Wrong-Path Instruction Prefetching

Jim Pierce
Intel Corporation

Trevor Mudge
University of Michigan

Abstract

Instruction cache misses can severely limit the performance of both superscalar processors and high speed sequential machines. Instruction prefetch algorithms attempt to reduce the performance degradation by bringing lines into the instruction cache before they are needed by the CPU fetch unit. There have been several algorithms proposed to do this, most notably next line prefetching and target prefetching. We propose a new scheme called wrong-path prefetching which combines next-line prefetching with the prefetching of all control instruction targets regardless of the predicted direction of conditional branches. The algorithm substantially reduces the cycles lost to instruction cache misses while somewhat increasing the amount of memory traffic. Wrong-path prefetching performs better than the other prefetch algorithms studied in all of the cache configurations examined while requiring little additional hardware. For example, the best wrong-path prefetch algorithm can result in a speed up of 16% when using an 8K instruction cache. In fact, an 8K wrong-path prefetched instruction cache is shown to achieve the same miss rate as a 32K non-prefetch cache. Finally, it is shown that wrong-path prefetching is applicable to both multi-issue and long L1 miss latency machines.

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 in superscalar architectures increases, the performance degradation caused by cache misses will become even more apparent. Designers have proposed several strategies to increase the performance of the cache memory systems which will be implemented in next-generation microprocessors. The option often used is to increase the cache size and/or its associativity. However, an increasingly important set of techniques are to prefetch instructions and data into the cache [2][3][5][6][8].

In the case of instruction caches, the focus of this paper, prefetching is an attempt to fetch lines from memory into the cache before the instructions in the line are referenced by the processor's fetch unit. To be effective, the prefetch strategy must accomplish two things. It must be able to guess which cache lines will soon be referenced and it must initiate the prefetch requests far enough in advance of instruction fetch so that the miss latencies are significantly reduced or eliminated entirely. Theoretically, an optimal prefetch algorithm could remove all cache misses by prefetching all instructions immediately 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. In this chapter, a new prefetching algorithm is proposed which makes no attempt to predict the correct direction. In fact, it relies heavily on prefetching the wrong direction. This method outperforms several previously proposed prefetching schemes, but it does so at a lower hardware cost. It's simplicity also makes it suitable for combining with more sophisticated prefetch techniques.

The work in this paper grew out of a study of instruction prefetching in [13], where the idea of wrong path prefetching was first proposed. It was further discussed in [11]. Proposals have also been made for prefetching data from data caches [1][9][14] however, this paper does not address data prefetching issues.

2 Prefetching Methods

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

0-8186-7855-0/96 $5.00 (c) 1996 IEEE

2.1 Long Cache Lines

The simplest form of prefetching is the use of long cache lines [15]. 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. Longer cache lines also reduce the amount of space required for tag storage. 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.

2.2 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. 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. 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 computed. 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 except in the case of short branches 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 the CPU to access it. However, increasing the fetchahead distance increases the probability that a branch will be encountered in the current line after the prefetch begins, 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% [7].

2.3 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, i.e., prefetch the line which was referenced next the last time the current line was executed. 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 the 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.

2.4 Hybrid Schemes

A hybrid scheme which combines both next-line and target prefetching was proposed in Hsu and Smith [8]. 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. They compared the performance of this scheme with next-line and target-line algorithms using supercomputer reference traces and their results were impressive—miss rates are reduced by 50-60%. In addition, they showed that the performance gain of the hybrid method was 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 drawbacks, however. First, significant hardware is required for the table and the associated logic which performs the table lookups and updates. This requires additional chip area that has the potential to increase CPU cycle time. Secondly, the extra hardware has only limited effectiveness in that it cannot be used to remove certain types of misses. First time accessed code does not profit from table-based target prefetching since the table must first 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 [8].

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 many 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 because only lines soon to be executed should be prefetched. The alternative, fetching wrong path lines into the cache, would seem to increase memory traffic and cause cache pollution.

2.5 Wrong-Path Prefetching

We will show that the intuition expressed above is partially false. Our earlier studies have shown that executing instructions down mispredicted paths actually reduced the number of cache misses occurring during correct path execution [13]. This suggests that prefetching instruction cache lines down mispredicted paths might have a positive result. Accordingly, we have termed this new prefetch algorithm *wrong-path prefetching.* It 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 the target prefetching component. No target line addresses are saved and no attempt is made to prefetch only the correct execution path. Instead, 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. In this case, if the target address is not in the cache, a fetch miss and a prefetch request of the same line will be generated simultaneously. Similarly, prefetching unconditional jump or subroutine call targets is useless since the targets are always taken and the prefetch address would be produced too late. To reiterate, the target prefetching part of the algorithm can only perform a potentially useful prefetch for a branch which is not taken. This is why the algorithm is called wrong-path prefetching. However, if execution returns to the branch in the near future, and the branch is then taken, the target line will probably reside in the cache because of the prefetch.

The hardware requirements for wrong-path prefetching are roughly equivalent to what is required for next-line prefetching since the target prefetch addresses are generated by the existing decoder and no target addresses are saved. The obvious advantage of wrong-path prefetching over the hybrid algorithm is that there is a lower hardware cost. 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 has a long latency, the prefetch may not be initiated soon enough. Conversely, wrong-path prefetching prefetches lines down a path which is not immediately taken thus it potentially has more time to prefetch the line from a slow memory before the path is executed.

The potential advantages of wrong-path prefetching would not come without cost. Prefetching down not-taken paths will put some 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.

3 Experimental Method

This section details the experimental procedure used to study the performance characteristics of the previously mentioned prefetching algorithms. A detailed, trace-driven memory system simulator was written to model the

performance of the instruction cache, memory bus, prefetch unit, and processor fetch unit in typical current and next-generation microprocessors. The simulator can model the behavior of a wide range of system configurations.

3.1 Memory System Model

The base model is a single issue pipelined processor with RISC-like properties. Each instruction takes one cycle to execute and all instructions are of uniform 4 byte length. The CPU clock speed is assumed to be high so the disparity between clock speed and memory access time is large. Conventional split L1 instruction and data caches are implemented which connect directly to memory or to a second-level, L2, cache. Instruction cache accesses are not pipelined and cache hits complete in one cycle. The base cache model is a 16K direct-mapped cache with 32 byte lines and a 16 byte/cycle refill rate. Four wait cycles are required before the first byte is transferred into the cache. These parameters will be varied in the experiments described in Section 4.

To handle both instruction fetch and multiple outstanding prefetches, a non-blocking instruction cache is necessary [1][10].

The prefetch unit suggests when a line should be prefetched, computes the line's address, and performs a cache tag lookup to see if the line is resident in the cache. If the line is not resident, the prefetch is initiated and a memory request is sent to the non-blocking cache handler. The cache tag structure can be accessed simultaneously by both the fetch and prefetch units. This requires that the tag structure be either dual-ported or replicated. Only one prefetch cache tag lookup can occur per cycle. If both a next-line and target-line prefetch is suggested in one cycle, the target-line address takes precedence. The next-line prefetch will be suggested during the next cycle if necessary.

For the next-line prefetching component of all algorithms, the default fetchahead distance is 3/4 of the line size. For table-based target prefetching, the default target table is direct-mapped and has 128 2-word entries. For wrong-path prefetching, a target prefetch is suggested during the cycle in which a conditional branch is decoded. The instruction decoder forwards the target address to the prefetch unit where the cache tag lookup occurs. To summarize the changes necessary for prefetching, the following hardware additions are required:

- Structures for a non-blocking instruction cache.
- Cache tag replication (or two tag ports) and tag comparison logic.
- Next-line prefetching requires logic for fetchahead distance calculation and next address calculation.
- Target-line prefetching requires next-line hardware, target table, logic to compare and update table entries, and data paths between table and cache tags.
- Wrong-path prefetching requires next-line hardware and a data path between instruction decoder and prefetch unit.

Further micro-architecture details can be found in [13].

3.2 Traces Used in Simulation

The traces were taken from the IBS traces obtained by Nagle and Uhlig using a hardware monitoring setup [12]. The platform is a MIPS R3000-based DECstation 5000/200 running a Mach 3.0 operating system kernel. The traces contain physical instruction and data addresses gathered directly from the system bus while executing between 40 and 130 million kernel and user mode instructions. The benchmark programs are listed with sample cache activity in Table 1.

Trace	Description	Instr	Misses	Traffic
gcc	Gnu C compiler	126M	111	16
gs	PS file viewer	86M	110	16
mpeg-play	Video displayer	111M	94	15
sdet	From SPEC SDM suite	43M	139	16
verilog	HDL tool	52M	115	17

Table 1 Trace Benchmark Descriptions. The miss column is the number of misses per thousand instructions. The traffic column is the percentage of time the bus is active with instruction information. The cache is 8Kbytes.

3.3 Performance Measurement

Most cache studies use the number of cache misses or miss ratio as the performance measure. However, miss reduction is not a sufficient measure of the performance of prefetched caches because it does not account for changes in miss latencies produced by prefetching. For instance, suppose the full miss latency for a processor is five cycles and one algorithm initiates a prefetch three cycles before the fetch while the other initiates a prefetch only one cycle before the fetch. An instruction miss will occur in both cases but the net result is not the same. The first prefetch algorithm will have reduced the miss penalty by two more cycles, thus giving better performance. Two statistics are

used to compare the algorithms: total CPU cycles and miss cycles. The latter is the number of cycles wasted due to instruction misses, i.e.,

Total CPU cycles = Instructions executed + Miss cycles

The measures of memory bus traffic are the total bus cycles and the bus utilization. The former is the number of cycles during which the bus is busy sending or receiving memory requests. The bus is deemed not busy during wait cycles. The bus utilization is the percentage of total CPU cycles during which the bus is busy. The results are based upon the sum of the cycle counts for all five of the IBS benchmarks.

4 Comparison Experiments

This section summarizes the results of a comprehensive empirical study of the aforementioned prefetching algorithms using the trace driven simulator. The complete results can be found in [13]. The first 2 subsections compare algorithm performance for different cache sizes, line sizes, refill rates, and cache associativity. Section 4.3 explains the reasons behind the hybrid algorithm's ineffectiveness and why these results differ from those found in [8]. Section 4.4 examines the effect of increased latency. Finally, Section 4.5 shows the effects of prefetching on multi-issue microprocessors.

4.1 Cache Size

The left graph in Figure 1 shows the number of CPU cycles needed to execute the system traces with no prefetching and with different prefetch algorithms. The right graph shows the percentage reduction in the execution cycles due to the prefetching algorithm. Wrong-path prefetching performed the best in terms of CPU cycles for all cache sizes, producing up to a 16% speedup over that of no prefetching. It performs up to 4% better than the other algorithms.The hybrid algorithm performed only slightly better than next-line prefetching. It seems to extract little performance gain from its additional hardware.

It is interesting to note in the left graph that a wrong-path prefetched 8K cache gives better performance than a non-prefetched 32K cache and a next-line or hybrid prefetched 16K cache.

Since prefetching can only reduce the miss latency cycles, which in the above caches comprise 20-40% of the total execution cycles, the prefetching algorithms must be removing a high percentage of miss cycles. We found that wrong-path prefetching reduces miss cycles by almost 40% in an 8K cache.

The major disadvantage of prefetching is the additional memory traffic it generates as can be seen in Figure 2. Two factors combine to cause this increased bus utilization. The first is that prefetching generates unnecessary references which increases the bus traffic. The second is that prefetching reduces the total CPU cycle count. More traffic in a shorter time period translates into higher bus utilization.

Bus bandwidth should be viewed as a resource just as a functional unit or memory port. For instance, if the bus bandwidth is allowed to grow by doubling the refill rate, implementing a prefetching scheme can significantly reduce the required cache size without greatly increasing the bus traffic. Figure 3 shows the effect of this resource allocation. Cache A, B, and C are non-prefetched 32K caches with a 32 byte line size and 8, 16, and 32 byte/cycle refill rates respectively. Cache E is a wrong-path prefetched 8K cache with a 32 byte line and 32 byte/cycle refill. Cache E has the best CPU cycle performance and its bus traffic is only slightly higher than that of Cache A. Caches B and C are included in the graph to verify that the performance improvement is a result of prefetching and not of the increased refill rate.

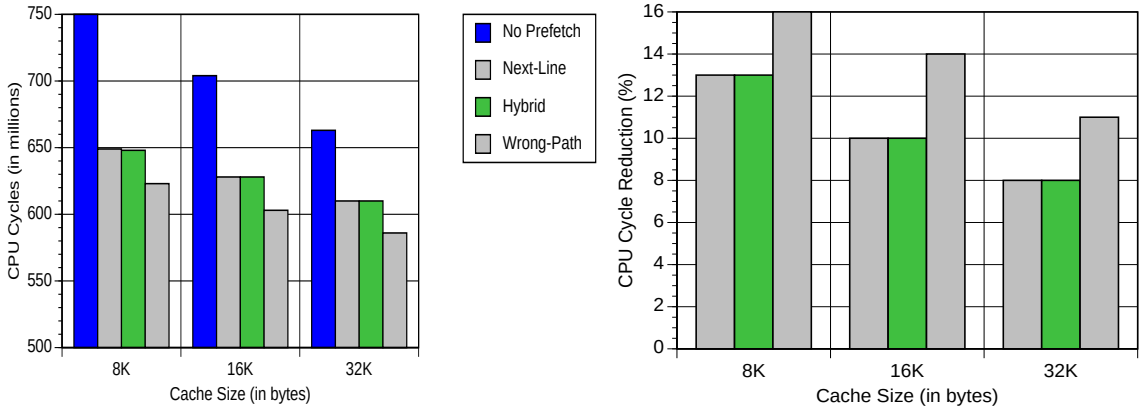

Figure 1 Cycle Time for Different Cache Sizes - The left graph shows total CPU cycles and the right graph shows the cycle improvement gained by prefetching.

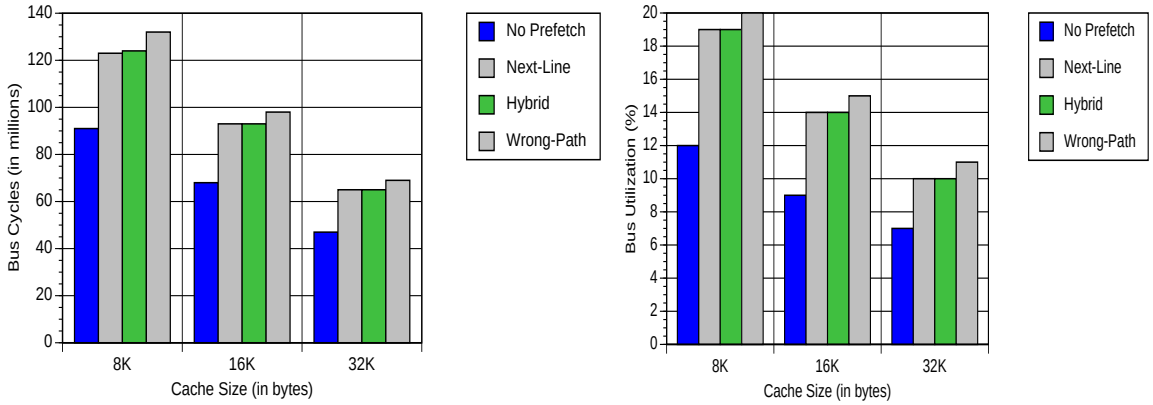

Figure 2 Bus Traffic For Different Cache Sizes - The left graph is the total bus cycles and the right graph is the bus utilization. System traces are used for simulation input.

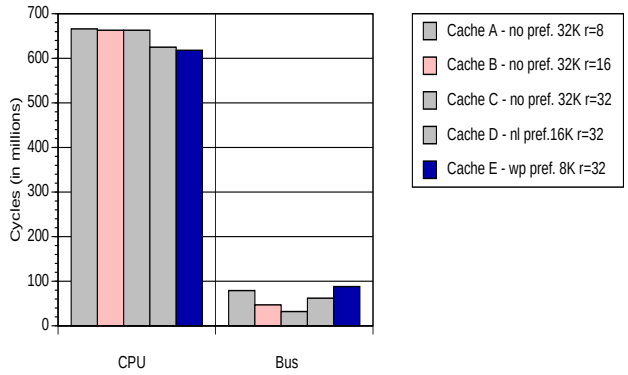

Figure 3 Prefetching Allows Better Performance with Smaller Cache - Caches A, B, and C are non-prefetched 32K caches with 8, 16, and 32 byte/cycle refill rates. Cache D is a next-line prefetched 16K cache with 32 byte/cycle refill rate. Cache E is a wrong-path prefetched 8K cache with 32 byte/cycle refill rate.

4.2 Cache Associativity and Refill Size

The cache line refill rate plays an important role in the effectiveness of prefetching. A small refill rate means that many bus cycles are required to transfer a line from memory to the cache. This exacerbates two detrimental effects. First, prefetching inevitably causes unnecessary lines to be transferred to the cache. A small refill rate puts a high cost, in terms of bus cycles, on these wasted transfers. Secondly, longer miss handling increases the fetch memory request delay. When a fetch miss occurs during the service of a prefetch request, the memory request cannot be sent until the prefetch is completed. Increased line segment transfer cycles, caused by the smaller refill rate, increase the fetch delay and thus the execution time. Our results show that the higher the prefetch rate, the better the prefetch performance.

An anticipated problem of prefetching, especially wrong-path prefetching, is its potential to pollute the cache with non-accessed lines. Caches with higher associativity can absorb this pollution and will improve prefetching performance. However, although overall performance improved with higher associativity, miss cycle reduction actually declines as associativity increases. Prefetching algorithms excel at bringing lines back into the cache which were previously displaced. Much of their performance benefit is derived from this behavior. High associativity takes a chunk out of the prefetch gain by reducing the number of line displacements in the cache. One might then suggest that higher associativity be used in place of prefetching. We found however that prefetching works in conjunction with increased associativity. Furthermore, prefetching is unlikely to affect the cache hit access times, while highly associative caches are likely to increase the CPU cycle time or require pipelined cache accesses which would negate its perceived advantages [4].

4.3 Prefetch Effectiveness

To summarize the results thus far, prefetching is effective in reducing CPU cycle times in all of our studied cache configurations obtained by varying cache size, line size, associativity, and refill rate [13]. The cost is an increase in bus traffic. Wrong-path prefetching performed slightly better than next-line prefetching in all cases, sometimes by as much as 4%, and the difference in hardware cost difference between the two methods is small. On the other hand, the hybrid scheme gives almost equivalent performance to that of next-line prefetching but at considerable additional hardware cost. This was not the result found by Hsu and Smith in their comparison study in which the performance gain from the hybrid algorithm was approximately the sum of the gains individually achieved by next-line and table-based target prefetching.

We believe the differences in results are mainly due to the difference in cache sizes and associativities modeled. They studied supercomputer cache behavior with cache sizes ranging from 128 to 2K bytes in size. To be effective, table-based target prefetching requires a high line turnover rate in the cache. The larger caches and high associativity of today's microprocessor caches limit the turnover rate and subsequently limit the effectiveness of table-based prefetching. These conclusions are verified by several experiments in [13]. We found that in the hybrid scheme, the miss reduction due to table-based target prefetching was negligible compared to that of next-line prefetching. This is due to a combination of two factors:

- Compulsory misses are not eliminated since a previous execution is required to add the necessary link in the target table.
- After the first miss and the table is updated, the line will be cache resident. In order for the line to be prefetched, the line will first have to be displaced by the cache. This is less likely to occur in a larger cache.

In wrong-path prefetching, target address prefetching accounted for roughly 1/3 of the reduction in miss cycles. Furthermore, it is interesting to note that target prefetches in wrong-path prefetching were useful over 75 percent of time. In other words, 75% of the lines prefetched because they were targets of not-taken branch were referenced before they were displaced. The affects of target table size and fetchahead distance were also measured and recorded in [13].

4.4 Effect of Higher L1 Miss Latency

How do the prefetch algorithms scale to increased memory latency? As the memory latency increases, prefetching algorithms must prefetch farther in advance of the current execution point in order to cover the latency period. Therefore, it might be expected that the performance advantage of prefetching will diminish as the latency increases. Nevertheless, of the three algorithms studied, the wrong-path algorithm might still be expected to suffer the least from increased latency since its target prefetching component does not prefetch immediately used instructions. Figure 4 shows the results of varying the number of memory wait cycles. As expected, the CPU cycle time increases as the number of wait cycles is increased. Unexpectedly, the CPU cycle reduction due to prefetching also increases as the number of wait cycles is increased. As memory latency increases, prefetching becomes more effective.

The reason for this is that the reduction in the number of instruction miss cycles becomes more visible as the portion of total CPU cycles due to instruction misses increases. Higher latency causes more instruction miss cycles which, in turn, allows the prefetching algorithm to have a greater overall effect as Figure 4 shows.

To investigate the effect of memory latency on each prefetching component, we define the *prefetch distance* to be the number of cycles between a prefetch initiation and the first instruction fetch request to the prefetched line. By having the simulator record these values for every next-line and target-line prefetch gain in the wrong-path algorithm, we verify that target-line prefetches do indeed prefetch farther ahead of the current execution point. Figure 5 shows the results. For both a high and a low fetchahead value, the target-line prefetch gains have statistically higher prefetch distances.

4.5 Effect on Multi-Issue Machines

It has been shown that prefetching allows increased performance on single issue architectures. To what extent does it also demonstrate performance gains in superscalar or VLIW machines with higher IPC? There are two opposing factors which are similar to those concerning the

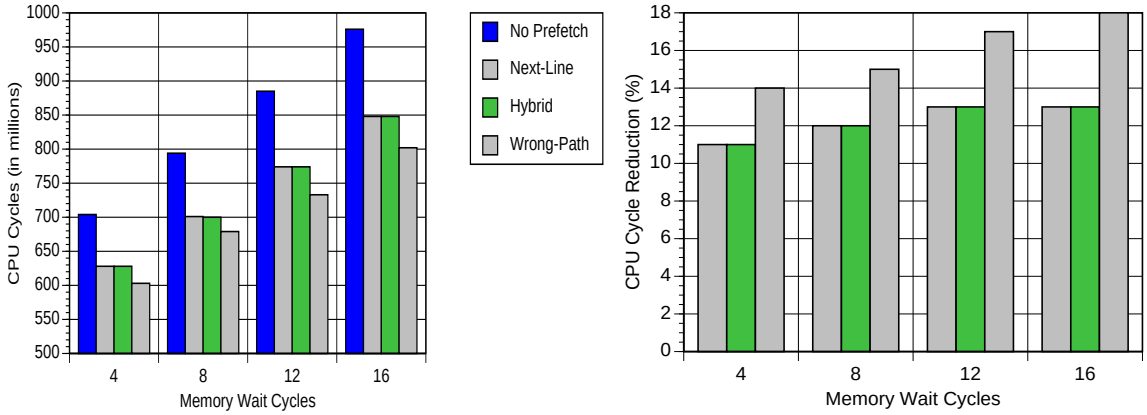

Figure 4 CPU Cycles for Different Memory Latencies.

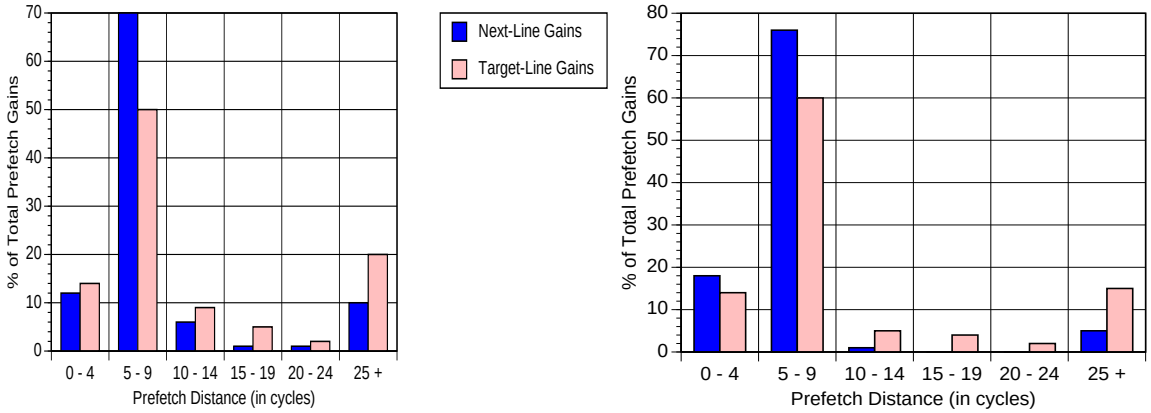

Figure 5 Prefetch Distance Distribution - The graphs show the distribution of prefetch distances for all next-line and target-line prefetch gains within the wrong-path algorithm. The caches are 16K, direct-mapped, line size 32, and refill rate 16 bytes/cycle. In the left graph, the next-line fetchahead distance is 24 while on the right it is 8.

variation in the number of memory wait cycles. In multi-issue machines, since the execution rate is higher, the prefetch algorithm must prefetch farther in advance of the current execution point. This will degrade prefetch performance since a larger portion of the prefetches will not arrive in time. Conversely, increasing the issue rate reduces the time spent executing instructions and makes the effects of misses more pronounced. In multi-issue machines, a larger fraction of the total execution time is due to miss latency cycles. Since prefetching removes a portion of the miss cycles, the effects of prefetching could be greater than in single-issue microprocessors.

Another element of design complexity which goes hand in hand with issue rate is the number of memory ports, i.e, the number of memory requests which can be processed concurrently. Adding memory ports will expand the memory-to-cache bandwidth so the seriousness of prefetch-induced traffic should be lessened. In addition, multiple ports will reduce the backlog of waiting memory requests which will have two positive effects. The first is a reduction in fetch miss request delays caused by in progress prefetches. With only a single port, if a previous prefetch is receiving a cache line on the bus, a fetch miss memory request will be stalled until the line is transferred. Multiple ports will often allow the fetch request to be initiated immediately. Secondly, prefetches will be initiated earlier which will increase the prefetch distance.

The effects of increasing the instruction issue rate and adding additional memory ports to the cache are examined in this section. Multiple instruction issue is simulated by maintaining a fetch window which, at the beginning of each cycle, contains the maximum number of instructions which can be issued per cycle. Each instruction which does not cause a fetch miss is removed from the window. At the beginning of the next cycle, the fetch window is refilled from the trace. This is only an approximation to that of an actual processor since number of instructions retired per cycle also depends upon data dependencies, exceptions, and branch mispredictions. The model for multiple memory ports is an interleaved memory with the same number of banks as ports. As long as two memory requests do not address the same bank, they can be handled in parallel. Each port has its own memory bus so request operations are truly independent if no bank conflicts occur. If a conflict does occur, one request waits in the non-blocking cache structure until the port is available. The memory is partitioned into banks at the cache line level, i.e., one memory bank contains every word in the cache line. Consecutive cache lines are stored in consecutive banks.

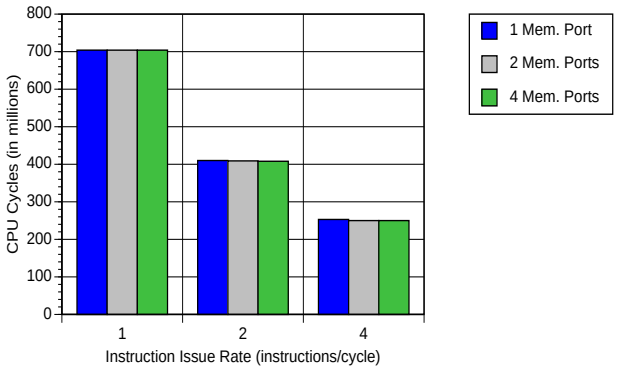

Figure 6 Effect of Issue Rate and Memory Ports with No Prefetching - The graph shows the cycles required to execute the system traces for different issue rates and

The following graphs show the results of varying the issue rate and the number of ports when using the system traces as input to the simulator. Notice that in Figure 6 when the issue rate is doubled the CPU cycles does not decrease by a factor of two. This is because the miss latency cycles are not reduced. Figures 7 and 8 show that

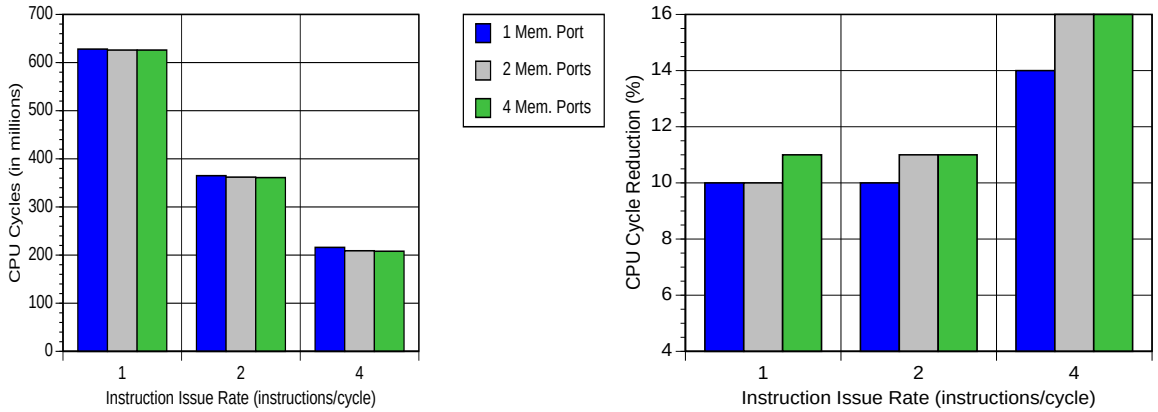

Figure 7 Effect of Issue Rate and Memory Ports with Next-Line Prefetching - The left graph shows the CPU cycles required to execute the system traces. The right graph shows the percent reduction in CPU cycles over the case when no prefetching is done.

the effectiveness of prefetching increases as the issue rate increases. Next-line prefetching improves performance by up to 16% and wrong-path prefetching by 21% in a machine with an issue rate of 4.

The graphs seem to show that the number of memory ports is rather unimportant but this is misleading. First, as mentioned previously, multiple memory ports are probably necessary to achieve an average IPC of 2 or 4. In addition, multiple ports are also required to handle the memory traffic. With superscalar execution, the same amount of memory traffic will occur in a shorter amount of time. Figure 9 shows the port utilization which is defined as the percentage of CPU cycles during which the port is busy sending or receiving information. For multiple port configurations, the highest utilization percentage is given but, since the requests are relatively randomly distributed over all the banks, there is little variation in the utilizations between the ports. It can be observed that even when no prefetching is used, additional ports are necessary to keep the traffic down to a reasonable level. If prefetching is implemented, multiple ports are a must.

5 Summary and Conclusions

The work reported here sheds light on the applicability of instruction cache prefetching schemes in current and next-generation microprocessor designs where memory latencies are likely to be longer. In particular, a new prefetching algorithm is examined that was inspired by previous studies of the effect of speculation down mispredicted paths. The highlights of the experimental results are:

- Prefetching achieves significant performance gains in terms of CPU cycle reduction - up to 14% reduction with standard cache configurations.
- Wrong-path prefetching achieves higher performance than other algorithms in all studied cache configurations. At the same time, its hardware

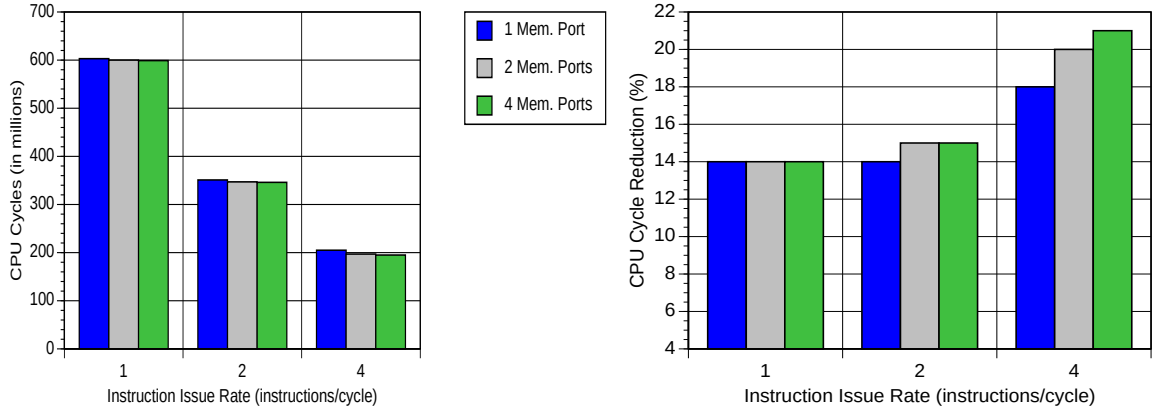

Figure 8 Effect of Issue Rate and Memory Ports with Wrong-Path Prefetching - The left graph shows the CPU cycles required to execute the system traces. The right graph shows the percent reduction in CPU cycles over the case when no prefetching is done.

cost is equivalent to next-line prefetching. Somewhat surprisingly, 75% of all not-taken path prefetches result in miss reductions.

- The cost of prefetching is the increased bus traffic. Bus utilization rises from 12% for no prefetching to up to 25% for prefetching with standard cache configurations. Bus bandwidth may be viewed as a resource which can be employed to reduce miss cycles. By implementing prefetching and adding hardware to increase refill rate, equivalent memory performance can be achieved with a much smaller cache.
- Table-based target prefetching performs poorly with current generation instruction cache sizes rendering the hybrid algorithm performance equivalent to that of next-line prefetching. The negligible performance gain is not worth the cost of the additional hardware required to implement the target table.
- Prefetching is more effective as memory latencies increase. Wrong-path prefetching reduced CPU cycles by 18% when the wait cycles rose to 16.
- Wrong-path prefetching becomes even more effective as instruction issue width increases. Our results show a greater than 20% cycle reduction is some cases.

6 Acknowledgments

Jim Pierce was supported by a grant from the Intel Corporation while pursuing this work at The University of Michigan. The work was also supported in part by ARPA under Grant DAAH04-94-G-0327.

7 Bibliography

[1] T. Chen and J. Baer, “Reducing memory latency via non-blocking and prefetching caches,” In *Proceedings of the 5th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 51-61, Oct. 1992.

[2] M. K. Farrens and A. R. Pleszkun, “Improving performance of small on-chip instruction caches,” In *Proceedings of 16th Annual Symposium on Computer Architecture*, pp. 234-241, June 1989.

[3] G. F. Grohoski and J. H. Patel, “A performance model for instruction prefetch in pipelined instruction units,” In *Proceedings of the 9th International Symposium on Parallel Processing*, pp. 248-252, Aug. 1982.

[4] M. D. Hill, “A Case for Direct Mapped Caches,” *Computer*, pp. 25-40, 1988.

[5] M. D. Hill, *Aspects of Cache Memory and Instruction Buffer Performance*, Doctoral Thesis, Department of Computer Sciences, University of California, Berkeley, 1987.

[6] M. Horowitz, M. Martonosi, T. Mowry, and M. Smith, “Informing memory operations: Providing memory performance feedback in modern processors,” In *Proceedings of the 23rd Int. Symp. on Computer Architecture*, pp. 260 -270, May 1996.

[7] P. Hsu, “Design of the TFP microprocessor,” *IEEE Micro*, 1993.

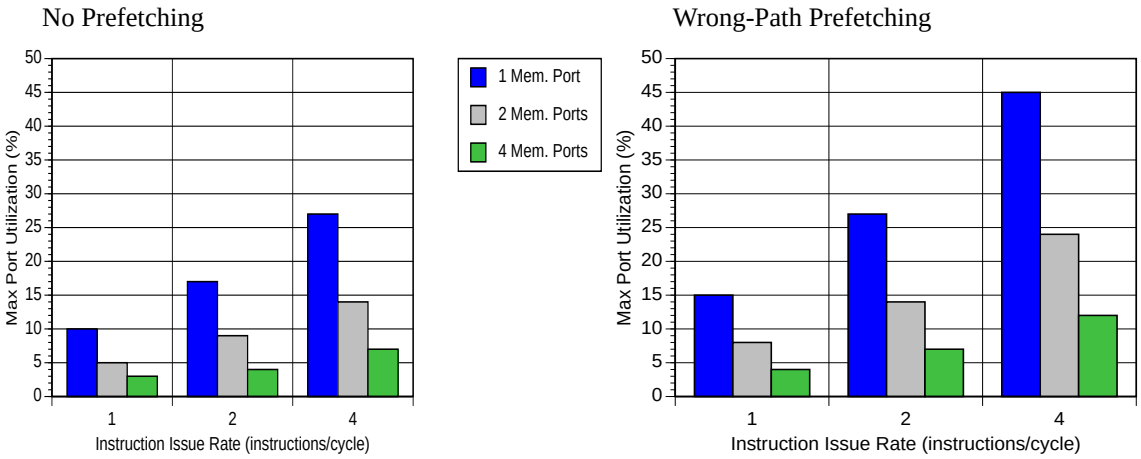

Figure 9 Port Utilization - These graphs show the percentage of cycles the busiest memory port spends transferring information during the execution of the system traces. The left graph is with no prefetching, the right graph with wrong-path prefetching.

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

[9] A. Klaiber and H. Levy, “An architecture for software-controlled data prefetching,” In *Proceedings of the 18th Int. Symp. on Computer Architecture*, pp. 43-53, May 1991.

[10] D. Kroft, “Lockup-free instruction fetch/prefetch cache organization,” In *Proceedings of the 8th Int. Symp. on Computer Architecture*, pp. 81-87, May 1981.

[11] D. Lee, J.-L. Baer, B. Calder, and D. Grunwald, “Instruction cache fetch policies for speculative execution,” In *Proceedings of the 22rd Int. Symp. on Computer Architecture*, pp. 357-367, June 1995.

[12] D. Nagle, R. Uhlig, and T. Mudge, *Monster: A Tool for Analyzing the Interaction Between Operating Systems and Computer Architectures*, Technical Report TR-147-92, The University of Michigan, 1992.

[13] J. Pierce, *Cache Behavior in the Presence of Speculative Execution—The Benefits of Misprediction*, Ph.D. Thesis, The University of Michigan, 1995.

[14] J. Pierce and T. Mudge, “The effect of speculative execution on cache performance,” *IPPS 94, Int. Parallel Processing Symp.*, Cancun Mexico, pp. 172-179, Apr. 1994.

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