penalty Index } (ML) = \sum_{i=1}^{ML} PD_i \times (ML - i)$$

To arrive at one penalty index number for the whole benchmark suite, the penalty index for each benchmark was normalized by dividing by the number of instructions in the benchmark. The normalized penalty index for the suite is then the average over all benchmarks.

5.4 Results

Figure 4 compares the penalty index of the different algorithms described in the previous sections. WP-1 is a wrong-path algorithm which prefetches 1 target line of each conditional branch while the WP-2 algorithm initiates the prefetch of the target line in one cycle and the next cache line following the target line in the next cycle. Similarly, WP-3 prefetches the target line and the next two lines following the target line. The hybrid scheme always uses a direct-mapped, 32 entry target buffer except where noted. Fetchahead distances were 3/4 of the line size except where noted. The wrong-path algorithms achieved the highest performance all the algorithms studied. The next-line and the hybrid schemes reduced the penalty index by as much as 51% compared to no prefetching. WP-1 reduced the penalty by 61% and WP-2 performed the best, reducing the penalty index by as much as 64%. The results of using WP-3 shows that it is possible to get carried away with prefetching consecutive lines. The pollution costs are too high and the performance is degraded after prefetching 2 consecutive target lines. We compared the algorithms

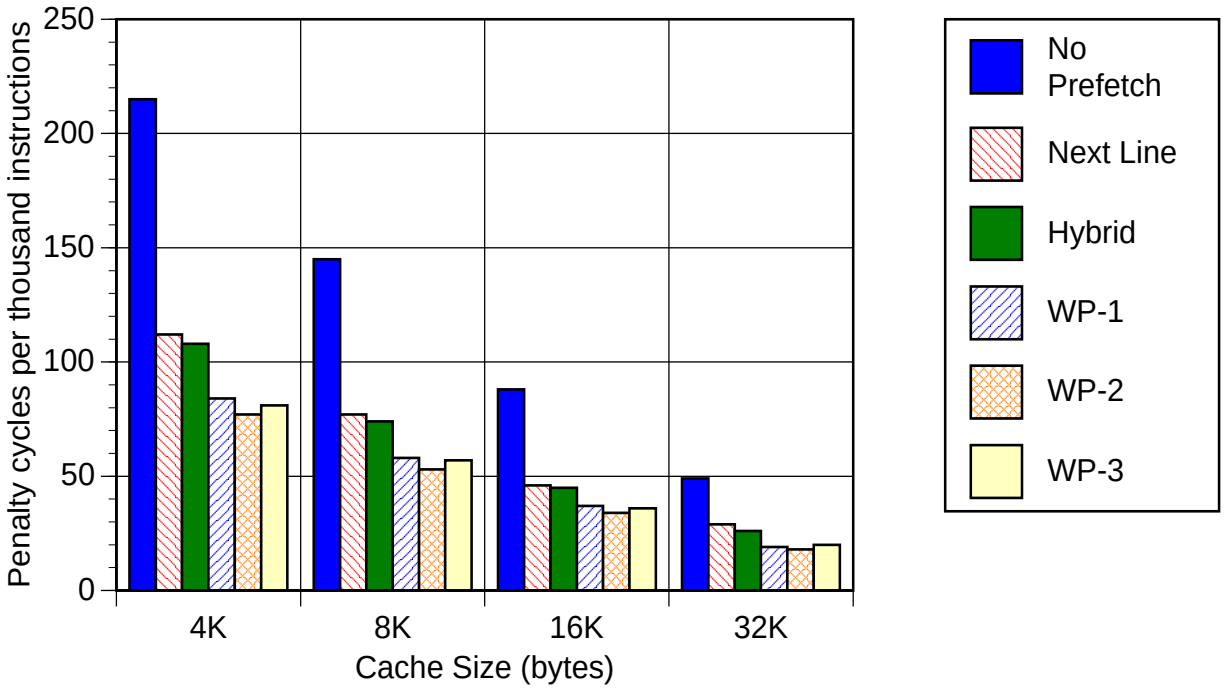

FIGURE 4. Penalty index comparison of different prefetch algorithms. Each cache is direct-mapped with line size of 32 bytes. Miss latency is 5 cycles.

over many cache configurations including different line sizes, associativities, and fetchahead distances. The results shown are representative of all the cache configurations we studied.

As was expected, the wrong-path algorithms generate more traffic than the other schemes, as shown in Figure 5. WP-3 was the worst offender, again showing the pollution effect of prefetching too many consecutive lines. For these benchmarks, the hybrid algorithm performed little better than next-line prefetching. The simulator was modified to show which types of prefetches were most productive. To do this a prefetch gain was defined to be a prefetch which removes one cache miss, i.e., had their been no prefetch, an instruction miss would have occurred. Figure 6 shows the breakdown of which prefetches remove the cache misses. Next-line prefetching, of course, does no target prefetching so has no target prefetch gain. In the wrong path algorithms, target prefetching sometimes prefetches lines which would have been prefetched by the next-line mechanism. This explain why the next-line gains are not as high as in pure next-line prefetching. It is interesting to note that the hybrid scheme receives little benefit from its target buffer; almost all its performance is gained from its next-line prefetching component. This is not the result found in [3] where the benefits from next and target prefetching were shown to be about equal. We reran the experiments using 64 and 128 entry target buffers but the results varied little from those using the 32 entry buffer. This implies that the lack of target prefetching is not due to table entry contention. We believe that the lack of target prefetching we see is due to our larger cache sizes. In [3] the caches were quite small. With larger caches, once the line is brought into the cache, it is more likely to be there on subsequent executions. As we noted earlier, table-based prefetching will not occur during first time code execution because the links are not set up in the table. During subsequent executions of the same code, the lines will still be resident in a large cache and again

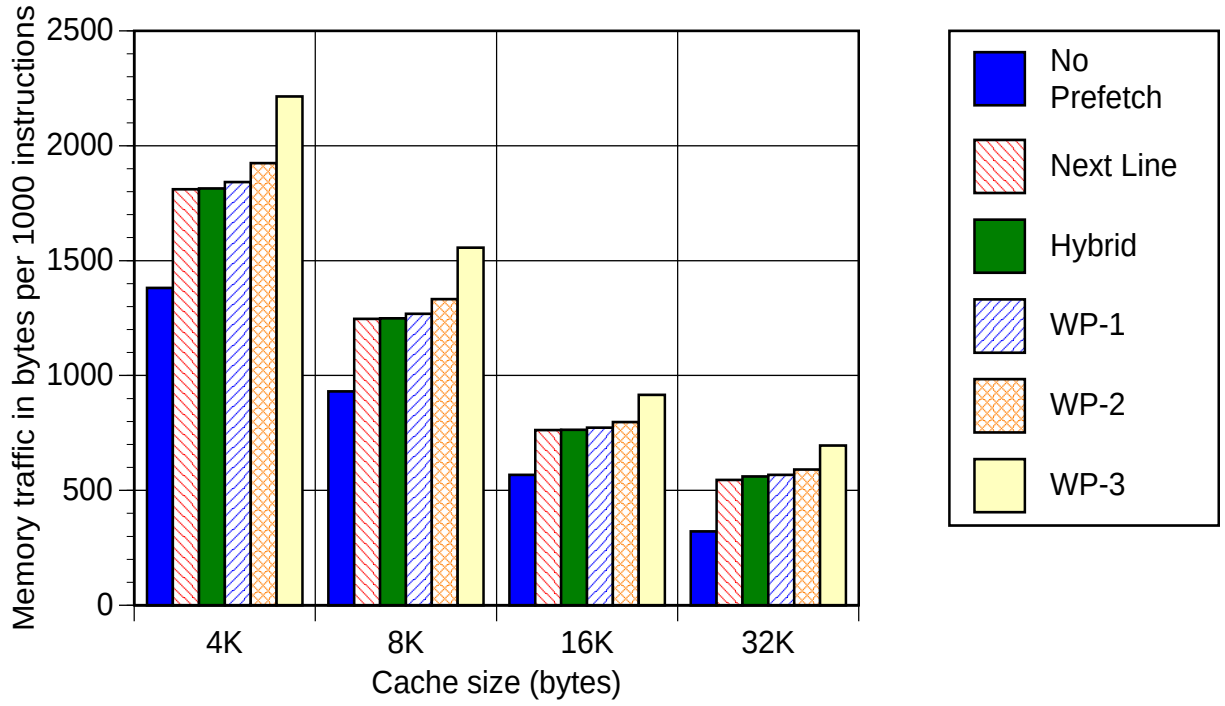

FIGURE 5. Traffic comparison generated by different prefetch algorithms. Each cache is direct-mapped with line size of 32 bytes. The fetchahead distance is 24 bytes.

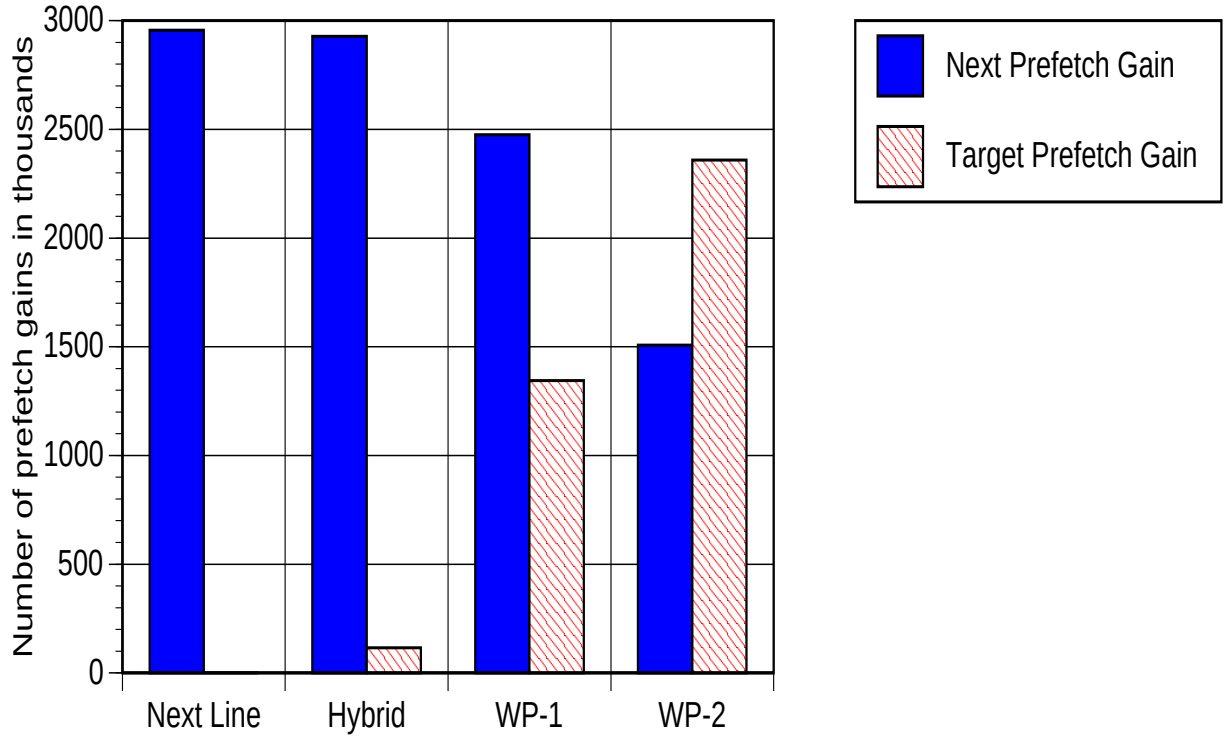

FIGURE 6. Prefetch breakdown for different algorithms. The fetchahead distance is 24 bytes.

no prefetching is done. Thus the prefetcher has little occasion to benefit from the target buffer. Other differences in our results and those found in [3] could stem from our having different hardware considerations (e.g., allowing only one memory access initiation per cycle) or our use of different architectures and benchmarks.

Our results also show that prefetching algorithms become less effective as the miss latency increases. Figure 7 shows that for a 5 cycle miss latency, the 1 line wrong-path algorithm reduces the misses by over 50%. When the latency increases to 10 cycles, the reduction is around 30% and

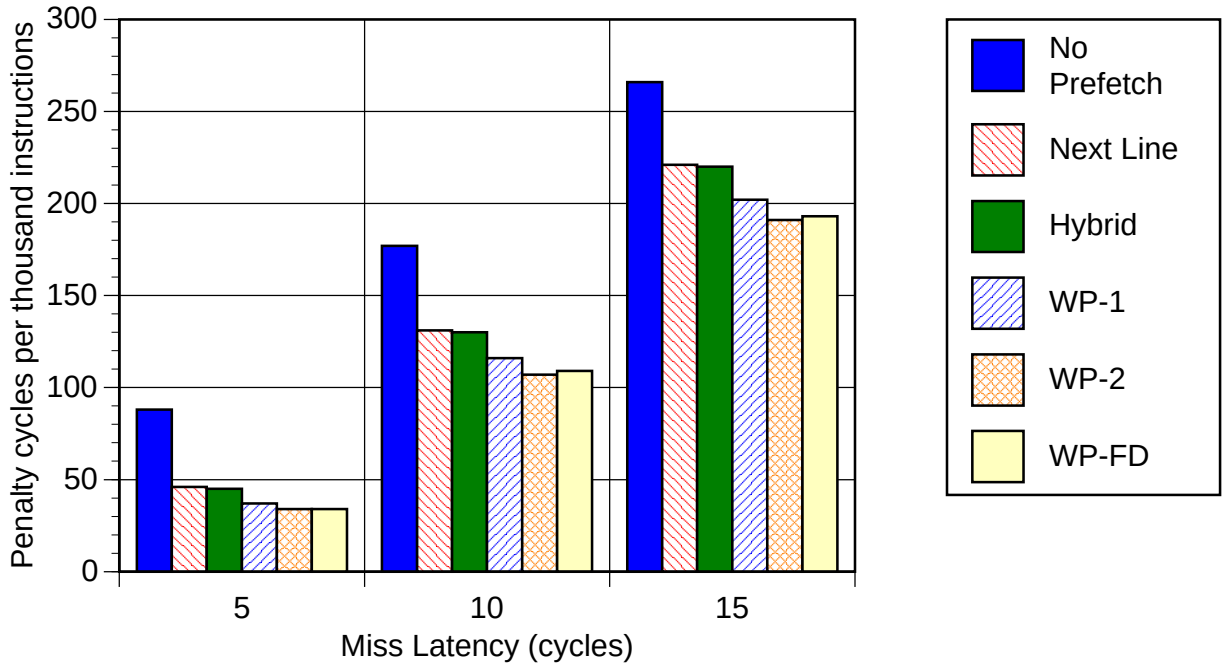

FIGURE 7. Prefetch performance for different miss latencies. The cache is 16K, direct-mapped, with a line size of 32 bytes.

at 15 cycles it is down to around 20%. The other algorithms showed a similar performance decline with increasing latency. The relative difference in performance between the wrong-path and hybrid algorithms remained about the same for different latencies. We expected to see wrong-path prefetching perform better relative to the other algorithms as the miss penalty increased. Our reasoning was that there is more time between a wrong-path line prefetch and its execution than between a correct-path line and its execution. This greater time between prefetch and execution would absorb a longer memory latency. Our results do not show this however, and it is unclear whether our intuition was wrong or something else is going one which offsets the prefetch time gained by prefetching the wrong path.

Another unexpected result was the unattractiveness of the decode buffer versions of wrong-path prefetching. The decode buffer algorithms require additional hardware to partially decode instructions in a prefetch buffer. Branch and jumps are detected and their targets are then calculated before the execution pipeline decode stage. This allows a target prefetch to be initiated a few cycles before the target instruction is fetched. The WP-FD algorithm uses a decode buffer the same size as the line size. Thus, if the line size is 8 words and instructions are one word long, 8

instructions are partially decoded in parallel. The control instruction target addresses are queued for possible prefetch. During each subsequent cycle, if the memory bus is free, the cache line corresponding to the first queued target address is checked for residency and is prefetched if necessary. WP-HD is similar except that only the last half of the words in the decode buffer are decoded. This is a cost-saving implementation since less hardware is required than full buffer decode yet many targets can still be prefetched in advance. However, as shown in Figure 8, it never performed as well as even the WP-1 version which requires much less hardware. WP-FD is not a clear-cut winner either. It does perform better than WP-1 but only when the cache is large does it do better than WP-2. When implemented with a 32K cache, WP-FD performed best over all the experiments with a penalty index reduction of 70%. Since WP-2 requires much less hardware, the best choice between the two algorithms depends upon cache size and hardware cost.

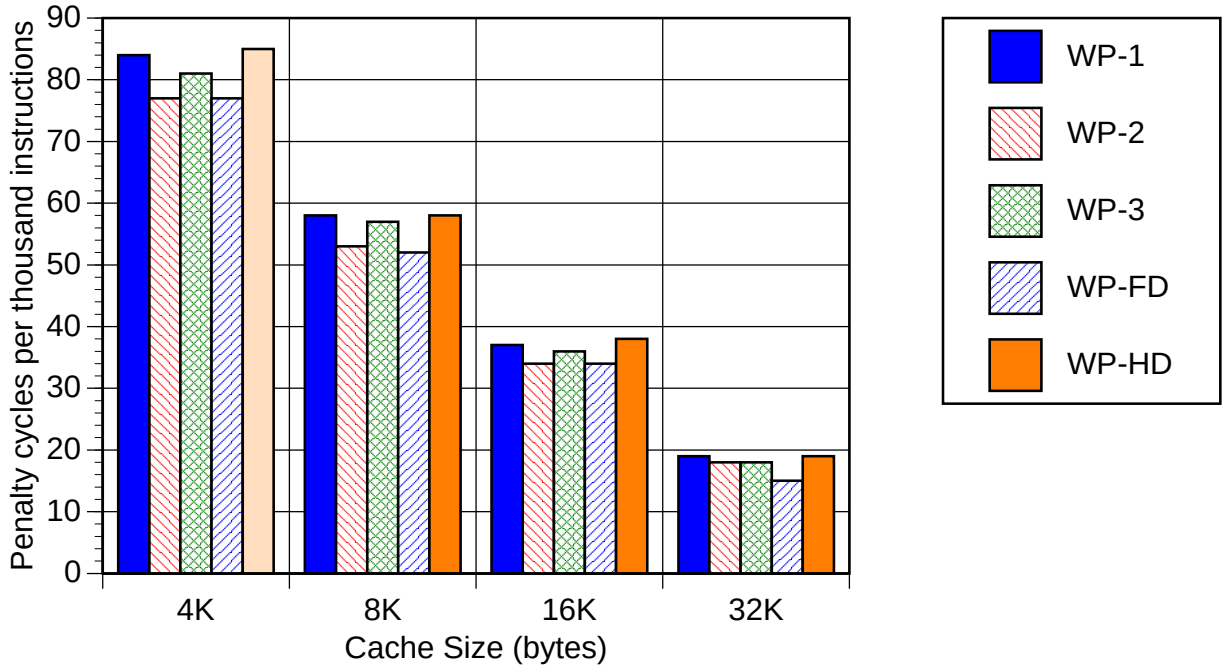

FIGURE 8. Penalty index comparison of more hardware intensive algorithms. All caches are direct mapped with line size of 32 bytes. The fetchahead distance is 24 bytes. Miss latency is 5 cycles.

Since WP-FD's relative performance improves with increased cache size, it would appear to suffer from the pollution caused by the prefetching of many targets. As the cache gets bigger or associativity increases, the pollution effect is hidden and its earlier prefetch initiation becomes evident in a slightly reduced number of penalty cycles.

6 Conclusions

Wrong-path prefetching combines next-line prefetching with the prefetching of all branch targets regardless of the predicted direction of branches. The algorithm substantially reduces the cost of instruction cache misses while somewhat increasing the amount of memory traffic. A measure called the penalty index is introduced to compare the performance of various prefetching algorithms. For all the cache configurations and benchmarks we studied, wrong-path prefetching

achieved higher performance than the other simulated prefetch algorithms. In addition, its hardware requirements are no greater than that of next-line prefetching and substantially less than table-based methods. If the cache is large, a slight performance gain can be acquired by using additional hardware to implement a partial decode prefetch buffer which enables target prefetching to be initiated earlier. In short, wrong-path prefetching is a surprisingly attractive method to reduce the detrimental effects of instruction cache misses.

Acknowledgments

We would like to thank Konrad Lai of the Intel Corp. for the ideas initiating this work and his continued support. We would also like to thank David Nagle for his careful proofreading and thoughtful suggestions.

References

[1] D. Alpert and D. Avnon, “Architecture of the Pentium Microprocessor,” *IEEE MICRO*, June 1993, pp. 11-21.

[2] T. Chen and J. Baer, “Reducing Memory Latency via Non-blocking and Prefetching Caches,” *Proc. of the 5th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, Oct. 1992, pp. 51-61.

[3] W.-C. Hsu and J. Smith, “Prefetching in supercomputer instruction caches,” *Supercomputing ‘92*, Nov. 1992, pp. 588-597.

[4] N. Jouppi. “Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers,” *Proc. of the 17th Int. Symp. on Computer Architecture*, Aug. 1990, pp. 364-373.

[5] A. Klaiber and H. Levy, “An Architecture for Software-Controlled Data Prefetching,” *Proc. of the 18th Int. Symp. on Computer Architecture*, May 1991, pp. 43-53.

[6] D. Kroft, “Lockup-free Instruction Fetch/prefetch Cache Organization,” *Proc. of the 8th Int. Symp. on Computer Architecture*, May 1981, pp. 81-87.

[7] E. McLellan, “The Alpha AXP Architecture and 21064 Processor,” *IEEE Micro*, June 1993, pp. 26-47.

[8] J. Pierce and T. Mudge, IDtrace: A Tracing Tool for i486 Simulation, Technical Report CSE-TR-203-94, Dept. of Electrical Engineering. and Computer Science, University of Michigan.

[9] J. Pierce and T. Mudge, “The Effect of Speculative Execution on Cache Performance,” *Proc. of the Int. Parallel Processing Symposium*, April 1994, pp. 172-179.

[10] J. Quinlan and K. Lai, Tynero: A Multiple Cache Simulator, Technical Report, Intel Corp., Hillsboro, OR, May 1991.

[11] A.J. Smith, “Cache Memories,” *ACM Computing Surveys*, Sep. 1982, pp. 473-530.

[12] M. Smith, Tracing with Pixie, Technical Report, Center for Integrated Systems, Stanford University.

[13] S. Weiss and J. Smith, *Power and the PowerPC*, San Mateo, CA: Morgan Kaufmann, 1994.