Exploring the limits of prefetching

P. G. Emma
A. Hartstein
T. R. Puzak
V. Srinivasan

We formulate a new approach for evaluating a prefetching algorithm. We first carry out a profiling run of a program to identify all of the misses and corresponding locations in the program where prefetches for the misses can be initiated. We then systematically control the number of misses that are prefetched, the timeliness of these prefetches, and the number of unused prefetches. We validate the accuracy of our approach by comparing it to one based on a Markov prefetch algorithm. This allows us to measure the potential benefit that any application can receive from prefetching and to analyze application behavior under conditions that cannot be explored with any known prefetching algorithm. Next, we analyze a system parameter that is vital to prefetching performance, the ***line transfer interval****, which is the number of processor cycles required to transfer a cache line. This interval is determined by technology and bandwidth. We show that under ideal conditions, prefetching can remove nearly all of the stalls associated with cache misses. Unfortunately, real processor implementations are less than ideal. In particular, the trend in processor frequency is outrunning on-chip and off-chip bandwidths. Today, it is not uncommon for processor frequency to be three or four times bus frequency. Under these conditions, we show that nearly all of the performance benefits derived from prefetching are eroded, and in many cases prefetching actually degrades performance. We carry out quantitative and qualitative analyses of these tradeoffs and show that there is a linear relationship between overall performance and three metrics: percentage of misses prefetched, percentage of unused prefetches, and bandwidth.*

1. Introduction

Historically the rate of increase of processor speed has exceeded the rate of increase of memory speed. Consequently, performance lost to cache misses has become more and more significant. Prefetching can mitigate this effect. Effective prefetching depends on being able to accurately predict the addresses sufficiently ahead of time and not causing cache pollution by replacing a more desirable line to accommodate the prefetched line.

In this paper we explore the limits of prefetching, independently of any specific algorithm, and generically characterize the prefetches made by any algorithm. First, we characterize a prefetching algorithm in terms of three metrics: the number of misses it is able to prefetch (coverage), the distance between a prefetch and the original miss (timeliness), and the probability that a prefetched line is used before being replaced (accuracy). In addition to these metrics, we analyze a system parameter that is vital to prefetching performance, namely the *line transfer interval*, which is the number of processor cycles required to transfer a cache line between different levels of the memory hierarchy. Additionally, we analyze application behavior and the potential benefits of prefetching that cannot be explored through any known prefetching algorithm—by increasing coverage, timeliness, and accuracy beyond any values that can be achieved today. Our goal is to understand the interplay among the different metrics and to quantify their effect on overall performance.

0018-8646/05/$5.00 © 2005 IBM

In real prefetching algorithms, the metrics work against one another. Aggressive prefetching, focusing on improving coverage, reduces the accuracy of a prefetch algorithm. Similarly, algorithms that improve the timeliness of prefetches may issue prefetches too soon and pollute the cache by replacing lines that are more desirable, thereby causing additional misses and decreasing the coverage. Finally, prefetching can also have unintended side effects. For example, prefetching reduces cache misses and can reduce application runtime; however, in doing so, prefetching can also generate additional misses. By removing memory stalls, prefetching allows the processor to run ahead and aggressively fetch and execute instructions along newer control flow paths (which were not explored previously because of stalls due to misses); in this process, new misses are generated. Our simulations show that new misses generated by this mechanism can account for up to 12% of the total number of misses! Ample bandwidth is required to cope with these additional misses caused by the above phenomena.

Specific prefetching algorithms have been evaluated in numerous other papers. Initial work in prefetching focused on identifying regular access patterns (i.e., constants or strides) for streaming applications and triggering prefetches using hardware- or software-based algorithms [1–5]. More recently, many more aggressive prefetch algorithms that predict nonsequential access patterns have been proposed. Some of them [6–14] focus on improving the prediction accuracy of these access patterns by either using confirmation (confidence) hardware counters or inserting prefetch instructions using profiling and compiler analysis to determine the data/control flow path of a program. Others [15–20] focus on improving prefetch timeliness by aggressively issuing prefetches with sufficient prefetch lookahead distance. We refer the reader to [21] for a survey of data prefetch algorithms.

The rest of the paper is organized as follows: Section 2 contains definitions and terminology. Section 3 describes performance metrics, and the prefetching model is described in Section 4. Section 5 describes our simulator and the benchmarks used in this study. In Section 6, we separately analyze the effects of timeliness, coverage, accuracy, and bandwidth. Section 7 covers prefetching when it is constrained by limitations inherent in real implementations. Results are summarized in Section 8.

2. Prefetching terminology

In this section, we define the metrics that we use to characterize prefetching: raw misses, good prefetch, bad prefetch, coverage, accuracy, timeliness, bus width, bus frequency ratio, and line transfer interval.

The number of *raw misses* for a given application when run on a processor is the number of misses generated without any prefetching algorithm.

Prefetches are broadly classified into two categories: *good* and *bad*. If a prefetched line is referenced by the application before it is replaced, the prefetch is called *good*; if the line is replaced before being referenced, the prefetch is called *bad*.

Coverage is the ratio of good prefetches G to the number of raw misses M. Since prefetching may generate new misses over and above the raw misses, having a coverage of unity does not imply that there are no misses.

Accuracy is the fraction of prefetches that are good. Poor accuracy results in more bus traffic, and in the possible replacement of live lines, hence in more misses. If there are G good prefetches and B bad prefetches, $Accuracy = G/(G + B)$.

We use prefetch distance to measure the *timeliness* of a prefetch, prefetch distance being the elapsed time between a useful prefetch initiation and the next use of the prefetched line; and we use the number of branches between a prefetch and its use to describe timeliness. Branches are chosen as the prefetch points (where a prefetch is issued) because they represent points of uncertainty in the instruction flow and often block or hinder the movement of prefetches back in the instruction stream to establish an earlier prefetching point. This can occur for both hardware and software prefetching algorithms. Prefetch distance can easily be converted to time, measured in cycles, by knowing the number of cycles per instruction and the average number of branches per instruction. As timeliness increases from zero, more memory latency is hidden. When all of the latency becomes hidden, further increasing the timeliness can be detrimental.

The *bus width* is the number of bytes moved per bus cycle. This is determined by the number of wires between the cache and the memory subsystem, which is determined by packaging and power constraints. The ratio of the cache line size to the bus width is the number of packets in a line, which is also the number of bus cycles required to move a line between levels of the memory hierarchy.

The *bus frequency ratio* is the ratio of the processor frequency to the bus frequency. It is determined by logic speed, packaging, and power limits. The *line transfer interval* (LTI) is the number of processor cycles needed to move a cache line (also called a "block") on the bus and is equal to the bus frequency ratio times the number of packets.

3. Performance metrics

The most commonly used metric for processor performance is *instructions per cycle* (IPC). In this paper,

we use its inverse, *cycles per instruction* (CPI). As shown in [22], the *number of cycles per instruction*, or CPI, is calculated directly as a product of event frequencies and their corresponding event delays. Since the event frequencies and delays are measured in our simulations, CPI follows directly from these measurements.

CPI for a processor system can be decomposed into two components: an *infinite cache* (CPI_{INF}) and a *finite cache adder* (CPI_{FCA}):

$$CPI = CPI_{\mathrm{INF}} + CPI_{\mathrm{FCA}}.$$

CPI_{INF} represents the performance of the processor in the absence of misses (even compulsory misses). It is the limiting case in which the processor has a cache that is infinitely large. CPI_{FCA} accounts for the delay due to cache misses and is used here to measure the effectiveness of prefetching.

Without prefetching, it accounts for the delay caused by the total number of misses (defined as *raw misses* in the previous section). Just as processor performance (for both in-order and out-of-order machines) can be expressed in terms of a CPI value, the *memory adder* can be expressed as the product of an event rate (specifically, the miss rate) and the average delay per event:

$$CPI_{\mathrm{FCA}} = \left(\frac{misses}{instruction}\right)\left(\frac{cycles}{miss}\right). \quad (1)$$

Cycles per miss is an average processor delay caused by each cache miss that we calculate through simulation over a run without cache misses, as illustrated through the following example: Consider a trace tape one million instructions long and a processor organization where each cache miss has a 20-cycle miss latency. If an infinite cache simulation run requires one million cycles ($CPI_{\mathrm{INF}} = 1$) and a finite cache simulation run requires 1.3 million cycles, cache miss stalls account for 300,000 cycles, and the total $CPI = 1.3$ and $CPI_{\mathrm{FCA}} = 0.3$. If the finite cache simulation run generated 25,000 misses, $(misses/instruction) = (25{,}000/1{,}000{,}000) = 1/40$ and $(cycles/miss) = (300{,}000/25{,}000) = 12$. Note that the $(cycles/miss)$ term can be significantly less than the miss latency. Obviously, an out-of-order processor can account for this difference by overlapping useful work during a cache miss.

Effective prefetching reduces CPI_{FCA}, and if prefetching reduces CPI_{FCA} to zero without introducing undesirable side effects (bad prefetches and new misses), the processor will appear to run at infinite cache speed: $CPI_{\mathrm{FCA}} \rightarrow 0$ *and* $CPI \rightarrow CPI_{\mathrm{INF}}$.

When prefetching is introduced, a second term is added to the memory adder equation: CPI_{FCA}. The misses in Equation (1) are now subdivided into two terms: prefetches and remaining misses. Thus, some of the misses (raw misses) from (1) are avoided (prefetched) and transferred to a prefetch term. The expression for CPI_{FCA} becomes

$$CPI_{\mathrm{FCA}} = \left(\frac{misses}{instruction}\right)\left(\frac{cycles}{miss}\right) + \left(\frac{prefetches}{instruction}\right)\left(\frac{cycles}{prefetch}\right). \quad (2)$$

Here, the misses-per-instruction term represents the total number of misses that occur after a prefetch algorithm is used; the prefetches-per-instruction term represents the average rate at which prefetches are issued. However, not all prefetches are used, and prefetching also causes some new misses to occur. Therefore,

$$\left(\frac{misses}{instruction}\right) + \left(\frac{prefetches}{instruction}\right) \geq \frac{rawmisses}{instructions}. \quad (3)$$

Also, in (2) the cycles-per-prefetch term defines the processor delay caused by each prefetch—and can again be calculated through simulation in a manner similar to the cycles-per-miss term. For timely prefetching, the delay incurred for prefetches must be less than the delay for misses. The more timely the prefetching, the stronger this relationship. Thus,

$$\left(\frac{cycles}{prefetch}\right) \leq \left(\frac{cycles}{miss}\right). \quad (4)$$

For CPI_{FCA} to approach zero, both terms in (2) must approach zero. This occurs when all misses are prefetched $[(misses/instruction) \rightarrow 0]$ and prefetched sufficiently far in advance of the miss to remove all delay $[(cycles/prefetch) \rightarrow 0]$. Here, we explore the limits of prefetching to determine the conditions under which $CPI_{\mathrm{FCA}} \rightarrow 0$.

Unused prefetches also add delay. They can increase both the $(cycles/prefetch)$ and $(cycles/miss)$ terms. A bad prefetch can add delay in four ways: a) It may interfere with processor accesses as it is written into the cache; b) it will use the bus for a line transfer interval, during which it may impede other miss and prefetch traffic; c) it may saturate the memory hierarchy subsystem and delay the initiation of a new miss or prefetch; and d) it may replace a useful line in the cache and cause an additional miss.

4. Prefetching model description

Trace-driven simulation is commonly used for evaluating processor systems. We use such a simulation in this study in a manner that is synergistic with our prefetching model. Specifically, we use a miss file that is collected from an initial simulation run to manipulate the prefetching timeliness, coverage, and accuracy directly as we rerun the trace. The manner in which we do this allows

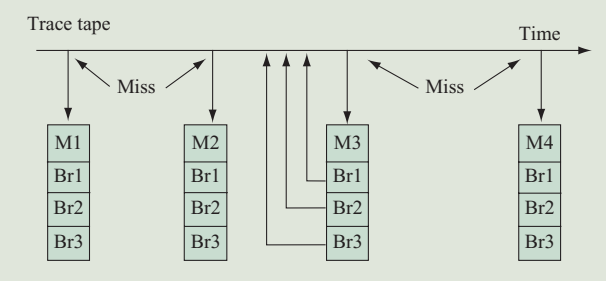

Figure 1

Illustration of prefetching model data collection. Whenever a miss occurs, the address of the miss along with the instruction numbers of *n* previous branches (Br1, Br2, ...) are written to a miss file. M1, M2, ... denote corresponding miss addresses.

us to vary these parameters independently, as explained next.

Within the simulator, we first configure the details of the processor to be studied (set cache parameters, pipeline characteristics, etc). We start with prefetching turned off, and run a trace through the simulator to produce a file of the *raw misses*, as defined in Section 3. The raw-miss file contains a list of the addresses of all L1 cache misses (both instructions and data) that occurred in the simulation in the order in which they occurred. This can be thought of as the set of *necessary* accesses made by the program for this processor organization.

Once a miss is recorded, we identify certain events prior to the miss that we will use to issue a prefetch. The distance between the prefetch and the miss is referred to as the prefetch distance. There are many ways to measure this distance: the number of cycles, the number of intervening branches, the number of intervening instructions, or the number of intervening misses. Each measure is valid and conveys unique information. We use the number of intervening branches between the prefetch and the original miss to denote the prefetch distance. Specifically, by expressing this interval in the number of intervening branches, we are able to impart a level of uncertainty between the prefetch and the miss due to possible branch prediction error. (We feel that expressing the prefetch distance in number of instructions or number of cycles does not capture the degree of uncertainty that can exist between the prefetch and miss.) Thus, if a prefetch must be executed many branches in advance of a miss in order to be sufficiently timely, this poses a real challenge to the design of a prefetching mechanism (hardware or software); i.e., to be able to accurately anticipate the correct program flow and calculate the correct prefetch address.

In addition to collecting all raw misses, with each miss in the file we include a record of the position of the instruction of the *n* branch instructions that precede the miss, as shown in **Figure 1**. The *n* branches preceding a miss are used as the set of possible prefetch trigger points for the miss. Branches are chosen as the prefetch points because they are the logical barriers to moving prefetches back in the instruction stream to establish an earlier prefetching point; i.e., they are points of uncertainty in the instruction flow.

In this study, timeliness (prefetch distance) is varied by moving prefetches back by a fixed number of branches; i.e., for the *n* branches preceding each miss in the miss file, we study the effect of increasing timeliness by rerunning the simulation *n* times with prefetching turned on. On the *i*th run, we issue prefetches *i* branches ahead of the miss. This is done by matching instruction numbers from the miss file with the corresponding instructions in the trace.

For example, on the first run, a prefetch for a miss is issued when the branch instruction that immediately precedes that miss is encountered. On the second run, a prefetch is issued when the second branch preceding the miss is encountered, and so on. In this study, we use $n = 20$, which gives us a very wide range of prefetch timeliness.

Since the prefetch distance is measured in the number of branches, it is not uniform in time. First, the number of instructions between successive branches is not a constant, but has a distribution that we show later. Second, the rate of instruction processing (instructions per second) varies. Nonetheless, we use number of branches as the unit of timeliness because the coverage and the accuracy of any prefetch algorithm are directly related to the degree of certainty in the future instruction flow—which is determined largely by the branches.

Using the miss file to generate prefetches guarantees that we can generate prefetches that match misses from the original run. The prefetches are issued at a specified time, and the penalty attributed to each miss (cycles per miss) depends on the pipeline dynamics. The burstiness of misses is preserved in the miss file and is carried over into the prefetching runs. If the miss rate is bursty, the prefetch rate will be just as bursty.

We vary the *coverage* and the *accuracy* of prefetching probabilistically. Specifically, *coverage* and *accuracy* are inputs to the simulator. For each miss in a run, we generate its corresponding prefetch with a probability equal to the desired *coverage*. To achieve a specified *accuracy*, we generate a bad prefetch (to a set of known unused addresses for each application) with the appropriate probability each time we generate a good prefetch. For example, to achieve an accuracy of 50%, we generate one bad prefetch for every good prefetch. This allows bad prefetches to occur in regions that are just as bursty as the original miss rate.

We use this methodology to analyze both hardware and software prefetching algorithms. When evaluating a hardware prefetching algorithm, our software simulator (timer) is supplied with coverage, accuracy, and timeliness parameters and prefetches are issued on the basis of these inputs. However, when studying a software prefetching algorithm, the effects of adding prefetching instructions to the program are not explicitly modeled; i.e., the effects of inserting prefetching instructions into a program and thus increasing the footprint (binary size) of the program are considered minimal. Prefetches are issued at the specified prefetching distance supplied to the simulator. Also, since prefetching timeliness is specified as the branch distance between the prefetch and its use, this distance is preserved for software prefetching because prefetch instructions are not considered branches.

Finally, we study the effects of bandwidth by varying the *line transfer interval*, which is done by varying the *bus frequency ratio*, the cache *line size*, and the *bus width*. These three parameters are inputs to the simulator, and the line transfer interval is calculated directly from them.

Note that changing the cache line size causes a different set of raw misses to occur. Therefore, simulation runs using different line sizes use different raw-miss files (as do simulation runs using different cache sizes). Typical cache line sizes are 32, 64, 128, and 256 bytes. For the runs carried out in this study, we focus mainly on a 128-byte line size.

Bus frequency is generally one, two, three, or four times slower than the processor, and could be larger in the future. The ratio increases with the processor frequency because the bus frequency does not tend to keep pace. In most of this study, we set the bus frequency ratio to unity. Thus, we vary the line transfer interval merely by varying the bus width. This can be changed from run to run, and it does not require a new miss file (although bandwidth effects may cause the order of misses to vary slightly). In the last section, we study the effects of setting the bus frequency ratio to 2, 3, and 4.

We use bus widths of 16, 32, 64, and 128 bytes. For a 128-byte line size and a bus frequency ratio of unity, we obtain line transfer intervals of 8, 4, 2, and 1 processor cycles, respectively. When the bus frequency ratio is 2, the respective line transfer intervals are 16, 8, 4, and 2 cycles. Recall that for a fixed number of misses, the bus utilization is proportional to the line transfer interval.

5. Simulation methodology

We use a proprietary, cycle-accurate simulator and traces produced for IBM zSeries* processors in this study. The simulator has three input files: 1) a *design parameter* file, which includes the characteristics of the processor and cache being simulated (including cache line size, bus width, bus frequency ratio, a prefetch on/off switch, and the desired prefetch coverage, accuracy, and prefetch distance); 2) a *raw-miss* file containing all instruction and data misses and corresponding prefetching points for each miss (which is only used when prefetching is turned on); and 3) a trace of the application code being studied. When prefetching is turned off, the simulator produces the raw-miss file. The simulator always produces a summary file of performance statistics (miss rates, cycles per instruction, etc.) including all of the statistics shown in this paper.

Table 1 Workload suite.

Application	*Suite*	*Description*
mcf	SPEC CPU2000	Combinatorial optimization
compress	SPEC CPU95	Data compression
oltp	Transaction processing	Database workload
perf1	Workstation	Machine design (simulator)

We set the *design parameter* input file to model a four-issue, superscalar processor with an instruction window of 16, up to eight outstanding loads, four functional units, a pipeline depth of nine cycles, and separate L1 instruction and data caches of sizes 32 KB, 64 KB, and 128 KB. Each cache is fed by a separate bus connected to backing memory, the L2. Miss latency is measured in terms of the number of cycles between detecting a miss at the L1 cache and receiving the first datum (packet) from the L2 for the miss. All misses are resolved in the L2.

We studied seventeen workloads from SPEC** CPU95, SPEC CPU2000 (or simply "SPEC95" and "SPEC2000")[1], database workloads, and C++ applications. Because of space limitations, we present (graph) the results for only the four workloads of **Table 1.** The workload *mcf* is from the SPEC2000 suite, *compress* is from the SPEC95 suite, *oltp* (online transaction processing) is a workload originally written in IBM System/360* assembler language, and *perf1* is a proprietary workload (a large-processor simulator written in C++). These workloads represent the typical results (trends) observed in all of the larger set of workloads studied.

As mentioned earlier, the interbranch distance is not constant. **Figure 2** shows the distance between a prefetch and its use (a miss) for the *perf1* workload. Shown are the fractional number of misses (y axis) having a given number of instructions (x axis) between the prefetch and

[1]SPEC CPU95 and SPEC CPU2000 are performance-measuring suites for comparing performance across different hardware platforms. They were developed by the Standard Performance Evaluation (SPEC) Corporation, a nonprofit group located in Fairfax, VA (*http://www.spec.org*).

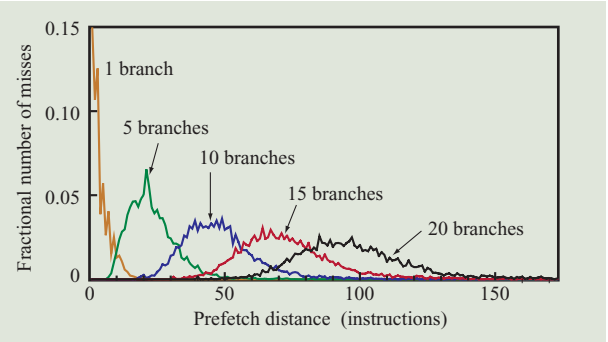

Figure 2

Distribution of distances between cache prefetches and subsequent use of the data for various branch prefetch distances. Data is for the *perf1* workload and a 64-KB cache with a 128-byte line size.

the miss. The associated distributions are for 1, 5, 10, 15, and 20 branches ahead of the miss.

These distributions are approximately Gaussian. Since the number of instructions between branches is (not quite) 5, each addition of five branches to the prefetch distance shifts the distribution (not quite) 25 instructions to the right. As the number of branches increases and the distribution shifts to the right, it also spreads.

The distribution for prefetch distance of ten branches shows that all prefetches will be issued at least 20 instructions ahead of a miss, roughly half of the prefetches will be issued 45 instructions ahead of a miss, and a very few prefetches will be issued 100 or more instructions ahead of a miss. **Table 2** shows the mean and standard deviation for these prefetch distances (1, 5, 10, 15, and 20) for each of the four workloads studied.

Real prefetching mechanisms have difficulty in generating accurate prefetches at large prefetching distances; uncertainty in the program flow and possible branch prediction error add to this difficulty. The branch prediction accuracy for *perf1, oltp,* and *compress* is slightly over 80%, while *mcf* has a 90% branch prediction accuracy. A 16K-entry branch-target-buffer is used in all simulation runs. Thus, *perf1, oltp,* and *compress* have, on the average, one mispredicted branch for each five branches that a prefetch is issued ahead of its use. That is, at a prefetch distance of 5, 10, 15, and 20 branches, *oltp, perf1,* and *compress* have nearly one, two, three, and four mispredicted branches, respectively. Similarly, *mcf* has, on the average, one and two mispredicted branches at a prefetch distance of 10 and 20 branches. Clearly, a real prefetching algorithm must trade off increasing prefetching timeliness (issuing prefetches at large distances) versus its ability to issue accurate prefetches even when the exact program flow is in doubt. However, in our simulation we specify the accuracy of a prefetching run through an input parameter. We examine the effects of coverage, accuracy, timeliness, and bandwidth on performance for the four applications discussed above. We select coverage and accuracy values between 50 and 100% at various prefetching distances and bandwidths. This allows us to study the effectiveness of prefetching independently of any particular or even existing prefetching algorithm.

Two popular prefetching algorithms are next-sequential prefetching (prefetching the next line) and Markov prefetching [8]. **Table 3** shows coverage and accuracy measurements (in percent) for these two algorithms for prefetching instructions and data. The cache modeled is 64 KB in size, four-way with a 128-byte line and a one-cycle LTI. The processor organization is four-issue, out-of-order, with a nine-stage pipeline. Instruction prefetching coverage and accuracy figures for *mcf* and *compress* are omitted because they have a small instruction footprint and have only compulsory misses.

Before prefetching results are presented (in the next section), it is useful to analyze the difference between the prefetches issued by a specific prefetching algorithm and those issued using our probabilistic treatment of prefetching. Consider two random processes P_1 and P_2. Let each random process select n elements from a total population of N objects, denoted as set n_1 and n_2. The cardinality of each set is denoted as $||n_1||$ and $||n_2||$, respectively. Then $(||n_1||/N) = (||n_2||/N) = C$ (coverage),

Table 2 Mean and standard deviation for instruction distance between prefetch and original miss. Prefetch distance is measured in numbers of branches.

Prefetch distance	*mcf*	*compress*	*oltp*	*perf1*
1	3.8 ± 2.8	1.5 ± 1.8	3.2 ± 4.8	2.7 ± 3.4
5	19.1 ± 5.4	25.3 ± 10.9	21.1 ± 11.2	23.1 ± 10.2
10	32.2 ± 9.7	63.6 ± 21.4	44.8 ± 15.9	48.2 ± 14.3
15	57.5 ± 14.2	98.2 ± 27.6	67.5 ± 20.4	73.1 ± 17.8
20	77.0 ± 18.5	135.1 ± 34.0	90.5 ± 24.9	98.1 ± 22.2

where C represents the percentage of items selected from N. We are interested in the number of elements that n_1 and n_2 have in common. The elements selected by P_1 ($||n_1||$ of them) can be partitioned into two disjoint subsets: those in n_1 and those not in n_1. The elements not in n_1 are in the set $(N - n_1)$. The elements selected by P_2 will have, on average, C percent of its elements selected from n_1 and C percent of its elements selected from $(N - n_1)$. The number of elements common to P_1 and P_2 is $C \cdot ||n_1||$, and the percentage of common items selected is $(C||n_1||/||n_2||) = (C||n_1||/||n_1||) = C$. Thus, on average, the percentage of common prefetches between our method and a real prefetching algorithm is equal to the coverage.

This suggests that our prefetching model can accurately measure the CPI_{FCA} of a real prefetching algorithm. For example, consider a prefetching algorithm that prefetches 75% of the misses. By specifying a coverage of 75%, our prefetching model prefetches 75% of the same misses as the prefetching algorithm. Thus, even though CPI_{FCA} might be substantially reduced because of a high prefetching coverage, we can expect a CPI_{FCA} comparable to that of the real prefetching algorithm. Similarly, if a prefetching algorithm achieves a relatively low coverage, say 25%, prefetching should have a limited benefit, and only a small change in CPI_{FCA} should result. Thus, our method prefetches 25% of the same misses as the real algorithm, and we can expect a similar CPI_{FCA}.

Figure 3 shows this analysis by plotting the percentage of prefetches that are common and unique to two random processes, P_1 and P_2, for a given coverage. We use P_1 and P_2 to represent two different prefetching algorithms. Coverage is varied between 0 and 100% (the diagonal line). Prefetches common to both random processes (P_1 and P_2) are shown in red, and prefetches unique to P_1 and not prefetched by P_2 are shown in green. The yellow section represents the rest of the misses not prefetched by P_1. The area between the dashed curve and the diagonal curve (coverage) represents the misses prefetched by P_2 and not prefetched by P_1. Note that these are P_1 misses. As long as the prefetches unique to both P_1 and P_2 have

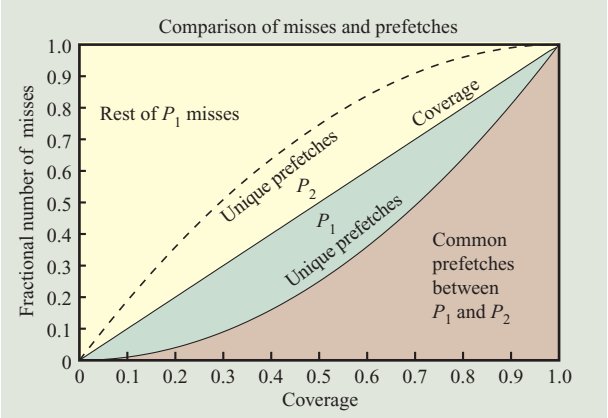

Figure 3

Number of common and unique prefetches between two independent prefetching algorithms, P_1 and P_2, as a function of coverage.

similar prefetching benefits [similar (*cycles/prefetch*) characteristics], there should be little difference in the performance of the two prefetching algorithms. The *Central Limit Theorem* allows us to assume that the unique prefetches issued by P_1 and P_2 will have similar average (*cycles/prefetch*) values. This suggests that the prefetching algorithms P_1 and P_2 should have similar prefetching benefits for the same coverage, accuracy, and timeliness.

To test this hypothesis, two experiments are performed. First, we study the variability of the CPI_{FCA} produced by our prefetching model. For each application we study, nine sets of simulation runs are performed. Prefetching coverage is set to 25, 50, and 75%, and prefetching distance is varied among one, five, and ten branches. Accuracy is fixed at 100% for each set. Each simulation run is repeated ten times, with different initial random seeds used to generate different groups of prefetches relative to the specified coverage and prefetch distance. For each simulation run, we calculate the maximum difference (Δ) in CPI_{FCA}; that is, $\Delta = \max(CPI_{FCA})$

Table 3 Coverage and accuracy measurements for next-sequential and Markov prefetching instructions and data.

Application	*Next-sequential*				*Markov*			
	Instructions		*Data*		*Instructions*		*Data*	
	cov	*acc*	*cov*	*acc*	*cov*	*acc*	*cov*	*acc*
oltp	40	98	25	85	64	92	50	84
perl1	50	90	19	74	61	85	56	81
compress	—	—	9	18	—	—	18	23
mcf	—	—	15	46	—	—	33	58

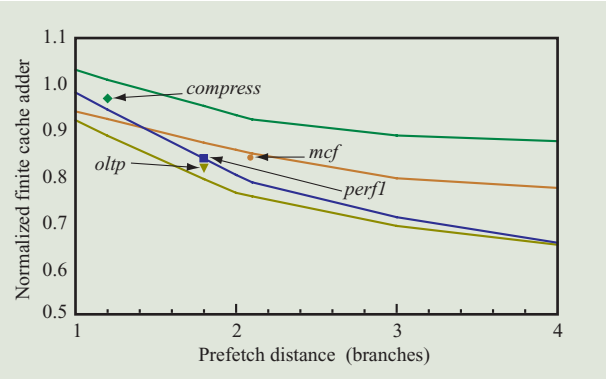

Figure 4

Direct comparison of the normalized finite cache adder for the Markov prefetching algorithm and probabilistic prefetching model.

$-\min(CPI_{\mathrm{FCA}})$ for the ten prefetching runs with different random seeds.

We then calculate the maximum variation (ε) as the ratio between the maximum difference and the CPI_{FCA} without prefetching; that is, $\varepsilon = (\Delta/CPI_{\mathrm{FCA}})$. This represents the percentage of difference relative to CPI_{FCA} without prefetching. These percentages (max. variation) are listed in **Table 4.**

As can be seen from the values in the table, the maximum variation is small. For all but two sets of simulation runs, the maximum variation is less than 1%. Only *perf1* has values greater than 1% (2.8% and 2.6%). For these two cases, the maximum difference has a CPI_{FCA} greater than the other nine simulation runs in the group, and the next greatest variation is less than 1%.

The second experiment measures the ability of our prefetching model to evaluate the performance of the Markov prefetching algorithm for each application listed in Table 3. We model the same cache and processor organization described above. While simulating the Markov algorithm, the average prefetch distance is determined for each application. For example, the *oltp* application has an average prefetch distance of nine instructions, or about 1.8 branches, while modeling the Markov algorithm. Next, the coverage and accuracy values from Table 3, along with a prefetch distance, are supplied as inputs to our prefetching model to produce estimates of the CPI_{FCA} for each application.

Figure 4 shows the results of these simulations. We plot the normalized CPI_{FCA} as a function of the prefetch distance (values 1 through 4 are modeled). That is, each CPI_{FCA} is normalized relative to the CPI_{FCA} of the application without prefetching. Our estimates for the Markov algorithm are plotted as four curves (with labels), one for each application, and the four CPI_{FCA} produced by modeling the Markov algorithm are plotted as four separate points (with workloads identified by arrows) at the appropriate prefetch distance. Note that the average prefetch distance for the Markov algorithm was never an integer. As can be seen, the agreement between our estimates and the measured CPI_{FCA} for the Markov algorithm is good. Values for *perf1* and *mcf* basically lie on the curve representing our estimate of Markov. The value for *compress* is slightly below our estimate (curve) but clearly within the upper and lower bounds of the nearest prefetch distance modeled (integer value). The value for *oltp* is slightly above the curve representing our estimate. The average error for all four applications is less than 2%.

We should offer a few comments regarding the last two experiments. First, we find that the variability of the CPI_{FCA} produced by our prefetching model for a fixed coverage was relatively small. Second, we find that our model can approximate the CPI_{FCA} of a real prefetching model when appropriate values for coverage, accuracy, and timeliness are supplied. Clearly, the model does not prefetch exactly the same misses as the Markov algorithm for a specified coverage, and accuracy. However, it attempts to issue a prefetch from the population of all previous misses, much like other prefetching algorithms. Also, it attempts to issue prefetches across the entire program, much like other algorithms. Thus, even if it does not duplicate the exact order of prefetches issued by another prefetching algorithm, and might prefetch the miss before or miss after a prefetch issued by that algorithm, the CPI_{FCA} produced by the model and the real algorithm are similar.

Table 4 Maximum percentage of difference of CPI_{FCA} among the ten simulation runs with varying random seeds.

Coverage	*compress*			*mcf*			*oltp*			*perf1*		
Prefetch distance:	*1*	*5*	*10*	*1*	*5*	*10*	*1*	*5*	*10*	*1*	*5*	*10*
25	0.3	0.3	0.3	0.1	0.5	0.4	0.3	0.3	0.9	2.8	2.6	0.5
50	0.3	0.3	0.3	0.1	0.1	0.3	0.3	0.2	0.3	0.2	0.4	0.3
75	0.3	0.3	0.5	0.2	0.5	0.4	0.3	0.6	0.4	0.2	0.3	0.3

6. Simulation results

We express all performance relative to the original finite cache adder. As above, we first measure CPI_{FCA} with prefetching turned off, and normalize all prefetching results to this number. We independently study timeliness, coverage, accuracy, and bandwidth, as described in the following subsections.

Most of the data obtained was for 64-KB, four-way set-associative caches with a 128-byte line size. In most cases, we use a miss latency of 20 cycles, with the cache-miss interface configured to be able to process six misses (or prefetches) at any time: three for instructions and three for data. In an investigation of timeliness, we also use a miss latency of 100 cycles. For this longer latency, we assume an interface that can handle more than six outstanding misses.

Timeliness and coverage

We first study performance as a function of timeliness for various coverages, eliminating the effects of accuracy and bandwidth by setting the accuracy to 100% and the bus width to 128 bytes. **Figure 5** is a plot of the computed *compress* and *mcf* workloads, showing normalized values of CPI_{FCA} as a function of timeliness for coverages of 50, 67, 80, and 100%. The miss latency modeled is 20 cycles.

These curves show that most (but not all) of the CPI_{FCA} is eliminated if every miss is prefetched (i.e., 100% coverage) five or more branches ahead. There is a very small improvement by prefetching ten branches ahead (although no real algorithm could make this change without losing more coverage) where the performance reaches a limit.

This demonstrates that at some point (in this case, ten branches), prefetching becomes early enough to hide all of the latency that it is possible to hide. Prefetching any earlier than this does not help; in fact, it can have undesirable effects, as discussed later.

For *compress*, the calculated performance improvement is nearly equal to the coverage, e.g., covering half of the misses eliminates half of the delay. For *mcf* the trend is the same, but the amount of delay removed is a little less than the coverage (e.g., 90% of the delay is removed at 100% coverage).

The conclusion is that with perfect accuracy, ample bandwidth, and adequate timeliness, the performance improvement is proportional to coverage. In general, it is not possible to remove all of the delay (i.e., the constant of proportionality is less than unity). In this particular case, "adequate timeliness" means ten branches. From Table 2, a timeliness of ten branches is an average of 64 instructions for *compress* and 32 instructions for *mcf*. In the next section we explore much larger miss latencies (100 cycles) and the impact on timeliness and maximum prefetching performance.

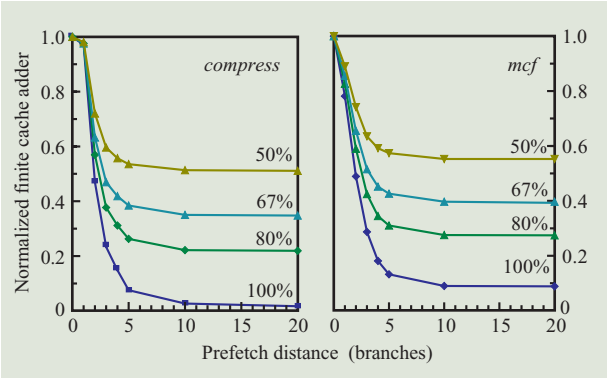

Figure 5

Normalized finite cache adder vs. prefetch distance for several values of prefetch coverage. Data are for both the *compress* and *mcf* workloads, a 64-KB cache, a 128-byte line size, and 128 bytes per cycle transferred to specified locations ("put away").

It is important to note that the flexibility of our model allows us to analyze prefetching by varying coverage, accuracy, and timeliness independently. In a real prefetching algorithm these parameters can actually compete against one another. For example, techniques used to increase prefetching timeliness can reduce accuracy; increasing coverage can also reduce accuracy. Similarly, increasing accuracy can lead to a reduction in coverage and timeliness. In our model we have the ability to vary timeliness from one to 20 branches without negatively affecting accuracy. However, it is known that it is difficult to design a prefetching algorithm that achieves an acceptable coverage and accuracy even with a timeliness of two or three branches, let alone 20.

In our next experiment, we fix the timeliness and vary the coverage (the reverse of the previous experiment). Again, we maintain the accuracy at 100% and the bus width at 128 bytes. **Figure 6** shows data for the *compress* workload. Shown in the figure is the normalized CPI_{FCA} as a function of coverage (between 50% and 100%) for timeliness values of one, two, three, five, and ten branches. The curves in the plot are regression lines.

The figure shows that the performance improvement is linear in coverage for each value of timeliness (given 100% accuracy and ample bandwidth). Again, for a timeliness of ten branches, most of the delay is eliminated at a coverage of 100%; also, the performance improvement approaches a limit at a timeliness of five to ten branches. For a timeliness of one branch, *compress* receives a negligible benefit from prefetching. This is roughly independent of coverage in this range (50% to 100%). For a timeliness of two branches, the amount of delay eliminated for *compress* varies linearly from 25% to 50% as the coverage is varied from 50% to 100%.

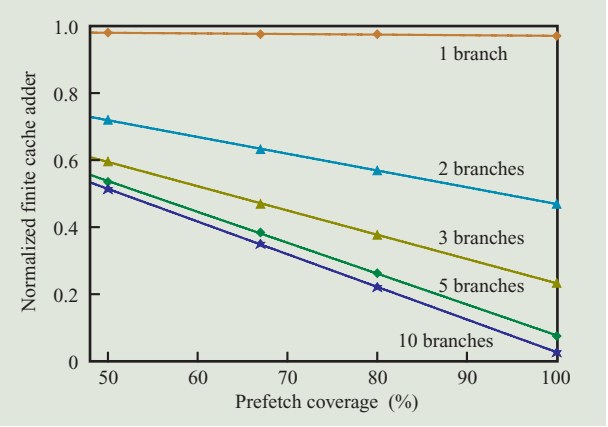

Figure 6

Normalized finite cache adder as the prefetch coverage is varied for several values of the prefetch branch distance. Data for prefetch distances greater than ten branches coincide with the 10-branch curve. Data are for the *compress* workload, a 64-KB cache, a 128-byte line size, and 128 bytes put away.

The linear dependence of performance on coverage is easily understood. For each cache miss that is not prefetched, due to lack of coverage, a demand fetch must subsequently be made. This incurs a penalty of a fixed number of cycles. Since the penalty for each miss is independent, unless queuing effects become important, the resulting decrease in performance is linear in the number of these coverage misses. This is observed for both in-order and out-out-order scheduling.

All four of the applications exhibited a linear relationship between coverage and performance. **Table 5** lists the linear coefficient of this change (slope of the line, expressed as a positive percentage) for all workloads at the prefetch distances simulated. Each value is listed as a percentage of change in performance relative to a change in coverage. The slope becomes constant once the prefetching is sufficiently timely (e.g., ten branches). It becomes as large as 0.99 for *compress* and as small as 0.6 for *perf1*. This means that for each 1% gain in coverage, there is a 0.99% decrease in CPI_{FCA} for *compress*, and a 0.6% decrease in CPI_{FCA} for *perf1* at a prefetch distance of 20 branches.

More on timeliness

To further investigate timeliness by itself, and to study the robustness of our prefetching model, we set both the coverage and accuracy to 100% and increase the miss latency to 100 cycles. (Here all misses have a miss latency of 100 cycles.) This variation allows us to focus on prefetching for the L2 and beyond—and to analyze changes in the prefetching distance needed to compensate for large miss latencies. For example, we are interested in studying the prefetch distance needed to run at infinite cache speeds when each miss has a latency of 100 cycles ($CPI_{\mathrm{FCA}} \rightarrow 0$). To fully investigate the longer latency, we generate new raw-miss files (with 100-cycle miss latencies) and allow the timeliness to be as large as 50 branches (it had been a maximum of 20 branches). We also enable the cache-miss interface to be able to handle 24 outstanding misses.

Table 5 Percentage of change in performance with respect to change in coverage.

Prefetch distance	*mcf*	*compress*	*oltp*	*perf1*
1	0.20	0.02	0.2	0
2	0.50	0.50	0.5	0.03
3	0.70	0.70	0.7	0.38
4	0.80	0.80	0.8	0.52
5	0.90	0.90	0.8	0.60
10	0.93	0.98	0.9	0.60
20	0.93	0.99	0.9	0.60

Figure 7 shows normalized values of CPI_{FCA} as a function of timeliness (to 50 branches) for the *oltp* and *perf1* workloads where the maximum number of outstanding misses allowed is either 6 or 24. In the figure, CPI_{FCA} using a 100-cycle latency is normalized to CPI_{FCA} with a 20-cycle latency and no prefetching. Thus, the normalized CPI_{FCA} becomes greater than 5 (100/20), and is very dependent on the number of outstanding misses allowed.

When the number of outstanding misses is limited to 6, the normalized *CPI* is 6 without prefetching, and asymptotically approaches 2 as timeliness is increased. A timeliness of five to ten branches is sufficient for nearing this asymptote. (Thus, allowing a timeliness of 50 branches is unnecessary, although helpful in illustrating the asymptote.) Recall that for a latency of 20 cycles, with perfect coverage, accuracy, and sufficient timeliness (five to ten branches in that case), CPI_{FCA} vanishes (i.e., all delay is eliminated). When the latency is 100 cycles, the asymptote at 2 shows that all delay cannot be eliminated. When prefetching runs five to ten branches ahead, it generates an average of six prefetches. If it tries to run any farther ahead, it cannot issue the prefetches, because all of the six prefetch buffers are busy. Thus, the limited number of buffers (six in this case) artificially constrains the prefetching to behave as if it is only five to ten branches ahead, even when it is much farther ahead (50 branches in the experiment). All prefetches beyond six are buffered instead of issued. They are not issued until a buffer becomes free.

When the number of outstanding misses is allowed to be 24, CPI_{FCA} keeps dropping until it reaches its asymptote at a timeliness of 15 to 20 branches. For *perf1*, the asymptote is at zero. For *oltp,* the asymptote is not zero, but the remaining delay cannot be imputed to either timeliness or buffer limits. Referring again to Figure 5: For *mcf*, 12 outstanding misses are needed for the performance asymptote to approach zero.

These results raise interesting design problems. With a 100-cycle miss latency and 24 active misses, a new miss is discovered nearly every four cycles. As processor frequencies increase, and thus miss latency (measured in processor cycles) increases, the cache-miss interface must allow for more misses in-flight to hide the added latency. Ideally, the maximum number of in-flight misses a bus can sustain (without multiplexing misses on the bus) is the miss latency divided by the line transfer interval (*miss latency* ÷ *LTI*). As the number of in-flight misses increases, so must the complexity of the cache coherency logic. Symmetric multiprocessor (SMP) clusters of hundreds of processors may require thousands of misses in flight at once.

In addition, if coverage or accuracy is less than perfect, the in-flight prefetch traffic will utilize buffers and bus cycles that cannot be used by exigent misses (that were missed by the coverage of the algorithm). This effect is compounded by bad prefetches and is exacerbated if data and addresses share the same physical bus [which is sometimes done to limit the number of input/output (I/O) pins needed].

All of these issues raise some challenging design problems. Memory designs should be more pipelined, and bus frequencies should be scaled in order to cope with the added bandwidth. Since the point-to-point wire latencies do not scale, signaling protocols and the logic that surrounds the associated buffers will become more complex. Addressing these issues drives additional demands for power in the I/O components and subsystems.

Timeliness and misses

It is important to note that non-prefetch misses still occur during a simulation run, even when running with 100% coverage and 100% accuracy. Note that the miss file contains all of the misses generated by a previous simulation run, and when coverage and accuracy are set at 100%, all of the misses are prefetched in the exact order in which they occurred—at a specified point in time (the prefetching distance). We measure the number of extra misses generated during one of these runs. For each experiment, we put each cache miss into one of three categories: a) *prefetch miss*—requests generated by the prefetching algorithm (from the miss file); b) *necessary demand-misses*—requests that caused a miss (i.e., not covered by prefetching) to data (or instructions) that are essential to the application; or c) *unnecessary demand-misses*—requests that caused a miss for speculative data (or instructions) that are not essential to the application.

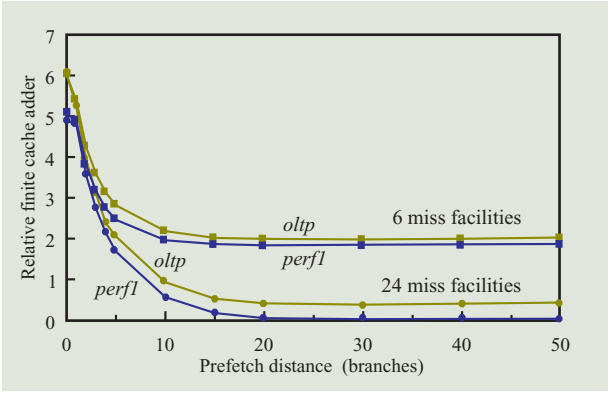

Figure 7

Finite cache adder for a 100-cycle memory access relative to a 20-cycle memory access as the prefetch distance is varied for two workloads (*oltp* and *perf1*), and for 6 and 24 miss facilities.

When running with 100% coverage, *necessary and unnecessary* demand-misses can still occur for two different reasons. First, a prefetch can cause a replacement of data that is still in use or about to be used. The earlier a prefetch is issued ahead of its use, the higher the probability that an additional miss (to prefetch the discarded line) can occur. If the processor requires the discarded line in order to decode or execute an instruction, the miss is *necessary*. Second, *unnecessary* demand-misses occur if the processor is able to speculate more because it is running faster because of the prefetching. In this case, a prefetch has removed a pipeline stall which occurred on a cache miss, and the processor is able to speculate farther down a program path and cause extra fetches and cache misses that were seen in a non-prefetching simulation run. Hence, the number of misses with prefetching turned on can be larger than the number of raw misses. We directly measure the increase in miss rate caused by prefetching by measuring the *unnecessary and necessary demand-misses*. Depending on the application, these can account for as much as 12% more misses.

Table 6 shows the percentage of necessary and unnecessary demand-misses for different prefetch timeliness. The data tabulated is for the workloads *oltp* and *perf1*; data for the workloads *mcf* and *compress* is omitted, since their small instruction working sets do not require many demand misses (increases were less than 0.1%). For *oltp* and *perf1*, the instruction working sets are large, and demand misses are very apparent. The

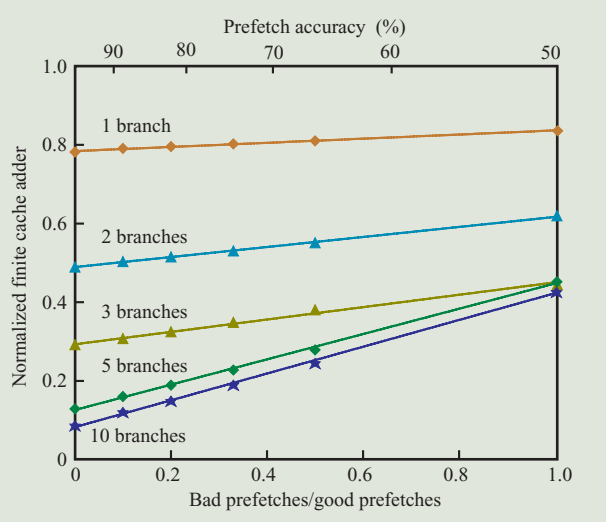

Figure 8

Normalized finite cache adder as the prefetch accuracy is varied for several values of the prefetch branch distance. Data for prefetch distances greater than ten branches coincide with that for the 10-branch curve. Data are for the *mcf* workload, a 64-KB cache, a 128-byte line size, and 128 bytes put away.

percentage of demand misses grows with increased timeliness.

The data shows that the likelihood of a prefetch replacing "live" lines (those that will soon be used) is small (these are the necessary demand-misses—less than 1% for *perf1* in all cases, and less than 4.5% for *oltp* at a prefetch distance of 50 branches). However, the likelihood increases as prefetching is made more timely—because of deeper speculation, resulting in additional misses. If misses are removed (via prefetching), the processor stalls less often and will fetch and decode more instructions, some of them unnecessary.

These effects are shown in Figure 7. The minimum CPI_{FCA} value is reached at a prefetch distance between 20 and 30 branches. As the prefetching distance is increased to 50 branches, more necessary and unnecessary demand misses are generated and a slight increase in the CPI_{FCA} occurs. This trend continues if the prefetching distance is increased.

Accuracy

In this experiment, we vary the accuracy from 50% to 100% for several fixed values of timeliness (exactly like the previous experiment for coverage). In order to study only accuracy, we set the coverage to 100% and the line size to 128 bytes. **Figure 8** shows the results of this experiment for the *mcf* workload. Performance is linear in the number of bad prefetches. Since in these runs

Table 6 Percentage of necessary and unnecessary misses as prefetch distance is increased.

Prefetch distance	*oltp*		*perf1*	
	Necessary demand-misses	*Unnecessary demand-misses*	*Necessary demand-misses*	*Unnecessary demand-misses*
1	1.6	4.3	0.27	1.6
2	1.7	4.4	0.27	1.9
3	1.9	4.6	0.30	2.3
4	2.3	4.9	0.32	2.4
5	2.6	5.2	0.35	2.6
10	2.8	5.4	0.46	3.5
20	3.6	6.5	0.52	5.8
50	4.3	7.7	0.82	6.5

the number of good prefetches is constant, the x-axis is normalized to the number of good prefetches. The number of useless prefetches then becomes proportional to the ratio of bad to good prefetches, which is a function of accuracy, viz.,

$$bad\ prefetches/good\ prefetches = (1 - accuracy)/accuracy.$$

An increase in the number of bad prefetches corresponds to a decrease in accuracy. The far right value ($bad/good = 1$) corresponds to an accuracy of 50%, and the far left value ($bad/good = 0$) corresponds to an accuracy of 100%. An additional axis is plotted above the curves, showing the accuracy in terms of percentage of all of the prefetches.

When the accuracy and coverage are both 100%, roughly 90% of the delay is eliminated for ample timeliness (ten branches). As the accuracy decreases (meaning that more useless prefetches are done), CPI_{FCA} increases proportionately. If the timeliness is insufficient, the performance loss can be severe. For *mcf*, with accuracy at 50% and a timeliness of one branch, the relative CPI_{FCA} is 80% of the original CPI_{FCA}. At a timeliness of two branches, CPI_{FCA} is improved by nearly 50%. Again, a limit is approached as the timeliness increases to ten branches.

The dependence of the performance on the number of bad prefetches is linear, as in the case of its dependence on coverage. Bad prefetches may result in no performance penalty if they neither delay demand misses nor push useful cache lines out of the cache. However, if a useful cache line is replaced, a new demand fetch must be made, and a penalty (linear in scale) will occur, just as in the coverage case. The only difference is that since only a fraction of the bad prefetches result in demand misses, the

Table 7 Percentage of change in performance with respect to change in accuracy.

Prefetch distance	*mcf*	*compress*	*oltp*	*perf1*
1	0.05	0.17	0.39	0.31
2	0.13	0.20	0.52	0.44
3	0.15	0.22	0.61	0.53
4	0.29	0.25	0.67	0.58
5	0.33	0.28	0.69	0.61
10	0.35	0.29	0.72	0.64
20	0.35	0.33	0.73	0.64

coefficient of the performance degradation with bad prefetches will be smaller than the coefficient in the coverage case.

All four applications exhibit a linear relationship for performance vs. number of bad prefetches. **Table 7** lists the relevant linear coefficients (slope, expressed as a percentage) for relating performance to the percentage of unused prefetches for each workload at the values of timeliness studied.

It is important to note that designers are often faced with the need to increase coverage at the expense of accuracy. For example, should prefetching coverage be increased by aggressively issuing more speculative prefetches which may have an accuracy rate of 50%? That is, does an application benefit if two new prefetches are issued, one good and one bad, i.e., unused? Using Tables 5 and 7, we can compare changes in performance when coverage and/or accuracy are varied (for a defined region of interest). Notice that this is done independently of a prefetching mechanism and is calculated for an application. For the applications for which the change in coverage is greater than the change in accuracy (at a given prefetch distance), performance benefits are possible. A performance loss will result if the accuracy slope is greater than the coverage slope.

Additionally, designers are faced with applications that do not respond well to prefetching even when the values of coverage and accuracy are acceptable. For example, consider the situation in which coverage is 50%, accuracy is 75%, timeliness is unknown, and simulation models show that the application loses performance only when prefetching is modeled. Our prefetching model allows a designer to analyze the performance sensitivity of the application to coverage or timeliness changes. By profiling the application (collecting misses and prefetches in the miss file), the designer can systematically increase accuracy and/or timeliness and determine performance sensitivity with respect to each parameter. Again, this

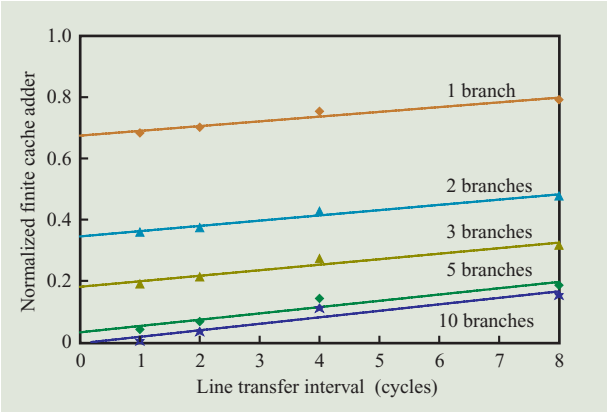

Figure 9

Normalized finite cache adder as the bus transfer rate is varied for several values of the prefetch branch distance. Data for prefetch distances greater than ten branches coincide with that for the ten-branch curve. Data are for the *perf1* workload, a 64-KB cache, a 128-byte line size, and 128 bytes put away.

allows designers to focus their efforts, in both time and dollars, to develop the most cost-effective means to improve application performance.

Bandwidth

In our next set of experiments, we study how bandwidth affects prefetching performance. We use LTI, as defined in the section on terminology, to analyze bandwidth. If the cycles per instruction were constant, bus utilization would be directly proportional to LTI. Although it is time-independent (which is why we chose it), we use it here as a tractable measure of "bandwidth."

First, we set the coverage and accuracy to 100%, the bus frequency ratio to unity, and the line size to 128 bytes. We vary the bus width between 128 and 16 bytes to give us LTI values of one, two, four, and eight cycles. **Figure 9** shows normalized values of CPI_{FCA} as a function of LTI for the *perf1* workload for timeliness values of one, two, three, five, and ten branches. Each curve in the figure is a regression line and is approximately linear. At one cycle, CPI_{FCA} vanishes at a timeliness of ten branches. From one to eight cycles (achieved by reducing the bus width, hence the bandwidth), CPI_{FCA} increases from 0 to 0.15, approximately linearly. The rate of increase (slope) increases slightly as timeliness increases.

Table 8 shows the linear coefficient (slope, expressed as a percentage) of each regression line for each of the workloads at a number of different timeliness values. The linear coefficient represents a percentage performance difference per cycle. In all cases, as the timeliness increases, the coefficient increases. This shows that as

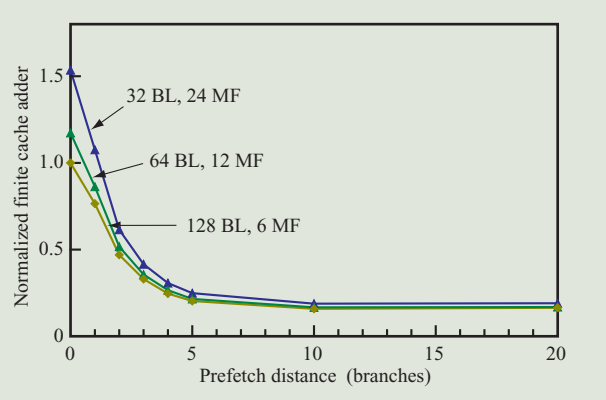

Figure 10

Normalized finite cache adder for three line sizes and number of miss facilities relative to a 128-byte line (BL) with six miss facilities (MF) as the prefetch distance is varied. Data shown are for the *oltp* workload and 32 bytes put away.

the timeliness increases, the performance becomes more sensitive to bandwidth.

In our next to last experiment, we fix the bus width at 32 bytes and vary the cache line size. Not only does this change the LTI, but it also changes the miss rate (recall that a different miss file is required for each line size). For a fixed cache size (64 KB in this experiment), cutting the line size results in doubling the number of lines if both halves of the lines are used. If the useful content remains the same, the system with the smaller line size will incur two misses for every miss incurred by the other system. The LTI for those misses will be cut in half, but the total traffic will be the same.

As before, we set the coverage and accuracy to 100%, the bus width to 32 bytes, and the bus frequency ratio to unity. We use line sizes of 128, 64, and 32 bytes, which results in LTI being equal to four, two, and one cycles, respectively. Since the number of misses that must be handled is larger for smaller line sizes, we change the capability of the cache-miss interface to allow for 6, 12, and 24 simultaneous misses in flight, respectively.

Figure 10 shows normalized values of CPI_{FCA} as a function of timeliness for the three line sizes of the *oltp* workload. The CPI_{FCA} is normalized to its value for the 128-byte line without prefetching. Without prefetching (timeliness = 0), CPI_{FCA} for a 64-byte line is 20% larger than that for a 128-byte line, and CPI_{FCA} for a 32-byte line is 50% larger. As timeliness increases, the three curves converge, at slightly different values of timeliness, indicating that regardless of line size, timeliness is a crucial factor for improving the effectiveness of prefetching.

Table 8 Percentage of change in performance with respect to change in bandwidth.

Prefetch distance	*mcf*	*compress*	*oltp*	*perf1*
1	1.2	0.2	2.0	1.6
2	1.3	0.7	2.0	1.7
3	1.4	0.8	2.3	1.7
4	1.4	1.1	2.4	1.8
5	1.4	1.1	2.6	2.0
10	1.5	1.9	2.7	2.1
20	1.5	2.3	2.7	2.2

7. Bandwidth and realistic prefetching limits

In most of the experiments thus far, we have used coverages and/or accuracies of 100%, a bus frequency ratio of unity, a fairly modest miss latency (20 cycles), and arbitrarily good timeliness to permit individual analysis of the components that affect prefetching. In real systems, coverage, timeliness, accuracy, and LTI are much less optimal; in fact, they interact antagonistically. Furthermore, as processor frequency continues to scale, and as multiprocessor systems become larger, these characteristics of prefetching may continue to degrade.

A 20-cycle miss latency at 1 GHz becomes a 200-cycle miss latency at 10 GHz if the miss access time, its physical size, and its distance to the processor do not scale. If more processors interact with the memory subsystem, the expected latency can be much worse than this, once prioritization logic, queuing, and coherency protocols are included. Furthermore, bus frequency cannot scale at this rate, since it is limited by different physical properties than logic speed. We should expect to see bus frequency ratios grow.

As logic density has increased and processor frequencies have increased, local caches have become progressively larger—both because they *could* be made larger, and because larger caches were required to hold the same temporal content (when time is fixed and the processor runs faster), i.e., to contain the temporal rate of misses. Since the speed of the directory is crucial, its size must not grow (much) to allow for a fixed access time (in cycles) as cycle time decreases. By increasing the line size, the cache size can grow while the directory remains the same size. This allows for a constant directory access even as the frequency of the processor increases.

While an increase in bus width is possible, any increase is limited to some extent by considerations regarding packaging costs, the number of I/O pins, and power. Therefore, as both the line size and the bus frequency ratio increase, the increase in line transfer interval, or LTI (hence bus utilization) compounds.

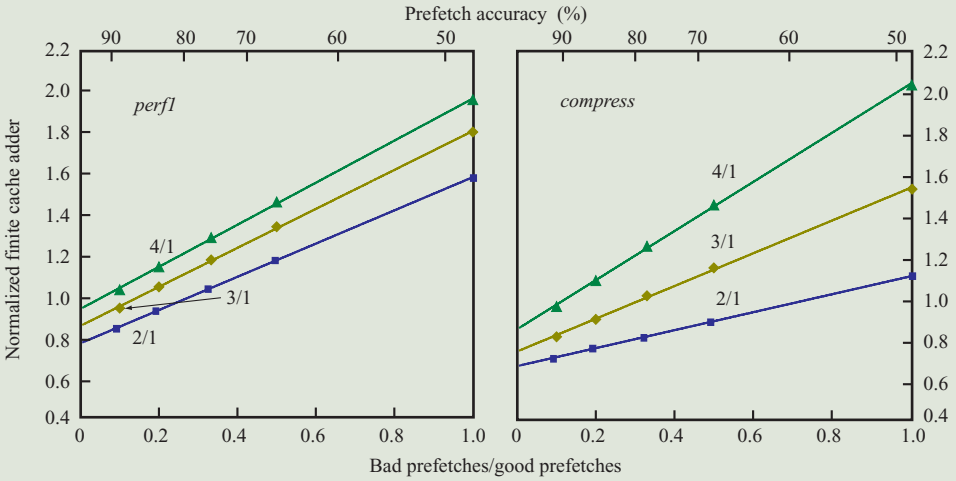

Figure 11

Normalized finite cache adder as the prefetch accuracy is varied for several values of bus frequency ratio. Data are for the *perf1* and *compress* workloads, a 64-KB cache, a 128-byte line size, 80% coverage, 16 bytes put away, and a prefetch branch distance of 10.

To evaluate these effects, we perform a final experiment in which we further increase the LTI. We use a 64-KB cache, a 20-cycle miss latency, a 128-byte line, a 16-byte bus, and bus frequency ratios of 2 to 1, 3 to 1, and 4 to 1. Thus, the LTI is 16, 24, or 32 cycles. To make the prefetching more realistic, we set the coverage to 80% (which is still very respectable for any real prefetching algorithm).

Figure 11 shows the normalized CPI_{FCA} as a function of the number of bad prefetches for the *perf1* and *compress* workloads at a fixed prefetch distance of ten branches. All points above 1.0 indicate that a design incorporating prefetching is inferior to one without prefetching. The figure shows that for even larger values of LTI at 80% coverage, it is very difficult to remove the cache penalty. At an accuracy of 50%, prefetching actually increases CPI_{FCA} by 50 to 90% (over no prefetching) for *perf1* and by 5% to 100% for *compress* at more than adequate timeliness (ten branches). As the prefetching accuracy increases to 100%, prefetching reduces the CPI_{FCA} by only 20% when the processor/bus ratio is 2/1; no reduction is observed if the processor/bus ratio is 4/1 for either application. In fact, a reduction of the CPI_{FCA} by only 10% is observed when the processor/bus ratio is 3/1 at 100% accuracy.

In *perf1*, prefetching causes a performance loss if accuracy is less than 67% with a 2/1 bus ratio; even with a very impressive 90% accuracy (which is nearly impossible to achieve at 80% coverage), performance gain is less than 5% at a 3/1 bus ratio. Since a coverage of 80% should remove 80% of the penalty under these conditions, the difference is primarily due to bandwidth limitations. Similar results are obtained for *compress*.

Therefore, limiting bandwidth should limit (or even reverse) any theoretical performance gains achieved with prefetching. When the LTI grows larger, queuing effects should emerge. Although we cannot examine these effects in detail, we can state briefly that there are nonlinear effects that can quickly dominate system performance if bus utilization is driven too hard. The underlying reason is that misses cluster, and the clustering of misses can quickly push bus utilization to undesired levels. These queuing effects can completely obviate any potential gains made by prefetching.

Figure 12 shows the cumulative distribution function for the number of instructions between successive cache misses for the four applications studied. Clearly, misses tend to cluster: Nearly 50% of the misses for *oltp* and *perf1* occur within a distance of five instructions or less of the prior miss, and 80% of the misses occur within a distance of 15 instructions. The same miss rate is observed for *mcf*, even though its misses are mainly data misses. Similar effects are observed for *compress*, though they are not as severe; 50% of the misses occur within 15 instructions of the prior miss. Recall that nearly all of the misses for *compress* were data misses.

At these instruction/miss distances (times), the bus is still processing the previous miss (prefetch) while the next miss (prefetch) has been initiated. Recall that the LTI is between 16 and 32 cycles for the memory subsystems discussed in Figure 11.

A typical queuing model demonstrates the problem. The average bus utilization ρ is the product of the arrival

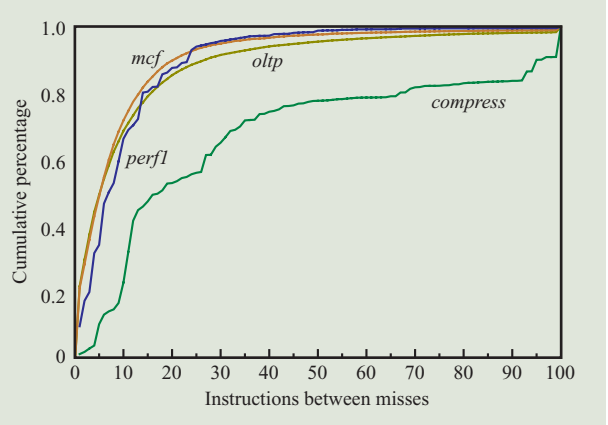

Figure 12

Cumulative percentage of number of instructions between successive cache misses for a 64-KB cache with 128-byte line size. Data are for the *mcf*, *oltp*, *perf1*, and *compress* applications.

rate λ and the average service time S; i.e., $\rho = \lambda S$. The average wait time W is equal to $S/(1 - \rho)$. The service time of a bus is what we have been calling the line transfer interval. Thus, $S = LTI$.

With the use of the bus configurations described in Figure 11 and a 2-to-1 processor/bus ratio, $S = 16$ cycles for a 128-byte line. If we assume that each application runs at $CPI = 1$, nearly 80% of the time a cache miss (or prefetch) will encounter a bus that is busy processing a prior miss. Even at a bus utilization of 50%, the average delay is doubled. That is, an LTI of 16 cycles "looks like" a 32-cycle delay when the bus is 50% utilized.

8. Concluding remarks

We have extracted the raw misses from trace-driven simulations and have associated possible prefetch points in a trace with each miss. (The prefetch points are the set of branches preceding each miss in a workload.) We have parameterized a prefetch with three parameters: timeliness, coverage, and accuracy. These parameters, which are inputs to the simulator that govern how information in the raw-miss file is used (or not) to prefetch, have enabled us, by varying the parameters and rerunning the simulation, to explore the entire space of prefetching algorithms while not simulating any particular algorithm (or even requiring that such an algorithm be possible).

Using this methodology, we have computed the limits of prefetching under perfect conditions, and have shown how degrading those conditions (hence, applying real prefetching algorithms in real systems) can affect the potential for improving performance. In much of the space explored, prefetching actually resulted in a loss of performance.

With perfect coverage and accuracy, sufficient timeliness, ample bandwidth, and sufficient buffering, prefetching can eliminate (almost) all delay caused by cache misses. Interestingly, when portions of this delay are eliminated, a superscalar processor runs farther down speculative paths and generates new misses. This effect is not major, but in principle it prevents prefetching from eliminating *all* misses.

In any real prefetching algorithm, the three parameters (which we varied independently in our study) are mutually antagonistic. Improving timeliness by issuing prefetches earlier introduces more uncertainty in the prefetching, hence reducing accuracy. Improving accuracy by being more careful about what to prefetch reduces coverage. When we model more realistic values of these parameters, the results are sometimes discouraging.

The maximum number of misses (including prefetches) that can be outstanding at any time (i.e., the number of buffers) is crucial to the effectiveness of prefetching. Insufficient buffering prevents the prefetching from being able to get ahead (i.e., from working), and it causes the prefetching to obstruct urgently needed miss data.

We used the line transfer interval to study the effects of bandwidth on prefetching performance. Misses tend to cluster in time, and they must contend with prefetching for bus cycles. Without very ample bandwidth, queuing grows during intervals in which the miss rate is high. This can add significant delays to the miss traffic.

In many processors today, line sizes are modest (e.g., 32 bytes), bus widths are sufficiently matched to the line size (e.g., 8 bytes), and processor frequency is comparable to bus frequency (e.g., 1:1 or 2:1). For such systems, the LTI is small (e.g., four cycles) but still measurable. However, the factors that drive packaging technology and logic speed and density will cause LTI to increase, resulting in serious effects that could render prefetching useless. We have shown this trend even at high levels of coverage and accuracy.

In this study, we have demonstrated that prefetching can be evaluated without regard to any particular algorithm by using a generic model that makes it possible to systematically and independently vary coverage, accuracy, and timeliness.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Standard Performance Evaluation Corporation.

References

1. D. Callahan, K. Kennedy, and A. Porterfield, "Software Prefetching," *Proceedings of the 4th International Conference*

on Architectural Support for Programming Languages and Operating Systems, April 1991, pp. 40–52.
2. T. Chen and J. Baer, "A Performance Study of Software and Hardware Data Prefetching Schemes," *Proceedings of the 21st Annual International Symposium on Computer Architecture* (IEEE), April 1994, pp. 223–232.
3. J. D. Gindele, "Buffer Block Prefetching Method," *IBM Tech. Disclosure Bull.* **20**, 696–697 (July 1977).
4. N. P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," *Proceedings of the 17th Symposium on Computer Architecture,* May 1990, pp. 364–373.
5. A. Smith, "Cache Memories," *ACM Computing Surv.* **14**, 473–530 (September 1982).
6. M. Charney, "Correlation-Based Hardware Prefetching," Ph.D. Dissertation, Cornell University, Ithaca, NY, August 1995.
7. R. Cooksey, S. Jourdan, and D. Grunwald, "A Stateless, Content-Directed Data Prefetching Mechanism," *Proceedings of the ACM Thirteenth Annual International Conference on Architectural Support for Programming Languages and Operating Systems,* October 2002, pp. 279–290.
8. D. Joseph and D. Grunwald, "Prefetching Using Markov Predictors," *Proceedings of the 24th International Symposium on Computer Architecture* (IEEE), May 1997, pp. 252–263.
9. M. Lipasti, W. Schmidt, S. Kunkel, and R. Roediger, "SPAID: Software Prefetching in Pointer and Call Intensive Environments," *Proceedings of the 28th Annual International Symposium on Microarchitecture* (IEEE), November 1995, pp. 231–236.
10. C.-K. Luk and T. Mowry, "Compiler Based Prefetching for Recursive Data Structures," *Proceedings of the ACM Seventh Annual International Conference on Architectural Support for Programming Languages and Operating Systems,* October 1996, pp. 222–233.
11. D. Ortega, E. Ayguade, J.-L. Baer, and M. Valero, "Cost Effective Compiler Directed Memory Prefetching and Bypassing," *Proceedings of the 2002 International Conference on Parallel Architecture and Compilation* (IEEE, ASM, etc.), pp. 189–198.
12. J. Pomerene, T. Puzak, R. Rechtschaffen, and F. Sparacio, "Prefetching System for a Cache Having a Second Directory for Sequentially Accessed Blocks," U.S. Patent 4,807,110, February 1989.
13. A. Roth and G. Sohi, "Effective Jump-Pointer Prefetching for Linked Data Structures," *Proceedings of the 26th International Symposium on Computer Architecture* (IEEE), May 1999, pp. 111–121.
14. Y. Solihin, J. Lee, and J. Torrellas, "Using a User-Level Memory Thread for Correlation Prefetching," *Proceedings of the 29th Annual International Symposium on Computer Architecture* (IEEE), May 2002, pp. 171–182.
15. A. D. Berenbaum and T. E. Jeremiassem, "History-Based Prefetch Cache Including a Time Queue," U.S. Patent 5,778,435, July 1998.
16. J. Collins, S. Sair, B. Calder, and D. Tullsen, "Pointer Cache Assisted Prefetching," *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture,* November 2002, pp. 62–73.
17. W.-C. Hsu and J. E. Smith, "A Performance Study of Instruction Cache Prefetching Methods," *IEEE Trans. Computers* **47**, 497–508 (1998).
18. J. Pierce and T. N. Mudge, "Wrong-Path Prefetching," *Proceedings of the 29th International Symposium on Microarchitecture* (IEEE), December 1996, pp. 165–175.
19. V. Srinivasan, E. S. Davidson, G. S. Tyson, and M. Charney, and T. R. Puzak, "Branch History Guided Instruction Prefetching," *Proceedings of the Seventh Annual International Symposium on High-Performance Computer Architecture* (IEEE), January 2001, pp. 291–300.
20. Y. Zhang, S. Haga, and R. Barua, "Execution History Guided Instruction Prefetching," *Proceedings of the International Conference on Supercomputing,* June 2002, pp. 129–147.
21. S. Van der Wiel and D. Lilja, "A Survey of Data Prefetching Techniques," *Technical Report No. HPPC 96-05,* University of Minnesota, October 1996.
22. P. G. Emma, "Understanding Some Simple Processor Performance Limits," *IBM J. Res. & Dev.* **41**, 215–232 (1997).

Received March 22, 2004; accepted for publication July 9, 2004; Internet publication November 24, 2004

Philip G. Emma *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (pemma@us.ibm.com).* Dr. Emma received B.S., M.S., and Ph.D. degrees in electrical engineering from the University of Illinois, joining the IBM Thomas J. Watson Research Center in 1983. He is currently the manager of the Systems Technology and Microarchitecture Department. Dr. Emma has worked in the areas of microarchitecture, architecture, systems, circuit design, packaging, and interconnection technology. He was a team leader in the IBM G4 and G5 zSeries processor design efforts, responsible for design reliability. Dr. Emma holds roughly 100 patents; he is an IBM Master Inventor, a member of the IBM Academy of Technology, and a Fellow of the Institute of Electrical and Electronics Engineers.

Allan Hartstein *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (hart@watson.ibm.com).* Dr. Hartstein is a Research Staff Member in the Systems Technology and Microarchitecture Department at the Thomas J. Watson Research Center. He received a B.S. degree in physics from the California Institute of Technology in 1969 and a Ph.D. degree in physics from the University of Pennsylvania in 1973. He joined the IBM Research Division in 1974. Dr. Hartstein's research interests have focused on the study of transport properties and electron tunneling phenomena in silicon MOSFETs, artificial neural networks, and, more recently, computer architecture. He has received an IBM Outstanding Innovation Award and two IBM Outstanding Technical Achievement Awards. Dr. Hartstein is a Fellow of the American Physical Society and a Senior Member of the Institute of Electrical and Electronics Engineers.

Thomas R. Puzak *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (trpuzak@us.ibm.com).* Dr. Puzak received a B.S. degree in mathematics and an M.S. degree in computer science from the University of Pittsburgh, and a Ph.D. degree in electrical and computer engineering from the University of Massachusetts. Since joining IBM in 1970, he has spent more than 25 years working in the IBM Research Division. Dr. Puzak has received IBM Outstanding Achievement, Contribution, and Innovation Awards and has served as Chairman of the Computer Architecture Professional Interest Community at the Thomas J. Watson Research Center. He holds several patents in computer architecture on topics pertaining to branch prediction, pipeline structure, and memory hierarchy. His areas of interest include processor design, concentrating on cache and pipeline performance.

Vijayalakshmi Srinivasan *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (viji@us.ibm.com).* Dr. Srinivasan joined the IBM Thomas J. Watson Research Center in 2001 as a Research Staff Member. She received a B.S. degree in physics from the University of Madras in 1990, an M.S. degree in computer science and engineering from the Indian Institute of Science in 1994, and a Ph.D. degree in computer science and engineering from the University of Michigan in 2001. From 2001 to 2003, Dr. Srinivasan was part of a power-aware microsystems project focusing on the development of high-level energy models to evaluate power–performance tradeoffs in high-performance server processors. Her areas of research include computer architecture, microarchitecture, and performance analysis.