Proceedings

International Conference on Computer Design

VLSI in Computers and Processors

October 12-15, 1997

Austin, Texas

Sponsored by

IEEE Computer Society Technical Committee on Design Automation

IEEE Circuits and Systems Society

Los Alamitos, California

Washington • Brussels • Tokyo

Instruction Prefetching Using Branch Prediction Information

I-Cheng K. Chen, Chih-Chieh Lee, and Trevor N. Mudge
EECS Department, University of Michigan
1301 Beal Ave., Ann Arbor, Michigan 48109-2122
{icheng, leecc, tnm}@eecs.umich.edu

Abstract

Instruction prefetching can effectively reduce instruction cache misses, thus improving the performance. In this paper, we propose a prefetching scheme, which employs a branch predictor to run ahead of the execution unit and to prefetch potentially useful instructions. Branch prediction-based (BP-based) prefetching has a separate small fetching unit, allowing it to compute and predict targets autonomously. Our simulations show that a 4-issue machine with BP-based prefetching achieves higher performance than a plain cache 4 times the size. In addition, BP-based prefetching outperforms other hardware instruction fetching schemes, such as next-n line prefetching and wrong-path prefetching, by a factor of 17-44% in stall overhead.

1. Introduction

Instruction prefetching is an important technique for closing the gap between the speed of the microprocessor and its memory system. As current microprocessors become ever faster, this gap continues to increase and becomes a bottleneck, resulting in the loss of overall system performance. To close this gap, instruction prefetching speculatively brings the instructions needed in the future close to the microprocessor and, hence, reduces the transfer delay due to the relatively slow memory system. If instruction prefetching can predict future instructions accurately and bring them in advance, most of the delay due to the memory system can be eliminated.

In this paper we propose an efficient instruction prefetching scheme that makes use of current advanced branch prediction mechanisms that are often already part of the architecture. The branch predictors are built into current microprocessors to reduce the stall time due to instruction fetching and, in general, can achieve prediction accuracy as high as 95% for SPEC benchmarks [SPEC95]. With such high prediction accuracy, the instructions needed in the future can also be predicted accurately and be prefetched in advance. Furthermore, this approach is inexpensive because it applies and shares the existing branch predictors with little additional hardware cost.

Prefetching based on branch prediction (BP-based prefetching) can achieve higher performance than a cache of 4 times the size in all the benchmarks and configurations we examined. BP-based prefetching achieves such high performance by speculatively running ahead of the execution unit at a rate close to one basic block per cycle. With the aid of advanced branch predictors and a small autonomous fetching unit, this type of prefetching can accurately select the most likely path and fetch the instructions on the path in advance. Therefore, most of the prefetches are useful and can fetch instructions before they are needed by the execution unit.

The paper is organized into five sections. In Section 2 we introduce some related prefetching schemes and BP-based prefetching. Section 3 describes our simulation environment and the benchmarks used. In Section 4, we present BP-based prefetching simulation results and provide some qualitative analysis. Finally, we present summary and conclusions in Section 5.

2. Description of Prefetching Schemes

2.1 Related prefetching schemes

The concept of a Look Ahead Program Counter (LA-PC) has been proposed by Chen and Baer [Chen95]. This is a pseudo-program counter that runs several cycles ahead of the regular program counter (PC). The LA-PC is then used to look up a Reference Prediction Table to prefetch data in advance. Although the concept is similar to our proposed scheme, the LA-PC scheme is more conservative; it only advances one instruction per cycle and is restricted to be, at most, a fixed number of cycles ahead of the regular PC. The studies in [Chen95] focused on data prefetching rather than instruction prefetching, and did not evaluate the effects of speculative execution, multiple instruction issue, and the presence of advanced branch prediction mechanisms. Some other data prefetching schemes extending their work can be found in [Liu96, Pinter96].

Another scheme, the next-*n* line prefetching scheme [Smith82], prefetches the next *n* sequential cache lines following the current program counter. This scheme is effec-

1063-6404/97 $10.00 © 1997 IEEE

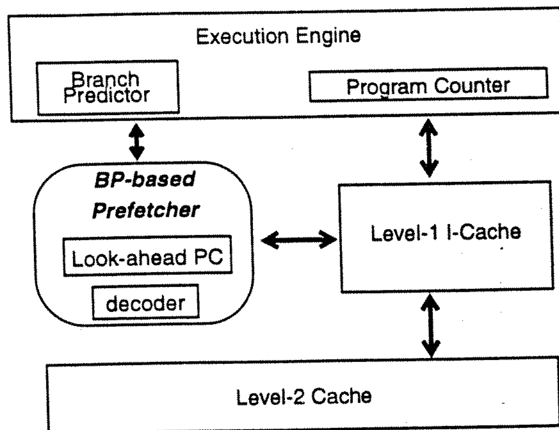

Figure 1: Organization of BP-based prefetching scheme.

tive because it exploits the characteristic that most programs tend to execute sequentially by fetching sequential lines in advance.

A third scheme, wrong-path instruction prefetching has been proposed by Pierce *et al.* [Pierce96]. The wrong-path scheme prefetches along both paths of a branch instead of simply prefetching along the predicted correct path. The wrong-path scheme is based on the observation that programs eventually execute instructions along the not taken or "wrong path." Wrong-path prefetching has been shown to be more accurate and cost-effective than hybrid schemes [Smith92], which are table-based schemes.

2.2 Branch prediction-based prefetching

Conceptually, the instruction prefetching scheme we propose is similar to the look-ahead program counter [Chen95], yet with much more aggressive prefetching policies. The prefetching unit is an autonomous state machine, which speculatively runs down the instruction stream as fast as possible and brings all the instructions encountered along the path. When a branch is encountered, the prefetching unit predicts the likely execution path using the branch predictor, records the prediction in a log, and continues. In the meantime, the execution unit of the microprocessor routinely checks the log as branches are resolved and resets the program counter of the prefetching unit if an error is found.

The hardware needed for prefetching includes a small subset of the main fetching unit: a program counter (PC), a branch history register (without the associated expensive 2-bit counter table), an adder to compute branch targets, and a return address stack. The branch predictors discussed in this paper are two-level branch predictors and return address stacks, which are already found in various commercial microprocessors, such as, Intel Pentium Pro and DEC Alpha 21264. Figure 1 shows a block diagram of the organization of BP-based prefetching scheme.

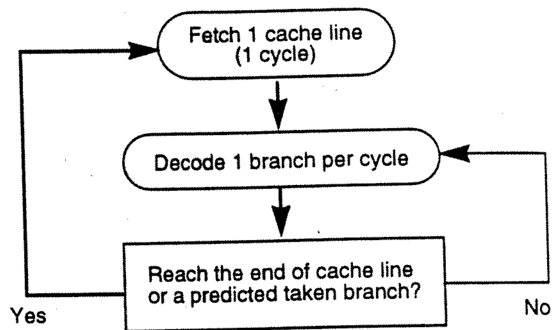

Figure 2: Flowchart of BP-based prefetching.

Figure 2 shows the detailed operation of our proposed prefetching scheme. Initially, the program counter (PC) of the prefetching unit is set to be equal to the PC of the execution unit. Then the prefetching unit spends one cycle to fetch the desired cache line.

The prefetching unit examines an entire cache line as a unit, and quickly finds the first branch (either conditional or unconditional) in that cache line using existing predecoded information or a few bits from the opcode. During the same cycle, the prefetching unit also predicts and computes the potential target for the branch in one of three ways: first, for a subroutine return branch, its target is predicted with a return address stack, which has high prediction accuracy [Kaeli91]. The prefetching unit has its own separate return address stack. Second, for a conditional branch, the direction is predicted with a two-level branch predictor and the target address is computed with the dedicated adder in the same cycle. A dedicated adder is used instead of a branch target buffer, because the first time the branch is encountered it will not yet be recorded in the target buffer. Also note that the two-level branch predictor used in the prefetching unit has its own small branch history register but shares the same expensive pattern history table with the execution unit (a *gshare* scheme is used—see Section 4). The prefetching unit only speculative updates its own branch history register, but does not update the pattern history table. Third, for an unconditional branch, its direction is always taken and its target is calculated using the same adder used for conditional branches. However, for an indirect branch, the prefetching unit stalls and waits for the execution unit because this type of branch can have multiple targets.

The cache line prefetched depends on the predicted direction of a branches. When a branch is predicted to be taken, the cache line containing its target is prefetched; otherwise, the prefetching unit examines the next branch in the cache line. The prefetching unit continues to examine successive branches until the end of the current cache line is reached, then the next sequential cache line is prefetched. The entire process is repeated again for the newly

prefetched cache line.

To verify the predictions made, when a branch is predicted, the predicted outcome is recorded in a log. This log is organized as a first-in-first-out (FIFO) buffer. When the execution unit resolves a branch, the actual outcome is compared with the one predicted by the prefetching unit, which is recorded in the log. If the actual outcome matches the one predicted, the item is removed from the log. However, if the actual outcome differs from the one predicted, meaning the prefetching unit has gone down a wrong path, then the entire log is flushed and the PC of the prefetching unit is reset to the PC of the execution unit. In addition, it is also necessary to reset the contents of the branch history register and the return address stack of the prefetching unit to those of the execution unit.

We must guarantee the prefetching unit always stays ahead of the execution unit to prefetch new instructions. A violation of this condition is detected when the execution unit resolves a branch but the log is empty at that time. If this occurs, we need to reset the PC and branch history information of the prefetching unit to those of the execution unit.

BP-based prefetching can run ahead of the execution unit because the average length of basic blocks is more than 4 instructions. In our experiments described later, BP-based prefetching can advance almost one basic block per cycle (if it is pipelined), while the execution unit can advance at most 4 instructions per cycle (4 instructions retired per cycle). If a wider instruction-issuing machine is considered, BP-based prefetching may employ multiple branch prediction to enhance its speed.

Finally, we also enhanced our proposed prefetching scheme with the next-n line prefetching. This was done by prefetching the next n cache lines following the PC of the prefetching unit rather than from the PC of the execution unit. This enhancement is very effective because it reduces the delay by fetching the sequential cache lines in advance. These next-n line prefetches are given the lowest priority, and are executed when the bus is free (not used by the execution unit or the prefetching unit).

3. Simulation Environment

3.1 Simulation of speculative execution

Due to the speculative nature of prefetching, the normal trace-driven simulation is not enough to capture the behavior of the microprocessor that we are interested in, because it records the actual execution path of a program, and, thus, only contains the instructions executed by the program. However, if there is incorrect speculation, the prefetching activity may bring redundant instructions not used by the program. These redundant instructions are not recorded in the traces, yet they are important in the evaluation of the pollution effect in the memory system. Therefore, to evaluate the effect of incorrect speculation and pollution, we add an instruction-fetching engine. This instruction-fetching engine enables us to fetch any instruction in the program. This engine is implemented in the following way: it first disassembles the binary program to get all the instructions. Then the engine reads all the instructions and keeps them in an internal data structure for future access. When the engine receives an address requesting for an instruction, it searches in the data structure and returns the corresponding instruction. With this addition, the instructions examined are not limited to the ones recorded in the trace-driven simulation; the redundant instructions due to incorrect speculation can also be accessed and included in the final simulation.

To simulate speculative execution, our final simulation combines both the trace-driven simulation and the instruction-fetching engine. The traces from the trace-driven simulation are used to guide the correct execution path of the program, while the instruction-fetching engine is responsible for the speculative behavior of prefetching activities. More specifically, our simulated microprocessor will execute the correct instructions from the traces; our prefetching mechanism will guess the instructions needed in the future and fetch them speculatively using the instruction-fetching engine. In this way, we can correctly model the execution of a program as well as the speculative behavior of prefetching.

3.2 Description of benchmarks

To assess the performance of the BP-based prefetching, we used the SPEC CINT95 benchmarks [SPEC95] to measure its performance against other prefetching schemes. However, some of the benchmarks in SPEC CINT95 have very small instruction footprints, such as *compress* and *ijpeg*, hence, we exclude these benchmarks because they hardly miss at all for the cache sizes examined.

For the SPEC CINT95 benchmark, we used ATOM [Eustace95], a code instrumentation interface from Digital Equipment Corporation, to generate and capture address traces. The benchmarks were first instrumented with ATOM, then executed on a DEC Alpha workstation running OSF/1 3.2 to generate traces. These traces contained instructions from user code and shared libraries. The statistics of traces from the SPEC CINT95 are summarized in Table 1.

3.3 Hardware assumption

For the execution engine, we assume instruction fetching is the only source of stalls. We simulated a 4-issue machine and the instruction fetching stops only at the boundary of a cache line or a branch. Under this model, all instructions are

SPEC CINT 95 benchmarks			
Benchmarks	conditional branches	prediction accuracy	total instructions
gcc	26,521,090	92.18%	191,548,351
go	17,873,434	84.15%	136,898,927
li	25,008,567	93.45%	248,490,436
perl	39,714,631	96.61%	365,938,737
vortex	27,792,013	98.72%	282,462,328

Table 1: Statistics of the benchmarks used

Input to the SPEC95 benchmarks was a reduced input data set; each benchmark was run to completion.

executed within one cycle after they are fetched. This simplified assumption adds more pressure to the instruction prefetching.

For the memory system, we assume 1 cycle access time for a level-1 instruction cache hit, and 6 cycles for a miss. We also assume a perfect level-2 instruction cache, so it will always have the instructions needed.

The branch predictor used in all the following simulation is a variation of two-level dynamic branch predictor, *gshare* [McFarling93], with 15 address bits and 9 global history bits. This predictor has a hardware cost of about 8K bytes of storage.

3.4 Bus arbitration policy

We assume the bus to the level-2 cache can only take one request per cycle, so a bus arbitration policy is needed between the execution engine and the prefetching unit. All requests are serviced in a prioritized order: First, requests from the execution engine are serviced, since these requests result from cache misses and directly affect total execution time. Second, prefetches based on branch prediction are serviced, because they are more accurate than sequential prefetches. Finally, sequential prefetches are serviced when neither of the above is present, otherwise they are postponed and stored in a first-in-first-out queue.

To avoid redundant bus traffic, all requests are compared against requests in transit on the bus, and any duplicated requests are canceled.

4. Simulation Results and analysis

In this section, we compare our scheme with the best of next-*n* line prefetching scheme, and wrong path prefetching. For the next-*n* line scheme, we examined the performance of next 1 to 4 lines, and selected the best configuration, next-2 lines, as representative. We also plotted the best configuration for BP-based prefetching, the basic BP-based scheme (a look ahead program counter) plus the sequential next-2 lines of the look ahead PC, denoted as BP-2 in the legends.

The metric we use to measure the performance is the total execution time in cycles. To measure how much improvement can be achieved, we further compute the percentage of stall time (stall overhead), as follows:

$$\text{stall overhead \%} = \frac{\text{total execution time} - \text{perfect execution time}}{\text{perfect execution time}} \times 100$$

Here the perfect execution time assumes a perfect cache with zero miss rate. Therefore, this perfect execution time is the best possible lower-bound we can ever achieve.

In Figure 3 to Figure 6, we compare the performance of the best configuration of next-*n* line prefetching, wrong path prefetching, and the best configuration of BP-based prefetching.

Figure 3 shows that BP-based prefetching outperforms both next-*n* line prefetching and wrong path prefetching in all benchmarks and cache sizes examined. The *y*-axis indicates the stall overhead, hence a lower bar indicates a better scheme. Averaging across benchmarks, BP-based prefetching is better by a factor of 17-32% in stall overhead than next-2 line prefetching, and by a factor of 34-44% than wrong-path prefetching.

To study the sources of improvement, we measure the number of prefetches generated per 100 instructions. In Figure 4, two cache configurations are shown for each benchmark: small (4K) and large (16K). Each bar indicates the total prefetches generated by each scheme, and these prefetches are further classified into three categories. Helpful prefetches (shown in black) are the prefetches that are actually used by the program and, hence, improve the total execution time. Neutral prefetches (shown in white) are the prefetches that were not used by the program, yet they do not cause any harmful effects either. Harmful prefetches (shown in gray) are the ones that replace useful data (pollute the cache), and, hence, lower the performance. These harmful prefetches cause misses that would not occur in a cache without prefetching.

In Figure 4, we can see that BP-based prefetching has more helpful and more total prefetches than other schemes. These extra helpful prefetches improve the total execution time. Also note that the portion of harmful prefetches in BP-based prefetching is slightly smaller than other schemes, and this fact helps to improve the execution time too.

To analyze the nature of useful prefetches, we further classify the useful prefetches into two categories: prefetches causing hits, and prefetches reducing miss penalties, as shown in Figure 5. The prefetches causing hits are prefetches early enough such that instructions are already in the cache by the time the program needs them (hit in cache). These prefetches completely reduce the penalty to zero. On the other hand, the prefetches reducing miss penalties are also correct prefetches, but they are not generated early

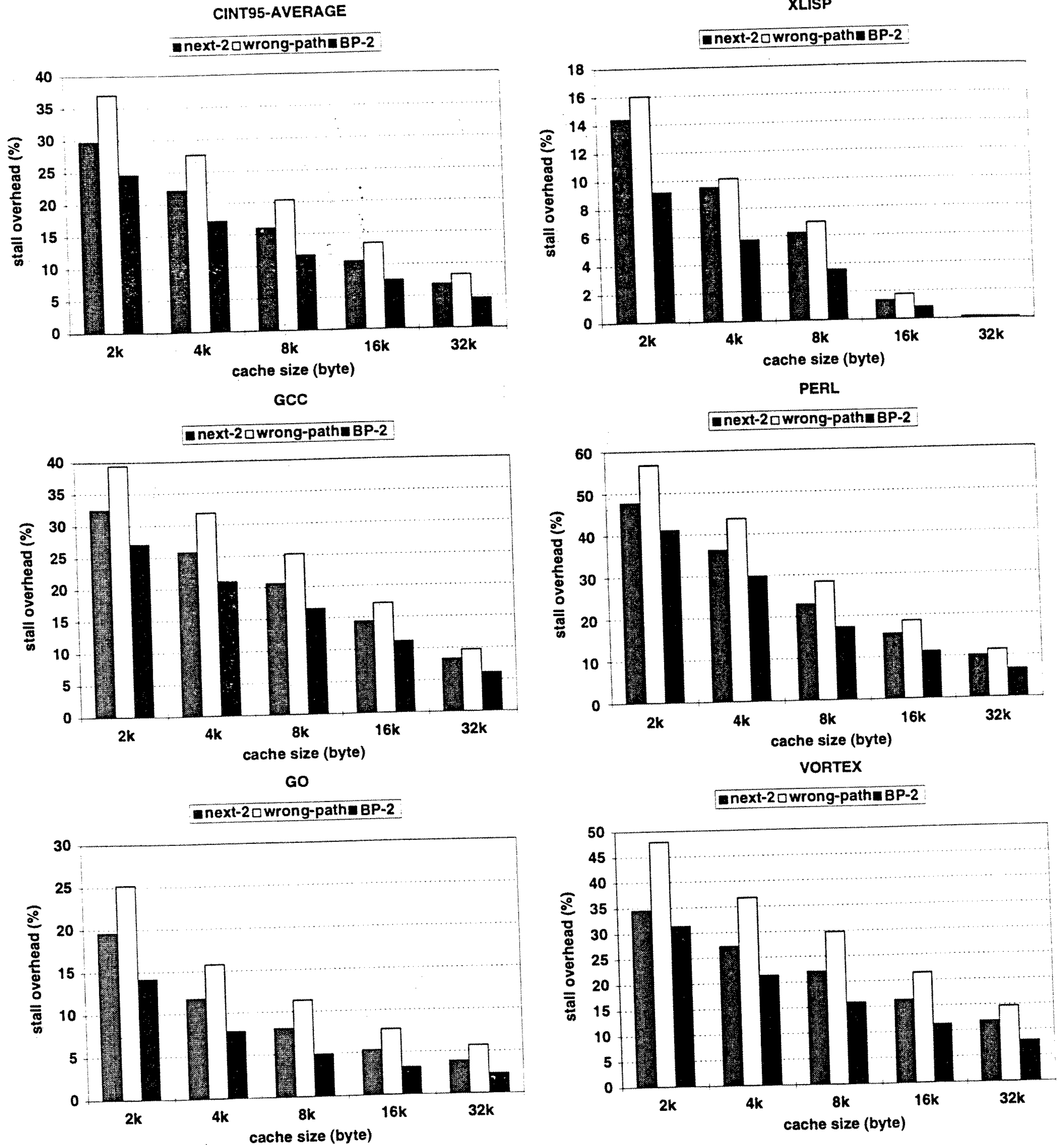

Figure 3: Performance measure: stall overhead for different schemes. Stall overhead measures the extra execution time needed over a perfect cache with zero miss rate.

enough. By the time the program needs the instructions, these prefetches have not brought the instructions into the cache yet (hit in transfer). Therefore, there are still some penalties associated with these prefetches, but the penalties are smaller than normal cache misses.

As shown in Figure 5, BP-based prefetching has more

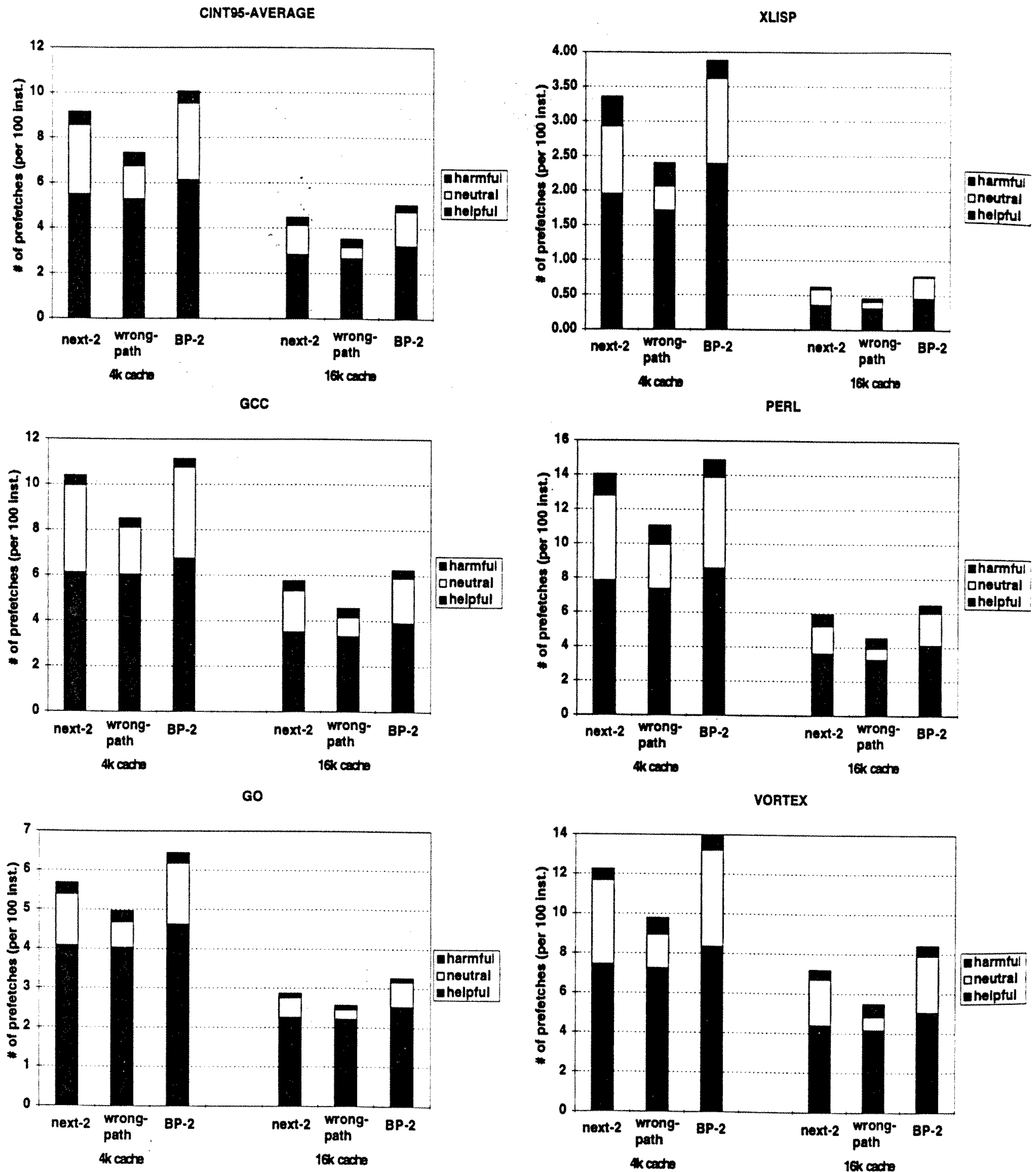

Figure 4: Total prefetches generated in each scheme and the classification of these prefetches.

prefetches causing hits than other schemes (hit in cache, shown in black). This means that BP-based prefetching is able to generate useful prefetches earlier, in addition to generating more useful prefetches, leading to better performance.

To study the impact of prefetching on the overall memory traffic on the bus to level-2 cache, we plot the utilization of the bus in Figure 6. The y-axis represents the percentage

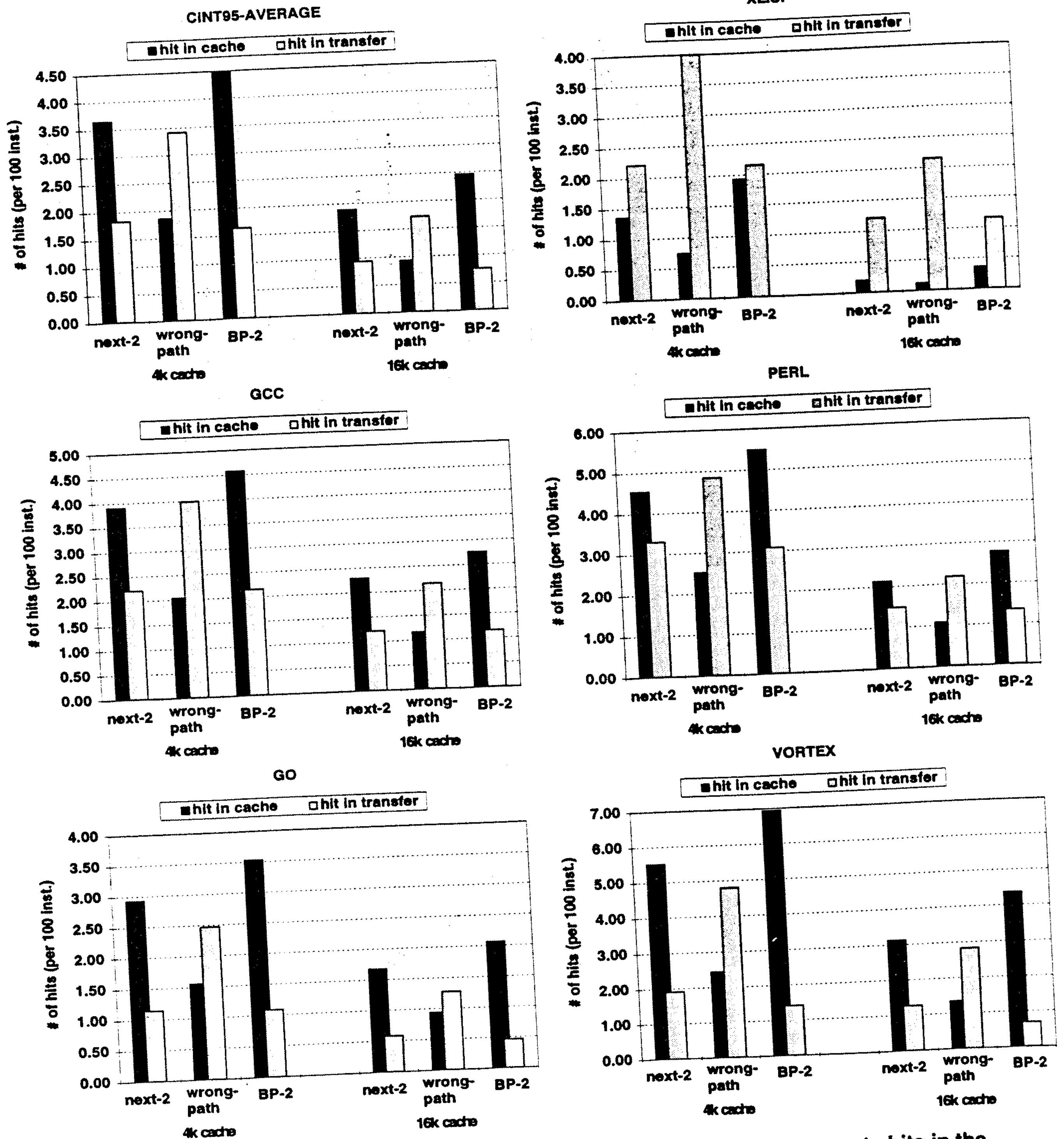

Figure 5: Further classification of useful prefetches. Early prefetches generate hits in the cache, and late prefetches generate hits being transferred.

of bus utilization, which is computed as: (total bus busy cycles)/(total execution cycles)x100. Here the traffic that keeps the bus busy includes the miss requests generated by the execution unit as well as the prefetching requests. Even though BP-based prefetching generates more prefetches, the overall bus utilization is only slightly higher than next-

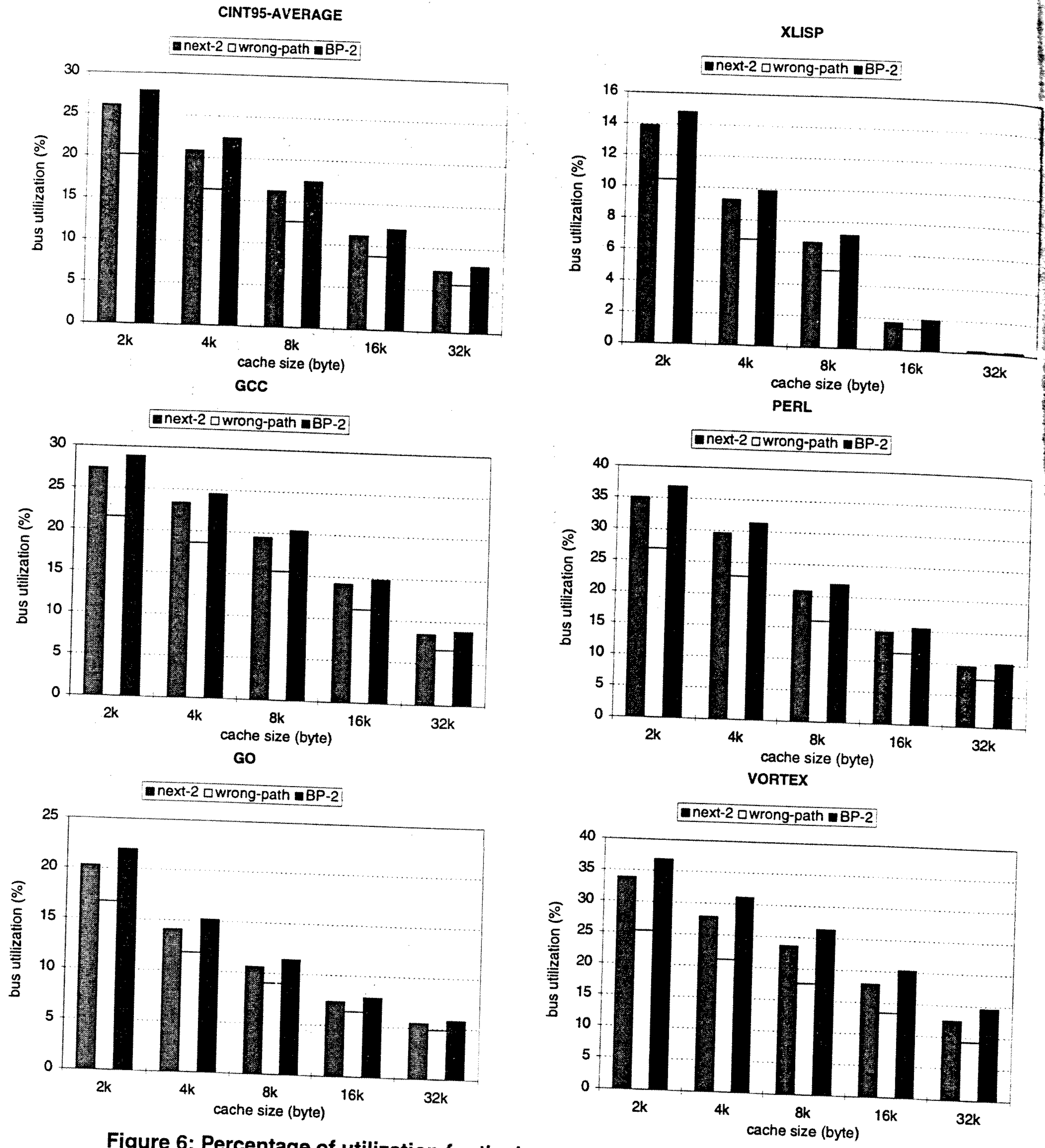

Figure 6: Percentage of utilization for the bus to level-2 cache for different schemes.

2 line prefetching. This is because most of the prefetches are helpful in BP-based prefetching and, thus, reduce the miss requests generated by the execution unit. Therefore, BP-based prefetching is very economical in bus traffic by being selective and accurate. Also note that the total execution time for BP-based prefetching is shorter than other schemes, so the percentage would appear higher even with similar level of traffic. Averaging across benchmarks, the bus utilization of BP-based prefetching is about 28%.

Finally, to get the big picture on the performance of BP-

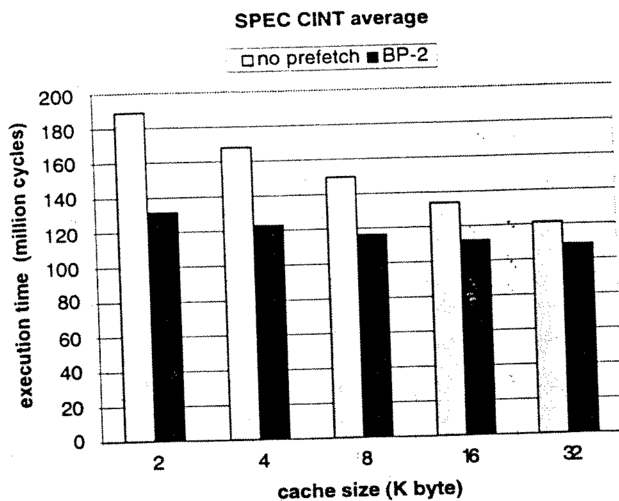

Figure 7: A cache with BP-based prefetching achieves lower execution time than a plain cache of 4 times the size.

based prefetching, the total execution time of BP-based prefetching is compared with a plain cache without prefetching. As shown in Figure 7, a cache with BP-based prefetching achieves lower execution time than a cache of 4 times the size. In particular, a 2K BP-based prefetched cache even outperforms a 16K cache without prefetching.

To gain further understanding, we compare BP-based prefetching scheme with other schemes. Unlike table-based schemes, the BP-based prefetching scheme is able to reduce the first time compulsory misses by pre-computing target addresses. This ability to independently compute target addresses eliminates the need for expensive tables and the awkward situation of having no initial history information at first. Furthermore, using advanced dynamic branch predictors offers much more accurate target-prediction than simple table-based schemes. Therefore, BP-based prefetching, like wrong-path prefetching, is more effective than table-based prefetching schemes.

By reducing stalls on taken branches, BP-based prefetching also outperforms wrong-path prefetching. Our prefetching scheme runs ahead of the real program counter, allowing it to compute and fetch targets in advance, even before they are requested by the execution unit. In contrast, wrong-path prefetching calculates target addresses at decode stage, too late to produce any useful prefetches when branches are actually taken.

From the above analysis, we can see the benefit of BP-based prefetching lies in the ability to run ahead of the real program counter.

5. Summary and Conclusions

In this paper, we present an effective instruction prefetching method, branch prediction-based prefetching, which applies branch prediction information and speculatively runs down the instruction stream. BP-based prefetching can achieve higher performance than a cache of 4 times the size. Examining other hardware instruction prefetching schemes, we see that BP-based prefetching is better by a factor of 17-32% in stall overhead compared to the best next-n line prefetching, and by a factor of 34-44% compared to wrong-path prefetching. BP-based prefetching is able to generate more useful prefetches than other schemes and generate them earlier. In addition, these prefetches are generated selectively, thus, the bus utilization is very close to next-n line prefetching, averaging 28% for SPEC95 benchmarks.

Acknowledgment

This work was supported by DARPA contract DAA H04-94-G-0327.

Reference

[Chen95] Chen, T.-F. and Baer, J.-L. *Effective hardware-based data prefetching for high-performance processors.* IEEE Transactions on Computers, Vol. 44, No. 5, May, 1995.

[Eustace95] Eustace, A. and Srivastava, A. *ATOM: A flexible interface for building high performance program analysis tools.* Proceedings of the Winter 1995 USENIX Technical Conference on UNIX and Advanced Computing Systems, January 1995.

[Kaeli91] Kaeli, D. and Emma, P. G. *Branch history table prediction of moving target branches due to subroutine returns.* Proceedings of the 18th International Symposium on Computer Architecture, May 1991.

[Liu96] Liu, Y. and Kaeli, D. R. *Branch-directed and stride-based data cache prefetching.* Proceedings of the International Conference on Computer Design, October, 1996.

[McFarling93] McFarling, S. *Combining branch predictors.* WRL Technical Note TN-36, June 1993.

[Pierce96] Pierce, J. and Mudge, T. *Wrong-path prefetching.* Proceedings of the 29th Annual International Symposium on Microarchitecture, December 1996.

[Pinter96] Pinter, S. S. and Yoaz, A. *Tango: a hardware-based data prefetching technique for superscalar processors.* Proceedings of the 29th Annual International Symposium on Microarchitecture, December 1996.

[Smith82] Smith, A. J. *Cache Memories.* Computing Surveys, Vol. 14, No. 3, 1982.

[Smith92] Smith, J. E. and Hsu, W.-C. *Prefetching in supercomputer instruction caches.* In Supercomputing'92, November 1992.

[SPEC95] SPEC CPU'95, Technical Manual, August 1995.