A Performance Study of Instruction Cache Prefetching Methods

Wei-Chung Hsu and James E. Smith

Abstract—Prefetching methods for instruction caches are studied via trace-driven simulation. The two primary methods are "fall-through" prefetch (sometimes referred to as "one block lookahead") and "target" prefetch. *Fall-through prefetches* are for sequential line accesses, and a key parameter is the distance from the end of the current line where the prefetch for the next line is initiated. *Target prefetches* work also for nonsequential line accesses. A prediction table is used and a key aspect is the prediction algorithm implemented by the table. Fall-through prefetch and target prefetch each improve performance significantly. When combined in a hybrid algorithm, their performance improvement is nearly additive. An instruction cache using a combined target and fall-through method can provide the same performance as a two to four times larger cache that does not prefetch. A good prediction method must not only be accurate, but prefetches must be initiated early enough to allow time for the instructions to return from main memory. To quantify this, we define a "prefetch efficiency" measure that reflects the amount of memory fetch delay that may be successfully hidden by prefetching. The better prefetch methods (in terms of miss rate) also have very high efficiencies, hiding approximately 90 percent of the miss delay for prefetched lines. Another performance measure of interest is memory traffic. Without prefetching, large line sizes give better hit rates; with prefetching, small line sizes tend to give better overall hit rates. Because smaller line sizes tend to reduce memory traffic, the top-performing prefetch caches produce less memory traffic than the top-performing nonprefetch caches of the same size.

Index Terms—Instruction caches, prefetching, target prefetch, lookahead prefetch, scalar processors, supercomputers.

1 Introduction

Instruction cache misses can result in significant performance losses in highly pipelined processors. This paper studies hardware algorithms for prefetching instruction cache lines in order to reduce instruction fetch misses (in the best case) or, at least, to reduce the time penalty for a miss (when a line is prefetched, but not soon enough to result in an immediate hit). We consider the problem in the context of the scalar supercomputer pipelines and programs, but the studied methods can also be applied to pipelined RISC microprocessors.

The instruction fetch portion of the pipeline is shown in Fig. 1. Here, a cycle is spent accessing the instruction cache, the next cycle is spent extracting and decoding instructions, and the third cycle is spent resolving resource and data dependences and issuing the instruction to the functional units. If there is an instruction cache miss, there are, at most, two instructions that can issue before the pipeline is starved for instructions. In some implementations, buffers may be placed after the instruction cache to make the situation a little better by providing a few additional instructions. In many cases, however, the miss follows a taken branch, and the pipeline is essentially empty. After the miss is serviced, it will take two additional clock periods to decode and issue the missed instruction. Hence, almost the entire cache miss penalty is exposed; instruction issue is stopped for a time period approximately equal to the miss latency. This is in contrast to data cache misses, where instruction scheduling and/or hardware dynamic scheduling techniques can be used to hide part or all of the data cache miss penalty.

1.1 A Simple Performance Model

Based on the preceding discussion, we can derive a simple expression to approximate the amount of performance loss due to instruction cache misses. We measure performance as average *CPI* (clocks per instruction). Let *MPI* (misses per instruction) be the instruction cache miss rate, and let *CPM* (clocks per miss) be the instruction cache miss penalty. If CPI_{miss} is the average number of *CPI* that are lost due to instruction cache misses, then $CPI_{miss} = CPM * MPI$.

As an example, consider the CRAY Y-MP [2] where the *CPM* is 23 clock periods and a typical *MPI* is 0.02. Then, CPI_{miss} is about 0.46. Let CPI_{100} be the performance when the instruction cache hits 100 percent of the time. Because code that would normally have low CPI is often vectorizable, the CRAY Y-MP CPI, when doing scalar (nonvectorizable) code, is relatively high, often as high as three (see Table 1). If CPI_{100} is 3.0, the performance loss is $CPI_{miss}/CPI_{100} = 0.46/3.0$, or about 15 percent. Even if the MPI is as low as 0.005, the penalty CPI_{miss} is still 0.11, and the performance loss is about 4 percent. It is important to note that supercomputer memory latencies (in clock periods) have been gradually rising with successive generations so that reducing the instruction cache miss rate or hiding the instruction cache miss penalty becomes more important with each generation.

- *W.-C. Hsu is with Hewlett Packard Company, 19447 Pruneridge Ave., Cupertino, CA 95014.*
- *J.E. Smith is with the Department of Electrical and Computer Engineering, University of Wisconsin-Madison, 1415 Johnson Dr., Madison, WI 53706. E-mail: jes@ece.wisc.edu.*

Manuscript received 27 Aug. 1993; revised 20 Jan. 1998.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 106493.

TABLE 1
SUMMARY OF BENCHMARK PROGRAMS

program name	source language	No. of lines	cod size (words)	inst. exec. (millions)	CPI	CPI_{100}	HPM miss rate (%)	sim. miss rate (%)
BMK21A	Fortran	440	65,153	135	3.55	3.14	1.81	1.82
BMK5	Fortran	584	69,682	985	2.87	2.39	2.07	2.15
CFT77	Pascal	400,000	573,578	8	3.89	3.37	2.27	2.27
SPICEG	Fortran	18,960	93,793	1,284	4.67	4.27	1.75	1.77
HULL	Fortan	2,939	70,879	2,172	2.76	2.40	1.56	1.58
TIMBER	C	18,171	45,696	9,949	4.45	4.03	1.85	1.79
UOFA	Fortran	22,947	126,109	3,541	3.79	3.21	2.54	2.74
QCD1	Fortran	2,375	81,737	3,014	2.38	2.00	1.65	1.67
TRACK	Fortran	2,375	69,313	38	3.68	3.44	1.06	1.18
SCHED	Fortran	20,102	111,199	3,624	3.77	3.31	2.01	2.85

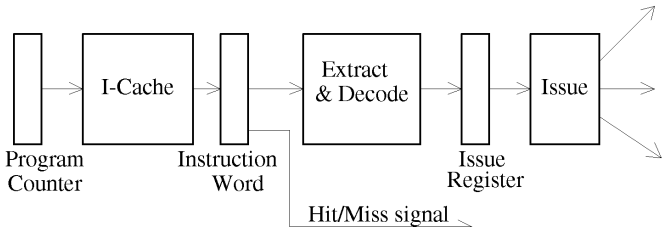


Fig. 1. Typical instruction fetch pipeline.

RISC microprocessors tend to have lower cache miss penalties, especially if a secondary cache is used. However, their *CPI* is also lower than in the scalar section of traditional supercomputers, like the CRAY Y-MP. For example, a RISC *CPM* may be eight clock periods, and *CPI* values are approaching 1.0 or even smaller with the latest superscalar processors. If *MPI* remains at 0.02, the performance loss is still about 15 percent. Hence, even though some RISC performance characteristics are different than supercomputers, the magnitude of the performance problem remains about the same.

Prefetching instruction cache lines can reduce the performance loss, but to hide the entire fetch penalty (i.e., to turn a miss into a hit), the prefetch must be initiated approximately CPM / CPI_{100} instructions in advance. Continuing the CRAY Y-MP example: If the CPI_{100} is three and the *CPM* is 23 clock periods, CPM / CPI_{100} is about eight instructions. Prefetches started fewer than eight instructions in advance may not completely avoid a miss. On average, they reduce the time penalty by an amount equal to the difference between the prefetch initiation time and the time of the cache miss.

1.2 Related Research

Cache prefetching (for both data and instructions) is discussed in depth in [12], and an updated summary is included in [14]. In these papers, it is suggested that "because of the need for fast hardware implementation, the only possible line to prefetch is the immediately sequential one." It is also stated that prefetching "when the last *n*th segment ($n = 1/2, 1/4$, etc.) of a line has been used is likely to be ineffective for reasons of timing: The prefetch will not be complete when the line is needed." We do not dispute these conclusions, given the types of systems and caches that were being considered. As we shall see, however, our conclusions are quite different because the properties of the systems and caches we study are different. In particular,

1) We concentrate on instruction caches only. In an instruction cache, there is a single reference stream, so nonsequential prefetches that follow a predicted flow of control may be quite effective.
2) Because instruction caches have high spatial locality, relatively large lines are useful. With larger lines, delaying prefetches into the last half or quarter segment of a line may be effective.
3) Today, hardware costs for nonsequential prefetch predictors and duplicate sets of cache tags are low. Duplicate cache tags allow candidate prefetches to test the cache without inhibiting normal references.

Nonsequential prefetch prediction is closely related to branch prediction [7], [13]. Traditional Branch Target Buffers (BTBs) holding branch targets are accessed using the addresses of branch instructions and are intended to allow an uninterrupted supply of instructions from the instruction cache to an instruction pipeline. In contrast, we use prediction based on instruction cache line addresses and the goal is to reduce instruction cache misses by prefetching into the cache. Furthermore, a cache line may have multiple "next" lines if it contains multiple branch instructions; hence, the problem is slightly more complex than in a BTB, where a branch instruction typically has only one target.

Recently, some superscalar microprocessors [3], [11] have begun to appear with next-line predictors that are intended to provide the same basic function as the BTBs, i.e., to provide smooth instruction flow into the pipeline. Because the focus is on a smooth flow from cache to pipeline, rather than prefetching from memory, these next-line predictors point to an instruction cache line rather than a memory address. By using next line predictors in this fashion, predictions do not have to wait until the actual branch instruction is decoded.

Other methods for reducing performance losses due to instruction cache misses have been proposed. In [9], it is proposed that the compiler can rearrange code to minimize misses in a direct mapped cache and to improve locality. A similar effort is reported in [5]. These methods rely on execution profile

data as a basis for packing basic blocks and procedures that are likely to executed consecutively into the same or sequential cache lines. Since such profile-based code repositioning improves spatial locality, it works more effectively with large cache lines or fall-through cache prefetching.

2 Instruction Cache Organizations

There are a number of cache characteristics that are related to the effectiveness of line prefetch methods. The specific relationships are discussed when we analyze simulation results. What follows is a brief review of some important cache organizations and characteristics.

An instruction cache holds a number of fixed-size blocks of instructions or lines. The CRAY Y-MP [2] instruction cache, which we use as a basis for simulations, contains four lines ("buffers" in Cray Research terminology) with 256 bytes (32 words)[1] per line. CRAY Y-MP instructions are 16, 32, or 48 bits long. Studies we have done show that 72 percent of the instructions are 16 bits, 15 percent are 32 bits, and 13 percent are 48 bits. So, instructions average about 22.5 bits each, or almost three per eight-byte word. We assume that when there is an instruction cache miss, the entire line is transferred from main memory to the instruction cache.

2.1 Associativity

There are three common ways to organize a cache. In a *fully associative cache*, any program line potentially can be found in any of the cache lines. In a *direct-mapped cache*, any program line can be placed in only one cache line, determined by address mapping. In a *set-associative cache*, cache lines are divided into *sets* and a program line may be within any of the cache lines in one set. For details, the reader is referred to any introductory computer architecture text.

2.2 Replacement Algorithms

When there is a miss in the cache, the program block containing the requested instruction is brought in from the memory, and it replaces some line in the instruction cache according to a replacement policy. In a direct-mapped cache, a program block can only be placed in one cache line, so there is only one possible replacement policy. In an n-way associative cache, different replacement policies can be considered. We consider the most commonly used strategies: LRU (Least-Recently-Used), FIFO (First In First Out), and random replacement.

2.3 Line Prefetch Strategies

We study two basic prefetch strategies that are aimed at different situations. The first is the situation where a consecutive series of lines are accessed as would be the case with straight-line code, or with code containing branches having short branch distances. In this case, we use a *fall-through* prefetch strategy, where the next consecutive line is fetched at some time before the end of the current line. Fall-through prefetch is referred to as "one block lookahead" in [14]. The other type of prefetch is directed at branches to nonconsecutive lines, as would be the case with procedure call/returns and branches closing large loops. We call this type of prefetching *target prefetching* and it has some characteristics of branch prediction. These two types of prefetching are not mutually exclusive, and we consider hybrid strategies that combine the two.

2.3.1 Fall-Through Prefetch

With fall-through prefetching, a prefetch request is initiated only when the instruction sequence falls within some specified distance from the end of an instruction line. We measure this distance in words, and define it as the *fetchahead distance*. In general, the closer the fetchahead distance is to the end of the line, the greater the chance that the prefetched line will be used. On the other hand, there is also a greater chance that the prefetched instructions will not return from memory soon enough to avoid any miss penalty.

2.3.2 Target Prefetch

Fall-through prefetch does not work well when a program contains a lot of small subroutine calls, a situation likely to increase with the use of C or object-oriented programming languages that emphasize procedure calls. To assist with prefetching in cases such as this, a straightforward approach is to implement a line target prediction table. For example, a simple table may have an entry for each active instruction cache line. The table entry contains the memory address of the line that most recently followed the active line. The instruction fetch unit can start a prefetch based on the prediction table contents and the current program counter value.

3 Simulation Benchmarks and Methodology

Trace driven simulation is used to study at a large number of instruction cache organizations and prediction methods. In this section, we describe the performance measures and benchmarks used.

3.1 Performance Measures

A primary performance measure of the cache simulations is the miss ratio. The *miss ratio* or *miss rate* is the ratio of instruction cache misses to instruction fetches.

Because prefetches sometimes are not initiated early enough to avoid an entire fetch latency on a miss, miss ratio measurements cannot be translated directly into performance gain/loss. The performance gain/loss is difficult to measure exactly in a trace driven simulation, unfortunately, so we define a measure, the *prefetch efficiency*, based on average performance information.

For each prefetched line, we keep track of the dynamic number of instructions between the time when the prefetch was initiated and the time the line was first referenced. Let this instruction count be N. Then, we define the prefetch efficiency of an individual prefetch as follows:

$$\text{If } N \geq \frac{CPM}{CPI_{100}}, \text{ the prefetch efficiency} = 1.0;$$

$$\text{If } N < \frac{CPM}{CPI_{100}}, \text{ the prefetch efficiency} = N * \frac{CPI_{100}}{CPM}.$$

We calculate prefetch efficiencies using CPM values taken from the CRAY Y-MP, i.e., 23 clock periods, and CPI_{100} values

1. Throughout this paper, a word is 64 bits.

taken from CRAY Y-MP Hardware Performance Monitor (HPM) numbers, deducting all instruction cache miss penalties. When $N \geq \frac{CPM}{CPI_{100}}$, the prefetch time would be completely hidden. If $N < \frac{CPM}{CPI_{100}}$, then $N \Big/ \frac{CPM}{CPI_{100}}$ or $N * \frac{CPI_{100}}{CPM}$ is the fraction of the miss penalty that is hidden. The overall prefetch efficiency is defined to be the average of the individual instruction prefetch efficiencies. It is the fraction of the miss penalty that is hidden by prefetching, averaged over all successful prefetches.

The above assumes that the miss penalty is independent of line size. We assume that the missed instruction is fetched first and that instruction processing can resume immediately when the first instruction is returned from memory. This technique is discussed in [1] and is a simplified form of "load forwarding" [4]. It significantly reduces the effect of line size on miss penalty and is used in the Cray Research processors and in many of today's microprocessors.

A secondary performance measure is memory traffic. *Memory traffic* is simply the total number of words fetched into the instruction cache. Some prefetch methods may increase the memory traffic and, for some system implementations, this could affect performance negatively.

Performance results were collected for each of the benchmarks and were averaged with the arithmetic mean. Because all the benchmarks are simulated for an equal number of instruction references, taking the arithmetic mean miss rate is equivalent to determining the miss rate of a job mix containing an equal number of instructions from each benchmark.

3.2 Benchmarks

We use 10 scalar benchmark programs in our trace simulation. Table 1 summarizes important characteristics of the benchmark programs. Among the 10 programs, BMK21A and BMK5 are from the LANL benchmark set [8], QCD1 and TRACK are from the Perfect Benchmark Suite [6], TIMBER is a circuit placement program, CFT77 is a Fortran compiler, HULL is a hydrodynamics code, SPICEG is a circuit simulation program, UOFA is a computational physics benchmark, and SCHED is a scheduling application.

As shown in Table 1, many of the benchmark programs execute more than one billion instructions. It would take considerable time to simulate all the cases we are interested in with full-length traces. Consequently, we decided to simulate small (four million instructions) portions instead of the entire programs. Of course, this poses the problem of accurately representing the whole program with a much smaller trace. To do this, we used our knowledge of the benchmarks to be sure that instruction tracing was initiated in the main body of the code (beyond any initialization portions). Then, to verify the accuracy of the approach, we simulated the selected four million instruction trace for the CRAY Y-MP with its native instruction cache organization, and executed the entire benchmark on a real CRAY Y-MP with the Hardware Performance Monitor (HPM) enabled. The results of the simulation and the entire execution are compared in Table 1. The miss rates for the four million instruction segments and the entire benchmarks are quite close.[2]

2. This does not guarantee accuracy, but it raises our confidence level considerably.

4 Simulation Results

If we consider wide ranges for key parameters, cache sizes, line sizes, associativities, replacement algorithms, and prediction strategies, then the number of different simulations is several thousand per benchmark. To keep the number of simulations manageable, we narrowed the ranges for some of the parameters. In particular, we consider only direct mapped and fully associative caches. These will bracket the performance of set associative caches. Furthermore, all Cray Research supercomputers to date have used fully associative instruction caches. And, as noted in the following subsection, after initial simulations, we narrowed the replacement algorithms for the fully associative caches to LRU. We consider caches of 128, 256, 512, 1,024, and 2,048 words (1 to 16 Kbytes) and line sizes of 16, 32, 64, and 128 words (128 through 1 Kbytes). These ranges are further narrowed for some of the specific simulations.

When we graph results for specific prefetch techniques, unless otherwise noted, we display performance for the best-performing line sizes. That is, we allow the line size to have the freedom to "float" to its best level for each cache configuration. Because cache lines provide some inherent prefetching due to spatial locality, if we were to use a fixed line size for all the prefetch methods, the prefetch aspect of the line size could hide the true contribution of the prefetch algorithm. In fact, determining the relative line sizes for which different prefetch methods are optimum is among our more significant results.

4.1 Performance with No Prefetching

First, we did some initial simulations without using prefetching. In these simulations we used direct mapped caches and fully associative caches with LRU, FIFO, and random replacement. We observed that the best miss rates occur with fully associative caches using LRU replacement; the only exception is the 2,048 word cache, where both random replacement and LRU give the same miss rate of 0.15 percent. Given this initial result regarding replacement algorithms, we decided to use only LRU replacement for associative caches in the remainder of our study.

Fig. 2 displays some of the results in graphical form. Fig. 2a is the miss rate for the fully associative cache with LRU replacement, Fig. 2b is the miss rate for the direct mapped cache, and Fig. 2c is the memory traffic for the fully associative cache with LRU replacement. We reach the following conclusions:

1) In general, miss rates are better with larger line sizes. The smaller caches perform best with 64 word lines, and the larger caches perform best with 128 word lines. Because there is no hardware prefetching, the larger line sizes give the effect of prefetching by exploiting spatial locality. The smaller caches work better with 64 word lines because cache pollution reduces performance with 128 word lines. *Cache pollution* occurs when lines containing useful instructions are thrown out in favor of lines which may contain fewer useful instructions.
2) For the better performing configurations, memory traffic tends to be quite high, as shown in Fig. 2c for

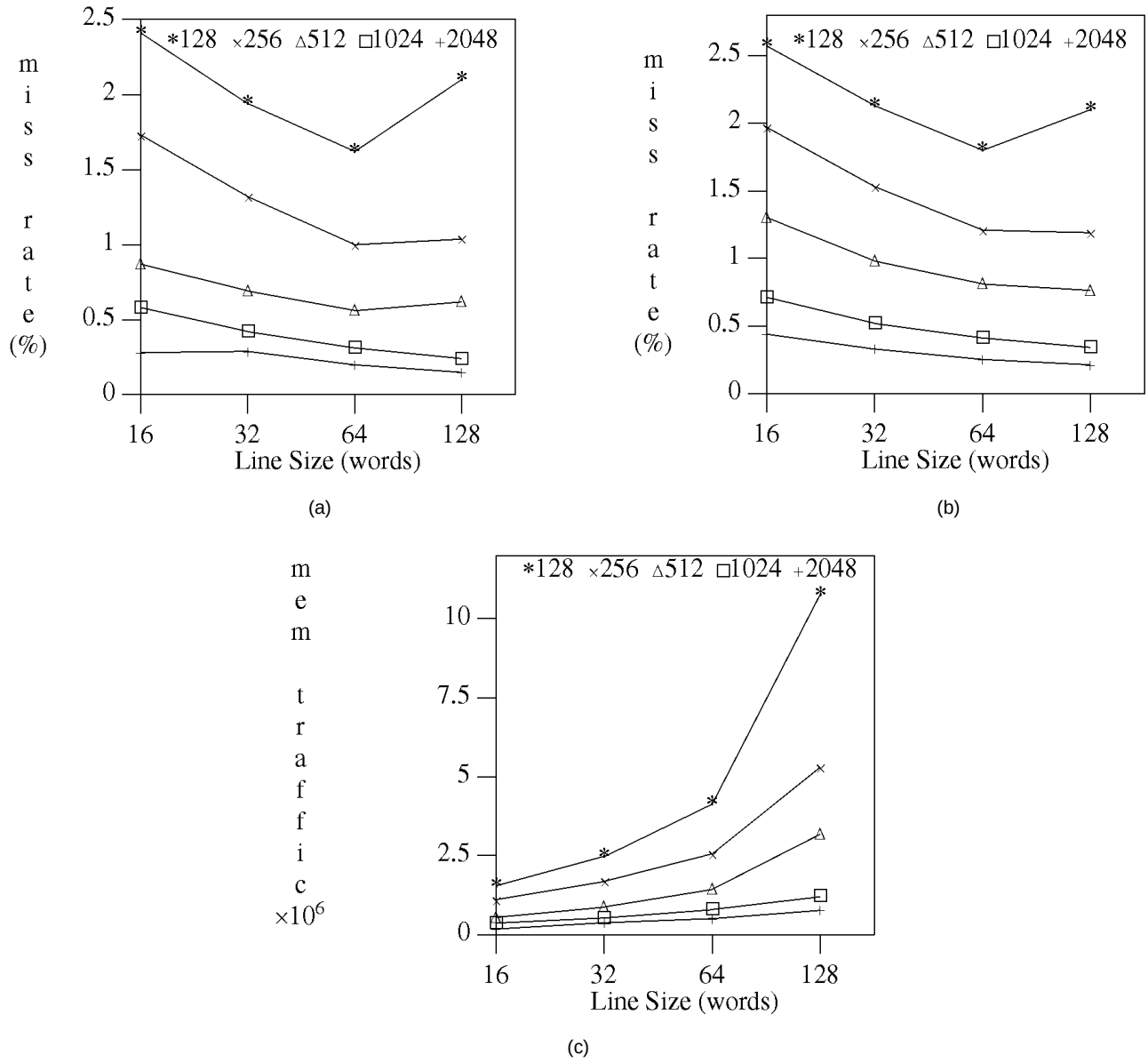


Fig. 2. Cache performance without prefetching: (a) miss rate for fully associative LRU cache, (b) miss rate for direct mapped cache, (c) memory traffic for fully associative LRU cache.

fully associative caches. The memory traffic for direct-mapped caches (not shown) is slightly higher. For example, the average number of words fetched for the better performing 128 word caches (i.e., with 64 word lines) averages over four million words per program, and only four million instructions are actually executed. Furthermore, memory traffic tends to track the line size; the larger the line size the more memory traffic. Taking Conclusion 1 into account, this means that, without prefetching, the better performing caches also require more memory traffic.

4.2 Fall-Through Prefetch

To study fall-through prefetch methods, we simulated fully associative (LRU) and direct-mapped caches with fetchahead distances of two, four, eight, and 16 words.

With only one exception, a fetchahead distance of eight was found to provide best performance (or is tied for best performance). The one exception occurs for the 2,048 word direct mapped cache, where the miss rate is slightly better for fetchahead 16 than for eight. Because it performs so well overall, we concentrate on eight word fetchahead in the remainder of the paper. We discuss reasons for the optimality of eight word fetchahead below. For fall-through prefetches, we make the following observations:

1) Overall, miss rates are improved significantly versus not prefetching. For both direct mapped and associative caches, the number of misses is reduced by about a third. This is illustrated in Fig. 3, which summarizes results for *all* the prefetch methods studied in this paper. For each cache size and prefetch method, the performance for the best-performing line size is given. The first and third curves from the top are for no prefetch and fall-through prefetch; other curves are for strategies yet to be described.

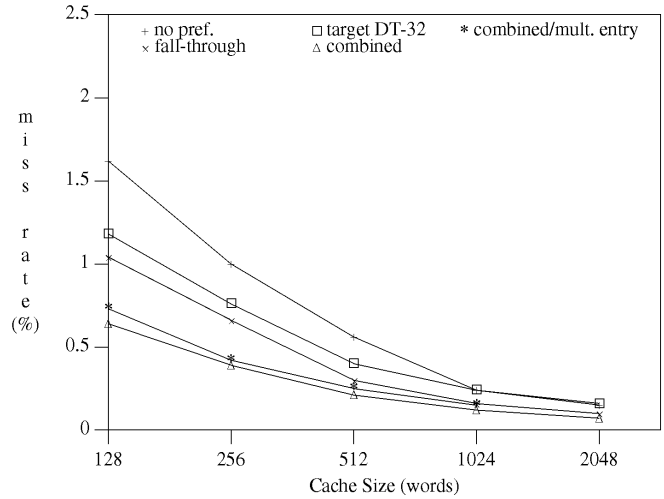
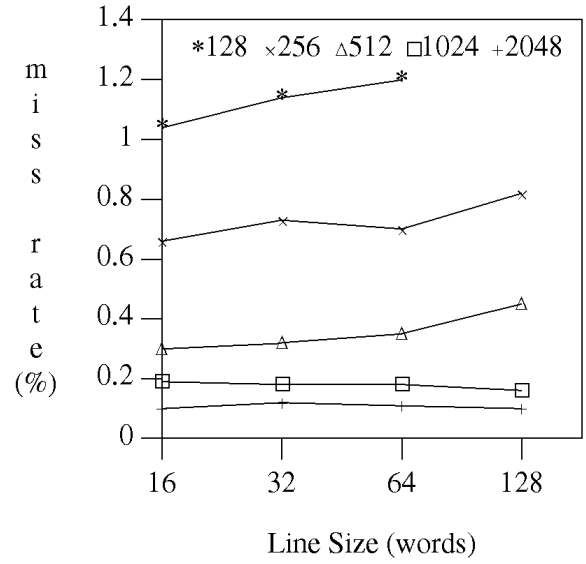


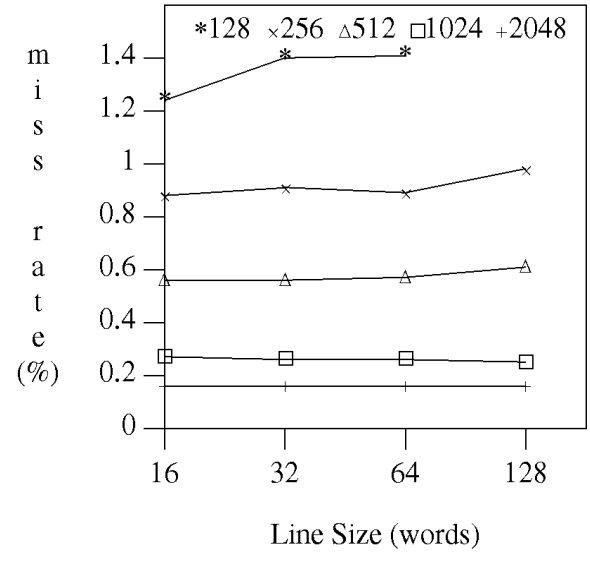
Fig. 3. Performance summary for all methods studied. Caches are fully associative; line sizes vary to give the best performance.

2) Larger line sizes are no longer an advantage, as was the case with no prefetch. This is illustrated in Fig. 4, where miss rates for fully associative LRU and direct mapped caches using fetchahead eight are graphed. In the few cases where a 128 word line performs better (for larger caches), the 16 word performance is extremely close. This is in contrast to the no-prefetch case, where large line sizes are better. Clearly, fall-through prefetch exploits the same spatial locality as large lines, so large lines are no longer as important when prefetching is used. Nevertheless, instruction cache lines must be large enough to realize a high prefetch efficiency.
3) Because smaller line sizes work better with prefetching, the memory traffic for top-performing prefetch caches is actually less than the memory traffic for the larger line size caches that perform best without prefetch. This is illustrated in Fig. 5, where the memory traffic values corresponding to the better performing miss rates for fully associative caches without prefetching and with fall-through prefetching are compared. For cases where there were ties for the best miss rate, the lower memory traffic values were used. The only anomalous case occurs for the 1,024 word cache. Here, the miss rates for the line sizes vary only between 0.19 and 0.16. The 0.16 case gives the memory traffic shown in the graph. The other cases are much better, with the 0.19 miss ratio giving memory traffic of 0.44×10^6. This data point is marked with a circle in Fig. 5.
4) High efficiencies correspond with low miss rates, a good situation indeed. For a full associative cache with an eight word lookahead, all the efficiencies are at least 92 percent. The direct mapped caches have efficiencies that are only 1 to 4 percent less.

As noted earlier, an eight word fetchahead distance generally works best. Intuitively, one might think that a shorter fetchahead may give a better hit rate (even though the efficiency might be lower). However, this is not necessarily the case. To illustrate this, consider two cases where a four word fetchahead prefetch may yield a different miss rate than an eight word fetchahead prefetch. Assume a 16 word line.



(a)



(b)

Fig. 4. Performance for fall-through prefetch: eight word fetchahead: (a) miss rate for full associative cache: eight word fetchahead, (b) miss rate for direct mapped cache: eight word fetchahead.

Case 1: There is a branch instruction between the eighth word and the 12th word that jumps to the following line. The eight word fetchahead prefetch will prefetch the following line, while the four word fetchahead (which is only triggered beyond the 12th word) does not.

Case 2: There is a long distance jump between the eighth and 12th words to some line other than the following line. The eight word fetchahead will initiate a not-effective (and cache polluting) prefetch. The four word fetchahead will not prefetch at all.

Case 1 would cause eight word fetchahead to perform better than four word fetchahead. Case 2 would cause four

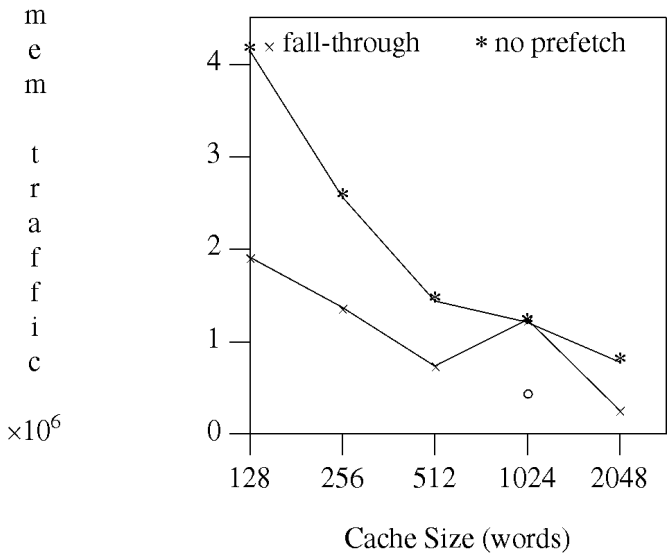


Fig. 5. Memory traffic for best performing full associative caches.

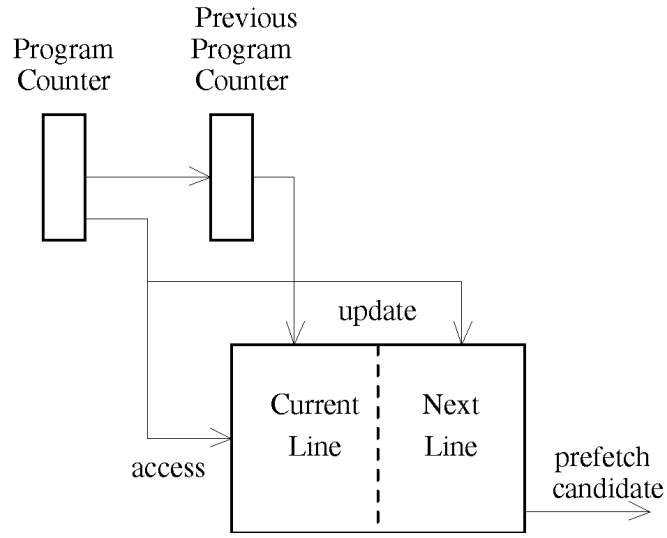


Fig. 6. Target prediction table.

word fetchahead to be better. Our results indicate that the performance gain due to Case 1 must be greater than the performance loss due to Case 2. For a larger fetchahead distance (16), the opposite is true.

4.3 Target Prefetch

Now, we consider the target prefetch approach. We first construct a next-line prediction table, as illustrated in Fig. 6. Conceptually, each entry in the table consists of a pair: (current line address, target line address). When the program counter changes to a new line, the prefetch unit uses the program counter's current line address to search the prediction table. If there is a match, then the target line address from the table is a candidate for prefetching; if the line is not already in the cache, it is prefetched. If there is a miss in the prefetch table, there is no prefetch request. When a cache miss coincides with a valid prefetch request, both the missed line and the prefetched line would be fetched from memory, in that order.

The table is updated whenever there is a line change. This requires the previous line address (from the previous program counter) and the current line address. If the missed line is the next consecutive line (as would be the case with a fall-through miss), the table is updated to reflect this.

A table as just defined must have a size (the number of entries; we assume a power of two) and a lookup algorithm. One possible lookup algorithm is a fully associative lookup, as is suggested by the above example. In this implementation, the entire current line address serves as a tag. One can also have set associative and direct mapped organizations. With associative tables, there is a choice of a replacement algorithm. The same choices typically used for caches are available here: LRU, FIFO, and random; in our simulations, we use random replacement for fully associative tables.

In a direct mapped cache, there should be logic to prevent a prefetched line from replacing the current line, i.e., prefetches that cause this type of conflict should be canceled. Also, in the case of a direct mapped prediction table, one has the option of dropping the tags altogether. In this case, there is always a hit in the prediction table, and the target line addressed by the current line is always prefetched (unless it is already in the cache). The "always match" assumption will never result in incorrect computations, but it may result in useless prefetches.

For our study, we consider three prediction table organizations:

1) fully associative with full tags,
2) direct mapped with tags,
3) direct mapped without tags.

We use the following shorthand notation: DT-n means Direct mapped with Tags and n table entries; DN-n means Direct mapped with No tags and n table entries.

To study target prefetch methods, we simulated all three of the above tables, each with eight, 32, and 128 entries. Cache miss rate and prefetch efficiency results are shown in Table 2. For each cache size and prediction table configuration, the best-performing (in terms of miss rate) line size is in bold. For target prefetching, we reach the following conclusions:

1) Overall performance is much better than no prefetch, and roughly the same as fall-through prefetch (depending on the prediction table used; for example, it outperforms fall-through prefetch when a large, fully associative table is used). For comparison with the other methods, Fig. 3 contains a series of points for target prefetch using a direct mapped prediction table with 32 entries (DT-32). We chose DT-32 because it is a practical implementation choice. Performance for other prediction table configurations are given in Table 2.
2) Optimal line sizes tend to be larger than with fall-through prefetch, although there are some exceptions. The optimal line size is usually 32 or 64 words. With fall-through prefetch there are many cases where the optimal line size is 16 words (see Fig. 4). Target prefetch does better with longer lines because of the spatial locality it gets; fall-through prefetch does not need the extra spatial locality.

TABLE 2
PERFORMANCE WITH TARGET PREFETCH

Miss Rate

cache & line sizes		Fully Associative Cache									Direct Mapped Cache								
		Fully Assoc Table			Dir. Table; tags			Dir. Tbl; no tags			Fully Assoc Table			Dir. Table; tags			Dir. Tbl; no tags		
		8	32	128	8	32	128	8	32	128	8	32	128	8	32	128	8	32	128
128	16	2.14	1.36	0.96	2.26	1.58	1.12	2.33	1.72	1.18	2.16	1.42	**0.99**	2.28	1.64	**1.14**	2.41	1.70	**1.16**
	32	1.49	**1.13**	**0.92**	1.66	**1.18**	**0.99**	1.78	**1.26**	**1.05**	1.65	**1.32**	1.12	1.80	**1.38**	1.19	1.87	**1.41**	1.19
	64	**1.35**	1.21	1.13	**1.39**	1.25	1.17	**1.47**	1.27	1.17	**1.50**	1.37	1.30	**1.55**	1.41	1.33	**1.63**	1.42	1.34
256	16	1.65	1.04	0.62	1.70	1.23	0.77	1.71	1.27	0.82	1.66	1.13	**0.70**	1.79	1.35	0.87	1.88	1.38	0.87
	32	1.08	0.76	**0.55**	1.17	0.81	**0.62**	1.23	0.91	**0.66**	1.20	0.94	0.73	1.35	0.99	**0.81**	1.42	**1.02**	**0.80**
	64	**0.81**	**0.72**	0.63	**0.85**	**0.76**	0.67	**0.92**	**0.78**	**0.66**	**1.00**	**0.89**	0.81	**1.05**	**0.93**	0.84	**1.12**	0.94	0.86
512	16	0.86	0.72	0.33	0.86	0.75	0.48	0.86	0.78	0.49	1.05	0.80	0.45	1.14	1.04	0.60	1.31	1.09	0.62
	32	0.65	0.45	**0.25**	0.65	0.50	**0.31**	0.69	0.56	**0.34**	0.78	0.62	**0.41**	0.85	0.67	**0.48**	1.20	0.72	**0.49**
	64	**0.47**	**0.36**	0.28	**0.51**	**0.40**	**0.31**	**0.58**	**0.46**	0.35	**0.65**	**0.54**	0.46	**0.69**	**0.58**	0.49	**0.74**	**0.62**	0.51
1,024	16	0.58	0.56	0.26	0.58	0.57	0.39	0.58	0.56	0.37	0.66	0.55	0.28	0.67	0.63	0.43	0.95	0.67	0.44
	32	0.41	0.34	0.14	0.42	0.36	0.20	0.42	0.36	0.21	0.46	0.37	**0.18**	0.47	0.42	0.25	0.96	0.45	0.27
	64	**0.30**	**0.20**	**0.12**	**0.30**	**0.24**	**0.15**	**0.31**	**0.24**	**0.16**	**0.35**	**0.26**	**0.18**	**0.36**	**0.29**	**0.21**	**0.77**	**0.32**	**0.22**
2,048	16	0.28	0.28	0.16	0.28	0.28	0.21	0.28	0.28	0.26	0.41	0.36	0.18	0.42	0.39	0.33	0.63	0.41	0.35
	32	0.29	0.26	0.11	0.29	0.27	0.15	0.29	0.27	0.15	0.29	0.23	0.10	0.30	0.26	0.16	0.40	0.28	0.18
	64	**0.20**	**0.13**	**0.07**	**0.20**	**0.16**	**0.09**	**0.20**	**0.17**	**0.10**	**0.22**	**0.14**	**0.09**	**0.23**	**0.19**	**0.11**	**0.27**	**0.22**	**0.14**

Prefetch Efficiency

cache & line sizes		Fully Associative Cache									Direct Mapped Cache								
		Fully Assoc Table			Dir. Table; tags			Dir. Tbl; no tags			Fully Assoc Table			Dir. Table; tags			Dir. Tbl; no tags		
		8	32	128	8	32	128	8	32	128	8	32	128	8	32	128	8	32	128
128	16	0.96	0.95	0.95	0.96	0.96	0.95	0.97	0.96	0.94	0.91	0.92	0.92	0.90	0.93	0.92	0.94	0.94	0.92
	32	0.97	0.96	0.96	0.96	0.97	0.97	0.96	0.97	0.97	0.94	0.95	0.94	0.96	0.95	0.94	0.96	0.95	0.95
	64	0.97	0.96	0.96	0.97	0.97	0.96	0.96	0.97	0.96	0.97	0.96	0.96	0.96	0.97	0.96	0.96	0.97	0.96
256	16	0.86	0.95	0.95	0.83	0.96	0.95	0.86	0.96	0.95	0.83	0.93	0.93	0.82	0.94	0.93	0.96	0.96	0.93
	32	0.97	0.96	0.95	0.87	0.96	0.96	0.98	0.97	0.96	0.91	0.94	0.94	0.81	0.94	0.94	0.92	0.94	0.94
	64	0.96	0.96	0.96	0.86	0.97	0.97	0.98	0.98	0.97	0.94	0.95	0.95	0.85	0.96	0.95	0.97	0.97	0.96
512	16	0.85	0.94	0.94	0.56	0.96	0.95	0.76	0.97	0.94	0.83	0.91	0.92	0.79	0.92	0.92	0.88	0.93	0.92
	32	0.85	0.95	0.95	0.84	0.96	0.96	0.98	0.96	0.95	0.93	0.93	0.93	0.83	0.94	0.93	0.91	0.94	0.94
	64	0.96	0.96	0.96	0.88	0.97	0.96	0.98	0.97	0.95	0.96	0.96	0.96	0.87	0.96	0.96	0.98	0.97	0.96
1,024	16	0.59	0.86	0.85	0.27	0.75	0.83	0.40	0.87	0.85	0.97	0.94	0.94	0.83	0.93	0.94	0.88	0.95	0.94
	32	0.77	0.86	0.85	0.56	0.80	0.85	0.66	0.86	0.86	0.95	0.95	0.95	0.87	0.95	0.94	0.86	0.96	0.96
	64	0.93	0.94	0.95	0.77	0.96	0.96	0.99	0.97	0.96	0.95	0.95	0.95	0.87	0.95	0.95	0.95	0.97	0.96
2,048	16	0.47	0.68	0.66	0.48	0.56	0.65	0.63	0.70	0.66	0.86	0.89	0.91	0.75	0.86	0.90	0.77	0.94	0.93
	32	0.46	0.68	0.66	0.28	0.61	0.66	0.30	0.69	0.67	0.83	0.94	0.93	0.81	0.86	0.91	0.93	0.96	0.95
	64	0.58	0.78	0.78	0.58	0.78	0.78	0.69	0.79	0.78	0.94	0.93	0.93	0.98	0.93	0.93	0.96	0.94	0.93

3) For smaller caches, the efficiencies (Table 2) are higher than fall-through prefetch with fetchahead of eight because prefetches of sequential lines will be initiated at the beginning of the line, rather than waiting until eight words from the end. Consequently, they get more of a head start. For larger caches, the efficiencies fall off, but this probably does not affect performance much because relatively few prefetches are made. For example, the 2,048 word fully associative caches with eight entry tables have efficiencies of only about 50 percent. However, they only make a few tens to a few hundred total prefetch requests.
4) Larger prediction tables are better. This is not a surprising result.
5) Fig. 7a shows the performance for fully associative caches using direct mapped tables with 32 entries. For each cache size, the best-performing line size is used. The performance difference between direct mapped caches with tags (DT-32) and without tags (DN-32) is larger for small tables than for large tables. Even for small tables, however, differences are small.

The direct prediction table without tags generates more memory traffic than with tags. This is expected because there is never a miss in the table, so it prefetches more often. This is illustrated in Fig. 7b, which compares memory traffic for the same cache configurations as in Fig. 7a.

4.4 Prefetching on Both Paths

We see that miss rates improved with both target prediction and fall-through prefetching. A natural extension is a method that combines both to achieve the advantages of each. This can be accomplished in the following way. When the instruction fetching switches to a new line, the instruction fetch unit searches the prediction table as with target prefetch. If there is a match, the target instruction line will be prefetched. In addition, if instruction fetching falls

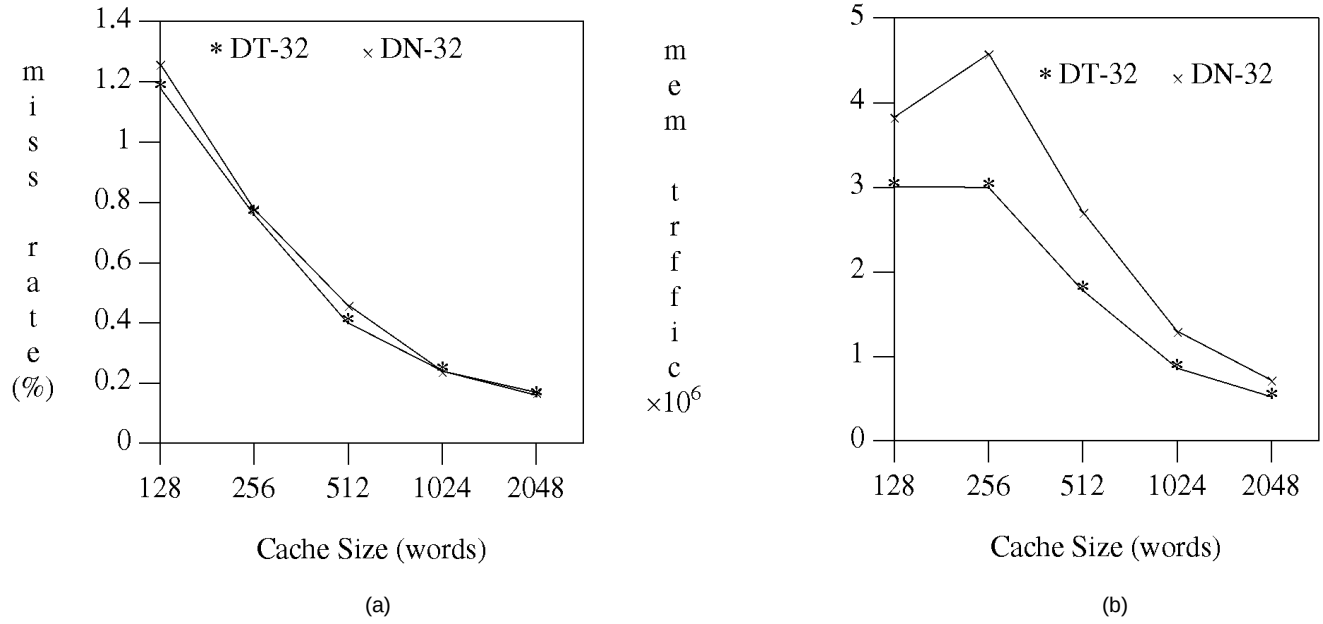


Fig. 7. Best performance for target prefetch: fully associative cache, D-32 prediction tables, with and without tags. (a) Miss rate using Direct-32 tables, with and without tags, (b) memory traffic with Direct-32 prediction tables, with and without tags.

within the fetchahead distance, the instruction fetch unit issues a fall-through prefetch. With this hybrid algorithm, fall-through lines do not need to be placed in the target prediction table, as they will likely be prefetched anyway if needed. This frees up table space for prefetch lines that are branch targets.

This approach appears to provide double protection against a line miss. A possible drawback is that it may tend to issue more unnecessary prefetches, thus increasing the memory traffic (although our results show that this is not the case). We simulated the same cache configurations as for target prefetch and used a fetchahead distance of eight for fall-through. Cache miss rate and memory traffic results are shown in Table 3. The best performing caches for a given cache size and prediction table configuration are in bold. We make the following conclusions:

1) Prefetching on both paths is much better than fall-through prefetch and target prefetch. Fig. 3 includes no prefetch, fall-through prefetch alone, target prefetch (DT-32) alone, and combined fall-through and DT-32 target prefetch. Improvements are almost as good as the sum of the two individual methods. As a specific example, consider a 128 word fully associative cache. With no prefetch, the miss rate is 1.62. With eight word fall-through prefetch, the best miss rate is 1.04 percent. With a DT-32 target prefetch table, the best miss rate is 1.18 percent. With the above prefetch methods combined, the miss rate becomes.64 percent. Thus, using fall-through prefetch improves the absolute miss rate by 0.58 percent versus no prefetch, target prefetch improves it by 0.44 percent, and both used together give an improvement of 0.98 percent
2) The fully associative prediction table is better than the direct mapped table with tags, and the direct table with tags is better than without tags. This was also the case for target prefetching. Now, however, the relative differences are somewhat greater.
3) Memory traffic (Table 3) correlates well with line sizes, and the better configurations have smaller line sizes, leading to reduced memory traffic. Furthermore, the memory traffic for prefetching both paths is not significantly larger than with the individual prefetch methods and, in some cases, is less.
4) As with the other prefetch methods, efficiency remains quite high, typically greater than 90 percent.

4.5 Multiple Targets per Line

Because the instruction cache lines we are considering are relatively large (at least 16 words), it is commonly the case that one instruction cache line may contain more than one branch or procedure call. Each such branch or call may have a different target line. In the prediction tables we have used thus far, however, a given instruction line can have at most one target entry in the prediction table.

We now consider tables that allow an instruction line to have multiple table entries. To implement such a table, we simply use a different set of program counter bits to address the table. In particular, we include bits that were formerly ignored because they were in the line offset field of the program counter. That is, when a line address is used, a line of size 2^n means that the lower n program counter bits would not be used (Fig. 8a). Now, to allow multiple table entries per line, we exclude only the lower two program counter bits (Fig. 8b). Because our program counter has a resolution of a quarterword (the smallest instruction size in the machine we are simulating), this means we access the table with a resolution of one word (up to four instructions). A direct mapped table with 2^m entries uses bits 2 through m + 1 to index into the table. A fully associative cache uses all the bits, beginning with bit 2 for tag bits. In the case of a direct mapped cache with no tag, we chose to hash, via XOR, the bits 2 through m + 1 with bits m + 2 to 2m + 1 (See Fig. 8c). The choice of excluding the lower two

TABLE 3
PERFORMANCE WITH COMBINED FALL-THROUGH AND TARGET PREFETCH

Miss Rate

cache & line sizes		Fully Associative Cache									Direct Mapped Cache								
		Fully Assoc Table			Dir. Table; tags			Dir. Tbl; no tags			Fully Assoc Table			Dir. Table; tags			Dir. Tbl; no tags		
		8	32	128	8	32	128	8	32	128	8	32	128	8	32	128	8	32	128
128	16	**0.75**	**0.55**	**0.46**	**0.80**	**0.64**	**0.51**	**0.83**	**0.71**	**0.57**	**0.89**	**0.69**	**0.61**	**0.93**	**0.79**	**0.65**	**0.99**	**0.81**	**0.67**
	32	0.86	0.76	0.72	0.96	0.82	0.74	1.11	0.92	0.78	1.13	1.04	1.00	1.19	1.05	1..02	1.33	1.13	1.05
	64	4.49	5.48	6.14	4.53	5.59	6.10	7.88	8.07	7.31	1.30	1.27	1.27	1.31	1.29	1.28	1.52	1.43	1.39
256	16	**0.49**	**0.31**	**0.24**	**0.52**	**0.39**	**0.28**	**0.53**	**0.41**	**0.30**	**0.62**	**0.43**	**0.36**	**0.66**	**0.54**	**0.40**	1.02	**0.55**	**0.42**
	32	0.54	0.42	0.39	0.58	0.47	0.41	0.66	0.55	0.47	0.72	0.62	0.59	0.75	0.66	0.61	**0.82**	0.74	0.65
	64	0.54	0.49	0.47	0.56	0.50	0.49	0.64	0.56	0.50	0.79	0.74	0.73	0.80	0.76	0.74	0.84	0.82	0.76
512	16	0.27	**0.15**	**0.10**	**0.27**	**0.21**	**0.13**	**0.28**	**0.25**	**0.16**	**0.42**	**0.30**	**0.23**	**0.44**	**0.36**	**0.26**	0.81	**0.37**	**0.27**
	32	**0.26**	0.17	0.16	0.28	0.22	0.17	0.34	0.32	0.23	0.43	0.34	0.32	0.45	0.38	0.33	0.75	0.48	0.37
	64	0.30	0.23	0.23	0.31	0.26	0.24	0.37	0.33	026	0.48	0.42	0.42	0.50	0.44	0.43	**0.53**	0.48	0.44
1,024	16	0.18	0.10	**0.06**	0.18	0.14	**0.08**	0.18	0.14	**0.08**	0.21	**0.13**	**0.09**	0.22	**0.17**	**0.11**	**0.58**	0.23	**0.12**
	32	0.17	**0.09**	0.09	0.17	0.13	0.10	**0.17**	**0.13**	0.11	**0.20**	**0.13**	0.12	**0.21**	**0.17**	0.13	0.60	**0.21**	0.17
	64	**0.15**	0.10	0.10	**0.16**	**0.12**	0.11	**0.17**	**0.13**	0.12	0.22	0.17	0.17	0.23	0.19	0.18	2.01	**0.21**	0.19
2,048	16	0.10	**0.06**	**0.04**	**0.10**	0.08	**0.05**	0.10	0.09	**0.05**	0.13	**0.06**	**0.04**	**0.13**	**0.09**	**0.06**	0.41	**0.12**	**0.06**
	32	0.11	**0.06**	0.06	0.11	0.08	0.06	0.11	**0.08**	0.06	0.13	0.08	0.07	0.14	0.10	0.08	**0.21**	0.16	0.11
	64	**0.09**	**0.06**	0.06	**0.10**	**0.07**	0.07	**0.09**	0.08	0.07	**0.12**	0.09	0.09	**0.13**	0.10	0.10	1.26	**0.12**	0.10

Memory Traffic (word × 10^6)

cache & line sizes		Fully Associative Cache									Direct Mapped Cache								
		Fully Assoc Table			Dir. Table; tags			Dir. Tbl; no tags			Fully Assoc Table			Dir. Table; tags			Dir. Tbl; no tags		
		8	32	128	8	32	128	8	32	128	8	32	128	8	32	128	8	32	128
128	16	2.08	2.19	2.25	2.09	2.16	2.23	3.27	3.50	3.19	2.31	2.43	2.50	2.28	2.40	2.49	3.64	3.84	3.47
	32	3.27	3.56	3.68	3.26	3.55	3.66	5.08	5.15	4.60	3.65	4.04	4.14	3.57	3.99	4.09	5.90	5.78	5.04
	64	32.3	40.5	45.7	31.9	41.1	45.2	57.8	59.7	54.4	6.49	7.10	7.27	6.39	7.07	7.18	10.3	9.69	8.77
256	16	1.40	1.45	1.47	1.40	1.43	1.47	1.88	2.33	2.17	1.67	1.77	1.85	1.67	1.74	1.81	3.00	2.89	2.58
	32	2.07	2.17	2.21	2.07	2.17	2.21	3.25	3.75	3.17	2.45	2.68	2.75	2.45	2.66	2.75	3.79	4.13	3.65
	64	3.59	3.95	4.07	3.59	3.93	4.04	5.18	5.30	4.56	3.85	4.12	4.33	3.82	4.18	4.29	6.03	6.69	5.82
512	16	0.75	0.78	0.80	0.75	0.78	0.80	0.90	1.38	1.34	1.07	1.12	1.15	1.07	1.10	1.14	2.22	2.05	1.82
	32	1.05	1.09	1.11	1.05	1.08	1.11	1.66	2.20	1.77	1.52	1.60	1.64	1.49	1.58	1.63	3.11	2.93	2.45
	64	1.85	1.97	2.02	1.80	1.97	2.01	2.87	3.06	2.51	2.47	2.62	2.77	2.41	2.67	2.73	4.12	4.13	3.55
1,024	16	0.44	0.45	0.47	0.44	0.45	0.45	0.46	0.59	0.65	0.58	0.60	0.61	0.57	0.59	0.61	1.46	1.42	1.26
	32	0.58	060	0.60	0.58	0.59	0.60	0.66	1.00	0.83	0.78	0.83	0.86	0.79	0.82	0.85	2.47	1.85	1.54
	64	0.85	0.90	0.93	0.85	0.89	0.92	1.25	1.43	1.32	1.32	1.41	1.54	1.25	1.48	1.52	11.80	2.56	2.41
2,048	16	0.26	0.27	0.31	0.26	0.27	0.30	0.27	0.35	0.43	0.35	0.35	0.39	0.34	0.35	0.36	1.12	0.82	0.92
	32	0.41	0.42	0.43	0.41	0.41	0.42	0.42	0.52	0.55	0.50	0.53	0.58	0.50	0.52	0.54	1.30	1.34	1.08
	64	0.54	0.57	060	0.54	0.56	0.58	0.60	0.77	0.76	0.82	0.88	1.01	0.75	0.95	1.00	7.77	1.87	1.87

bits was rather arbitrary and was based on the assumption that two branches or two calls in the same instruction word are less likely (such an assumption may not be true for processors with compare-and-branch instructions). Other sets of program counter bits could be used. We have not studied this further, however.

To study word-addressed tables, we used target prefetching in combination with fall-through prefetching because the previous simulations showed that combining the two methods is quite effective. We simulated the same set of configurations as in the previous subsection; the only change is word addressing of the tables.

For word-addressed prediction tables, we get mixed results with respect to line-addressed tables. Fig. 3 shows performance with the same 32 entry direct mapped prediction table as used in the curves for target prefetching and the combined method with a line addressed table. For the DT-32 prediction table configuration, the results are somewhat negative: The performance is slightly worse than with the combined method using line addressing. However, with the more expensive 128 entry fully associative prediction table, word addressing provides considerable performance improvement for small caches. This is shown in Fig. 9, where the two combined prefetch methods are compared. Both the line-addressed and word-addressed versions use 128 entry fully associative prediction tables. Here, the small cache performance is truly impressive: a 128 word cache can provide a miss rate of 0.19 percent. This rate of 0.19 percent is as good as the best 1,024 word cache with no prefetching. In this configuration, the prediction table can hold a prefetch prediction for virtually every branch in the cache.

Due to the cost of a large (128 entry), fully associative table, our results are somewhat negative, although such tables do permit small caches. For tables of a practical cost (small fully associative or direct mapped), there is no performance advantage. However, the performance gains due

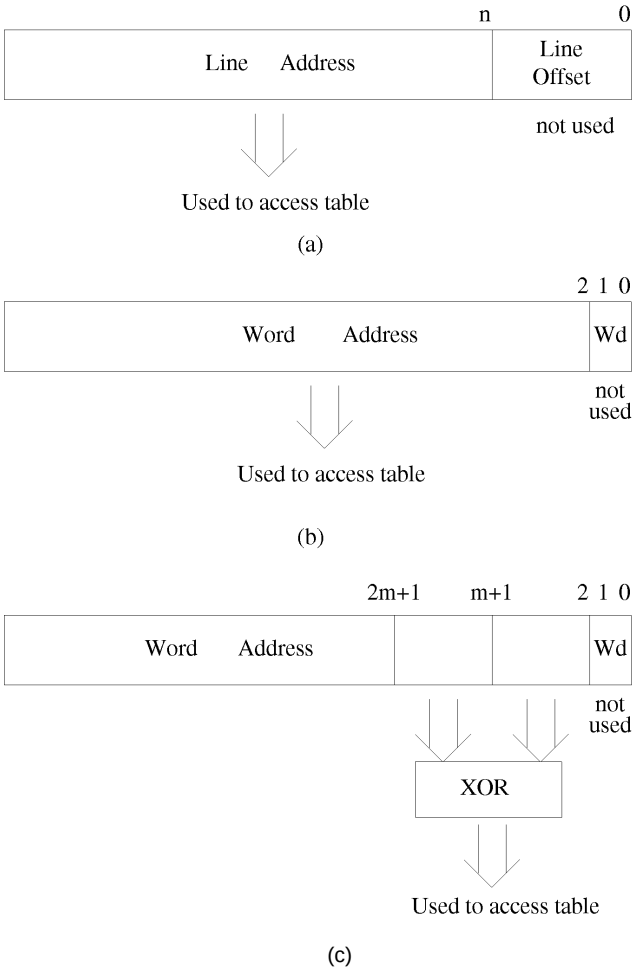


Fig. 8. Methods for addressing prediction tables: (a) line address, (b) word address, (c) word address with hash.

to the large associative tables do indicate that there is enough information available to the hardware to make significant performance improvements. The problem becomes one of extracting the information inexpensively. Perhaps an addressing resolution between line and word addressing or using a set-associative table would provide the high performance with a reasonable cost. We leave tables of this type for later study.

5 Summary and Conclusions

Refer once again to Fig. 3, the summary of all the prefetch methods. Our results show that instruction cache prefetching can cut miss rates by a factor of 2 to 3. The prefetch efficiency, i.e., the fraction of the miss penalty that is successfully hidden, is relatively high for the better performing cases—indicating that the relative miss rates are a fairly good measure of overall performance. Alternatively, one could conclude that a smaller cache with prefetch can replace a larger cache and get the same performance. This latter approach may be particularly good for supercomputers where memory bandwidth available for instruction fetches (especially for scalar code) is very high.

Of fall-through prefetch methods, eight word fetchahead works best. For 16 word lines, this means that one should

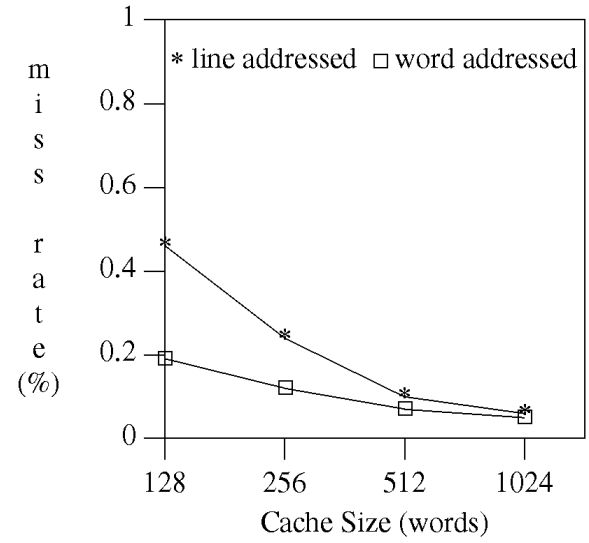


Fig. 9. Comparison of combined fall-through and target prefetch with line-addressed and word-addressed tables. Both use 128-entry full associative tables.

initiate a prefetch when instruction fetching is halfway through a line, rather than when entering the line. This contrasts with the conclusions reached in [14]; the difference is probably due to the large line sizes we use.

Target prefetching is useful both by itself and in combination with fall-through prefetching. When combined with fall-through prefetch, the performance benefits are additive.

Using word addresses leads to improved performance in some cases, mostly for large fully associative prediction tables. Tables of this type may lead to expensive implementations, however. At a minimum, our results for word-addressed tables indicate the high potential of hardware prefetch methods.

When comparing optimal caches with and without prefetching, prefetching leads to smaller line sizes, which cause a reduction in memory traffic. The optimal (in terms of miss rate) prefetch methods actually have less memory traffic than the optimal nonprefetch caches.

Efficiencies are uniformly high for the good prefetch methods. They are typically around 90 percent, indicating that an average of 90 percent of the memory latency for prefetching is hidden. Usually, when efficiencies are low, the number of prefetches being made is also low.

Finally, we observe that prefetch methods will be more important in the future for the following reasons.

1) Future high performance computers are facing the challenge of longer memory latencies. Both data references and instruction references suffer from the increased memory latency. But, unlike the data references, which can often be scheduled by programmers or compilers to tolerate long memory latency or through software controlled data cache prefetch [10], instructions usually depend on the instruction cache to overcome long memory latency.
2) With parallel processing techniques that spread loops over multiple processors, loop counts become reduced, so temporal locality is reduced.
3) Modern programming techniques yield more procedure calls and less spatial locality.

All of these mean that instruction caches in future high performance processors must be larger and/or smarter. Our results indicate that smarter caches, using hardware prefetching, yield significant improvements and should be considered as a viable design alternative.

ACKNOWLEDGMENTS

This work was done while the authors were with Cray Research, Inc.

REFERENCES

[1] W.Y. Chen et al., "The Effect of Code Expanding Optimizations on Instruction Cache Design," *IEEE Trans. Computers*, vol. 42, no. 9, pp. 1,045-1,057, Sept. 1993.
[2] "CRAY Y-MP System Programmer Reference Manual," Cray Research, Inc., 1988.
[3] L. Gwennap, "UltraSparc Unleashes SPARC Performance," *Microprocessor Report*, pp. 1, 6-9, 3 Oct. 1994.
[4] M.D. Hill and A.J. Smith, "Experimental Evaluation of Microprocessor Cache Memories," *Proc. 11th Ann. Symp. Computer Architecture*, pp. 158-166, June 1984.
[5] W. Hwu and P. Chang, "Achieving High Instruction Cache Performance with an Optimizing Compiler," *Proc. Third Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, 1989.
[6] D. Kuck et al., "The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers," CSRD report, Univ. of Illinois at Urbana-Champaign, 1988.
[7] J.K.F. Lee and A.J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *Computer*, pp. 6-22, Jan. 1984.
[8] O. Lubeck, J. Moore, and R. Mendez "A Benchmark Comparison of Three Supercomputers: Fujitsu VP-200, Hitachi S810/20, and Cray X-MP/2," *Computer*, Dec. 1985.
[9] S. McFarling, "Program Optimization for Instruction Caches," *Proc. Third Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, 1989.
[10] T.C. Mowry, M. Lam, and A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 62-73, 1992.
[11] M. Slater, "AMD's K5 Designed to Outrun Pentium," *Microprocessor Report*, pp. 1, 6-11, 24 Oct. 1994.
[12] A.J. Smith, "Sequential Program Prefetching in Memory Hierarchies," *Computer*, pp. 7-21, Dec. 1978.
[13] J.E. Smith, "A Study of Branch Prediction Strategies," *Proc. Eighth Ann. Symp. Computer Architecture*, pp. 135-148, 1981.
[14] A.J. Smith, "Cache Memories," *ACM Computing Surveys*, Sept. 1982.

Wei-Chung Hsu obtained the PhD degree in computer science from the University of Wisconsin-Madison in 1987. He is currently a senior engineer at Hewlett-Packard Company's California Language Laboratory. His research interests include high-performance computer architectures and optimizing compilers.

James E. Smith is with the Department of Electrical and Computer Engineering at the University of Wisconsin-Madison. He first joined the University of Wisconsin in 1976, after receiving the PhD from the University of Illinois. From 1979 to 1981, he took a leave of absence to work for Control Data Corporation in Arden Hills, Minnesota, participating in the design of the CYBER 180/990. From 1984 to 1989, he took a leave of absence to participate in the development of the ZS-1, a large scale scientific computer employing a dynamically scheduled, superscalar processor architecture. In 1989, he joined Cray Research, Inc., in Chippewa Falls, Wisconsin. While at Cray Research, he headed a small research team that participated in the development and analysis of future supercomputer architectures. In 1994, he rejoined the Department of Electrical and Computer Engineering at the University of Wisconsin, where he holds the position of professor. His current research interests focus on new paradigms for exploiting instruction level parallelism.