

# Non-Sequential Instruction Cache Prefetching for Multiple-Issue Processors

Alexander V. Veidenbaum  
*Dept. of Information and  
Computer Science*  
University of California Irvine  
alexv@ics.uic.edu<sup>1</sup>

Qingbo Zhao  
DSET Inc.

Abduhl Shameer  
Microsoft Inc.

## Abstract

This paper presents a novel instruction cache prefetching mechanism for multiple-issue processors. Such processors at high clock rates often have to use a small instruction cache which can have significant miss rates. Prefetching from secondary cache or even memory can hide the instruction cache miss penalties, but only if initiated sufficiently far ahead of the current program counter. Existing instruction cache prefetching methods are strictly sequential and do not prefetch past conditional branches which may occur almost every clock cycle in wide-issue processors. In this study, multi-level branch prediction is used to overcome this limitation. By keeping branch history and target addresses, two methods are defined to predict a future PC several branches past the current branch. A prefetching architecture using such a mechanism is defined and evaluated with respect to its accuracy, the impact of the instruction prefetching on performance, and its interaction with sequential prefetching. Both PC-based and history-based predictors are used to perform a single-lookup prediction. Targeting an on-chip L2 cache with low latency, prediction for 3 branch levels is evaluated for a 4-issue processor and cache architecture patterned after the DEC Alpha-21164. It is shown that history-based predictor is more accurate, but both predictors are effective. The prefetching unit using them can be effective and succeeds when the sequential prefetcher fails. In addition, non-sequential prefetching is better at hiding latency due to earlier initiation. The two types of prefetching eliminate different types of misses and thus can be effectively combined to achieve better performance.

## 1 Introduction

Instruction-level parallelism is one of the main factors allowing the high performance delivered by state-of-the-art processors. Such a processor is designed to issue and execute  $K$  instructions every cycle.  $K$  is typically 4 in today's processors. A fetch unit's task in such a processor is to supply the decode unit with  $K$  instructions every cycle. This is accomplished by having a wide instruction cache (I-cache) supplying  $\alpha * K$  instruction words every  $\alpha$  cycles, where  $\alpha$  is typically between one and three. This is a difficult task for a typical processor with a clock speed of 200+MHz and more so for a

---

<sup>1</sup> This work was performed while the authors were in the Dept. of Electrical Engineering and Computer Science at University of Illinois at Chicago

high-end processor with a 600MHz clock. It will become even more difficult with clock rates reaching 1GHz and beyond.

Two major problems limiting the instruction issue width  $K$  are branches and instruction cache misses. The former has received a lot of attention. Branch prediction has been used to allow execution to speculatively continue while a conditional branch is resolved. Overall, branch prediction has been very successful although conditional branches remain a major problem in further increasing the instruction issue rate. Branch Target Buffers (BTB) and call/return stacks have been used to tackle the other types of branches, but this as well can be further improved.

The second problem, I-cache misses, is also a difficult one but with fewer solutions proposed for solving it. One, brute-force solution is to increase the primary I-cache size. This may not always be possible or desirable for an on-chip I-cache because the cycle time of the cache is determined by its size [JoWi94] and is a major factor in determining the CPU clock speed. This limits a typical I-cache size to between 8 and 32KB in the current generation. Fast processors, like the DEC Alpha-21164 [ERPR95], are bound to have small I-caches in this range and thus higher miss rates. The Alpha-21164 8KB I-cache has miss rates as high as 7.4% for SPEC92 codes, as reported in [HePa96]. A cache hierarchy is used to reduce the miss penalties. For example, the DEC Alpha-21164 has a unified on-chip second-level cache (96KB) and an off-chip third-level cache (typically 2MB). A large L2 cache has a low (instruction fetch) miss rate, typically well under 1%.

The problem of high primary I-cache miss rates has been traditionally addressed via sequential instruction prefetching. Sequential prefetching typically starts with an I-cache miss address and stops when a branch instruction or an I-cache miss are encountered. The key to successful prefetching is to issue a predicted future instruction address to the L2 cache at least  $T$  cycles before it is needed, where  $T$  is the L2 cache latency. Stated in a different way, instruction fetch prediction needs a lookahead of  $T*K$  instructions.  $T*K=24$  for the Alpha 21164 is representative modern processors and therefore, sequential prefetching cannot get far enough ahead of the fetch unit given average branch probabilities.

This paper presents an I-cache prefetch mechanism with a longer look-ahead. The term "prefetching" is used here to mean fetching from a second-level cache to the I-cache ahead of the current program counter (PC). Given typical branch frequencies in programs, this calls for predicting a prefetch address across several branches. "Multi-Level" Branch Prediction (MLBP) [YeMP93] is utilized to get around this obstacle and predict and prefetch branch target instructions before they are requested. Sequential prefetching is also studied and its effectiveness and relationship with branch target prefetching is explored. Only blocking I-caches are considered, although lockup-free caches can help to combine branch prediction and prefetching.

This paper makes three major contributions. First, it defines two multi-level predictors and a prefetching mechanism which can predict a likely future PC over several intervening branches and initiate the prefetch. Second, several benchmarks are analyzed to understand the importance of different miss types, the predictor accuracy, and the effect on miss rate and CPU time. Third, the complimentary nature of sequential prefetching and the branch target prefetching are shown and a combination of the two is shown to have best results. It is also shown that MLBP-based prefetching gets ahead of the fetch unit while sequential prefetcher initiates but often does not complete many of its requests in time.

## 2 The Approach

Our mechanism to predict a prefetch address with a sufficient time to complete the prefetch divides instruction addresses in two classes. First is a class of sequential addresses defined as  $\text{Mem}[\text{PC} + \delta]$ , where the range of  $\delta$  addresses contains no transfer of control instructions. Sequential prefetching for this class has been widely used utilizing an instruction prefetch buffer. The second class are addresses of branch targets to which a transfer can occur from a given branch. Conditional branches, unconditional branches, and call/returns will be treated in the same way since all require a new prefetching path.

For the second class, a branch has to trigger prefetching along one of the two possible paths as specified by a branch predictor. In this case an address of the branch target needs to be predicted in addition to a taken/not taken prediction. For an even longer lookahead, several future branches need to be predicted and one of the possible target addresses predicted. This has been called multilevel branch prediction [YMP93]. In other words, if a branch  $B_i$  is currently executed we would like to predict branch outcome and target address of branch  $B_{i+K}$  that will follow  $B_i$  in the dynamic execution sequence.  $K$  is the lookahead distance in the number of branches past the current PC, e.g.  $K=3$  predicts a target of a branch 3 (dynamic) branches past the current PC.  $K=0$  is a standard branch prediction.

The approach combines the ideas of branch prediction and a BTB in the following way. The BTB concept is extended to look ahead  $K$  branches and return a prediction for the next prefetch address. We will call this a multilevel BTB (mBTB). A mBTB lookup is performed when a current instruction is a branch using its PC. The mBTB returns  $2^K$  possible branch targets. A 2-bit history counter is stored with each target.

The saturating up/down counter is used to select the most frequently encountered target as the most likely. A counter of a correctly predicted target is incremented while all other counters are decremented. Again, for  $K=0$  our approach reverts to a standard BTB. A Branch History Register (BHR) maintains taken/not taken branch status plus a PC and a target address of the last  $K$  executed branch instructions. The history and target addresses in the BHR are used to update the mBTB. A new taken target, that of  $B_i$ , is added to a mBTB entry pointed to by the PC of  $B_{i-K}$  in the BHR.

I-cache prefetching architecture consists of  $K$  sequential prefetch buffers, each holding one or more cache lines. A prefetch buffer  $P_j$  is used to prefetch branch  $B_i$  target such that  $(i \bmod K) = j$ , so that buffers are used in a circular fashion. The next prefetch buffer in sequence is allocated and prefetching with a  $K$ -branch lookahead is initiated every time a branch is executed. The  $K$  prefetch buffers are looked up in parallel with the I-cache and can return a line directly to the fetch unit.

The hardware complexity of the mBTB is  $(2^K + 1) * \log(\text{Addr\_size})$  and grows rapidly with  $K$ .  $K$  also affects the latency: an associative lookup of  $K$  buffers needs to be performed in parallel with the I-cache lookup and one of the  $K+1$  results selected. To keep the complexity down and the table access time low, only taken branch targets will be predicted and stored in the mBTB. Thus to predict over two branches, for instance, requires only 4 addresses and 4 two-bit counters.

Multilevel branch prediction based prefetching can be combined with sequential prefetching which can pick up some of the fall-through paths, for instance by using multi-line prefetch buffers  $P_j$ . This deals with not-taken paths not in mBTB. Overall, every branch encountered by the CPU starts a new branch target prefetch.



address/2-bit “saturating” counter pairs for each possible branch target and a cache tag accessed with low-order bits of instruction address.

The predictor supports two operations: a lookup and a change/update. The lookup is performed when a new prediction is needed, i.e. when the current instruction is a branch. In this case the PC is used to access the PT entry, the counter with a maximum value among the four counters is identified, and the target address associated with the counter is returned as the prediction. On update, a PT entry is determined by the PC in  $BHR_2$  and the address/counter pair in a slot pointed to by the  $BHR_{2,1}$  History Bits is changed or updated. The pair is changed to the current PC and the counter set to a selected initial value if the target address is new, otherwise the counter is simply incremented while the other three counters in this entry are decremented.

## 4 System Organization

The focus of this paper is the instruction fetch logic of a processor and its I-cache. Ideal instruction issue logic, execution units, and primary D-cache are assumed and their stalls are not modeled. To anchor our system in reality and support the claims of very high clock rates we base the processor pipeline, its behavior, and timing on the DEC Alpha 21164.. The I-cache organization and interface to on-chip, pipelined L2 cache, the branch predictor, and the sequential prefetcher are based on the description in [RPPR95]. Of course, our model is only an approximation. The general system organization studied is shown in Figure 2. Four variants of this architecture are modeled to study instruction prefetching. An architecture may thus omit a particular prefetching unit, such as a stream buffer or MLBP prefetcher. The various system units are described first, followed by the description of the four systems.

### 4.1 Instruction Cache Organization

The instruction cache is a direct-mapped, 8KB cache with 32B blocks and a 2-cycle latency. Thus a cache block contains eight instructions. A block is fetched into two 16Byte staging buffers for access by the fetch unit. Blocks have to be aligned and a 32B fetch cannot cross cache block boundaries.

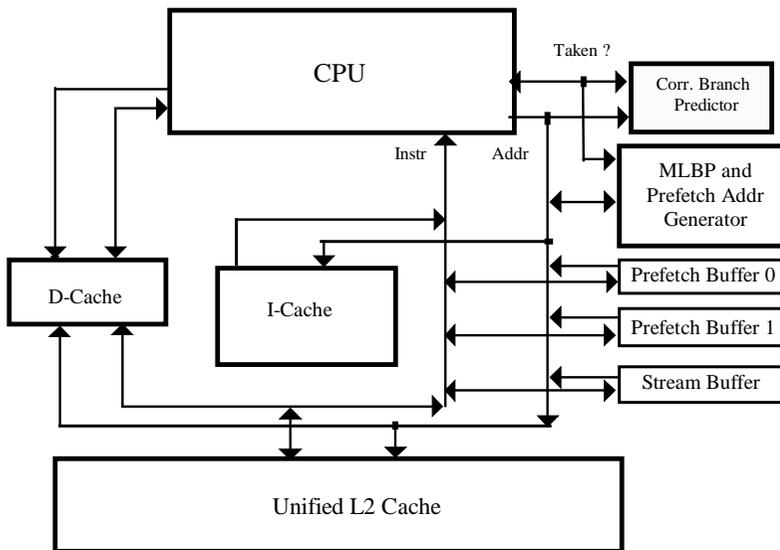
For comparison, we also present some results for 16KB I-caches, 2-way set associativity, and with 64B blocks. The 2-way set associative cache uses a random replacement policy.

### 4.2 Processor Organization

The processor is a quad-issue superscalar processor. Every cycle an aligned 16-Byte set of four instructions is accessed from the I-cache staging buffer. The processor can issue a maximum four instructions per cycle. This corresponds to half a cache block and, in any cycle, the instruction issue will not cross the half block boundary.

The following are the only stages in the instruction pipeline we model:

- S0** - cache access, delivers four instructions plus information for decoding and slotting
- S1** - branch decoding, branch prediction, next PC computation
- S2** - instruction slotting to execution units
- S3** - instruction issue/register access
- S4** - execute stage I



**Figure 2.** Overall System Organization

During **S0** the instruction cache returns a naturally aligned block of four instructions along with necessary information for instruction decoding and slotting. During **S1**, the CPU decodes the four instructions, predicts branch outcome if one of the instructions is a branch, and uses branch prediction to generate the next I-cache half-block address to be used. If one of the instructions is a branch and it is predicted taken the rest of the block is not executed. There is a 1-cycle stall for any taken branch to access the new block. The instructions flow through **S2** and **S3** without any stalls. The conditional branch outcome is known in **S4** resulting in a 5-cycle branch misprediction penalty.

### 4.3 Instruction Fetch Unit

The fetch unit issues a new address to the I-cache every other cycle. It also generates the next half-block address as either  $PC+16$  or  $PC+Branch\_displacement$  if there is a taken branch in the current half-block. The current half-block comes from one of the 2 staging buffers in the I-cache. If a half block contains a branch and the branch is predicted taken, the sequential instruction fetch is stalled for one cycle while the new PC is requested from I-cache. Note that a dedicated branch predictor is used here only for fetching from the I-cache and *not* for I-cache prefetching.

### 4.4 Branch Predictor

A standard, (2,2) correlated branch predictor with a table size is 1K entries is used.

## 4.5 L2 Cache and Memory

The L2 cache with 0% miss rate has its access pipelined and a latency of 6 cycles. A new fetch or prefetch can be issued to it every other cycle. In our system 3 units may simultaneously attempt to issue a request to the L2 cache: the instruction fetch, the sequential prefetch, and the MLBP-based prefetch units. An arbiter selects one of these with the following priorities: instruction fetch, MLBP prefetcher, sequential prefetcher, and stalls the other units.

## 4.6 Sequential Instruction Prefetcher

A sequential instruction prefetcher (or an instruction stream buffer) consists of an address register, an incrementor, and one 32Byte cache line buffer. The stream buffer is accessed in parallel with I-cache in one cycle. The stream buffer initiates a prefetch on an I-cache miss and stops when a branch or another cache miss are encountered. On a stream buffer hit the line is loaded into the I-cache.

## 4.7 Multilevel Branch Predictor

In this work, different implementations of a multilevel branch predictor are studied. The multilevel branch predictor consists of a table in which each entry contains  $2^{K-1}$  branch target addresses and frequency counters. The predictor receives the current branch address, branch direction, and branch target address from CPU, updates its internal prediction table, and predicts branch targets for prefetching. The direct-mapped and set-associative table organizations will be studied.

## 4.8 Overall System Organization under Study

The effect of MLBP-based prefetching and its interaction with sequential prefetching are studied by analyzing performance of four systems described below. These systems add MLBP and/or sequential prefetch units to the base system and allow the miss rate and CPU time changes to be observed.

- **Baseline Architecture (B)**  
This architecture performs no prefetching and models a simple I-cache. It is used to as the basis for comparing the improvement from prefetching.
- **Baseline plus Instruction Stream Buffer (BI)**  
This architecture, adds an instruction stream buffer in parallel with the instruction cache. A CPU address is issued to both I-cache and stream buffer. This stream buffer targets sequential instruction prefetching.
- **Baseline plus MLBP-based Prefetch Unit (BP)**  
This architecture, adds a 2-level MLBP-based prefetch address generator and two 1-line buffers to store prefetched lines. This architecture attempts to generate a prefetch request for *every* branch, conditional or unconditional, using an earlier branch as a trigger. It targets branch targets for prefetching.  
The two buffers are used cyclically and a new block overwrites the oldest block. A prefetch address is issued to the L2 cache. A CPU address is issued to both I-cache and the two prefetch buffers. If an address is present in a prefetch buffer but not in instruction cache, the block is loaded into the I-cache.
- **Baseline plus Instruction Stream Buffer plus Prefetch Buffer (BIP)**

This architecture combines the I-cache with the two types of prefetch units. Figure 2 shows this system organization. A CPU address is checked in all three units. The instruction cache is loaded if a block is found in either the stream buffer or the MLBP-based buffers. A true instruction cache miss is considered to occur when a requested block is not present in any of the three units. The stream buffer will issue a prefetch for the next sequential block when either a true instruction cache miss occurs or when a hit occurs in either the stream buffer itself or the MLBP-based buffers.

## 5 Experimental methodology

Trace-driven simulation is used to evaluate the effect of MLBP-based prefetching. Six benchmarks are used: five SPEC92 benchmarks with high I-cache miss rates and tpm-C Oracle data base manager (dbm). The SPEC benchmarks are compiled on an SGI system using SPEC scripts. Pixie software [CHKW86] is used to generate an instruction trace. The data base manager benchmark is a 10-minute sample of one data base manager process. This process is one of thirty such processes simultaneously executing.

The programs were compiled and traced using MIPS-I instruction set, compiler optimizations (-O3 flag) and statically linked libraries (non-shared flag). The one major deviation from the SPEC scripts was in compilation and tracing of `gcc`. We used only one copy of input files instead of five and “merged” these files into a single C program to avoid multiple startups. This lead to a large increase in unconditional branches. Also, both `gcc` and `cc1` compilation were traced. Table 1 shows the basic benchmark statistics.

Program Name	Instruction Counts (Millions)	Percentage of Conditional Branches	Percentage of Unconditional Branches
<b>doduc</b>	1,350	7.312	0.993
<b>fpppp</b>	2,139	0.945	0.115
<b>gcc</b>	477	2.595	8.823
<b>sc</b>	72	19.284	1.279
<b>xlisp</b>	1,179	14.500	4.519
<b>dbm</b>	46	13.3	2.4

**Table 1.** Benchmark statistics.

## 6 Prediction Accuracy

Multi-level branch prediction accuracy is key to prefetching. We start by examining a correlated branch predictor used for “regular” branch prediction and the effect of the table size on its performance. Next, a direct-mapped 2-level branch predictor is studied

while varying the size of its table, followed by a fully-associative predictor. The size of a direct-mapped table ranges from 32 to 4K entries and from 64 to 512 entries for a fully-associative table. The latter are kept smaller to approximate the same access time since associative lookup is slower. Finally, the 1- and 3-level predictors are investigated in addition to the 2-level predictor.

### 6.1 Correlated Branch Predictor Accuracy

The standard correlated predictor is used and the effect of table size on prediction accuracy for each benchmark program is summarized in Figure 3. The prediction accuracy reaches 90% and above for all benchmarks once a table size of around 1K entries is reached. A 1K entry predictor is used for CPU branch prediction in the rest of experiments.

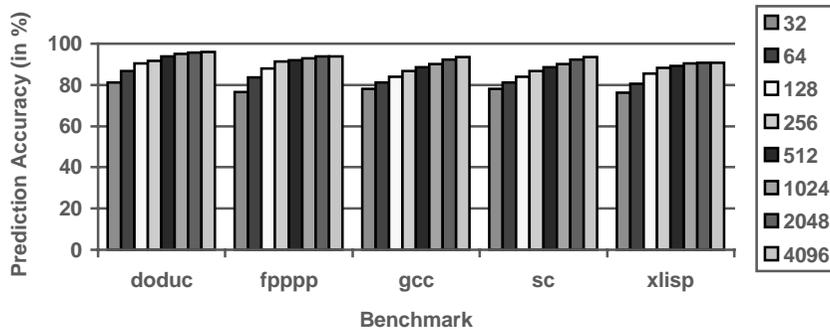


Figure 3. Prediction accuracy for different size correlated predictors

### 6.2 Direct-mapped, Multi-level Prediction

Figure 4 summarizes the overall prediction accuracy of selected benchmark programs relative to the total number of taken branches (since only taken branches are predicted)<sup>2</sup>. The prediction accuracy increases smoothly with the increase in table size in most cases and ends up in the range of 55 and 85% for all programs except gcc. The problem with gcc is a large number of register-based jumps. While the results are not as high as for correlated branch prediction (Figure 3), they are over 3 branch levels and predicting a branch target address as well! The accuracy may still improve instruction prefetching since there is a high probability of finding data in the cache.

<sup>2</sup> Oracle results for some of the cases are not available

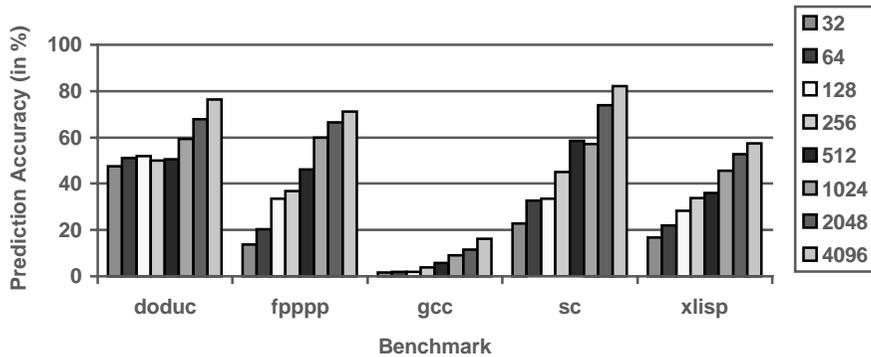


Figure 4 Direct-mapped prediction accuracy for various table sizes

### 6.3 Fully-associative prediction

The performance of a fully-associative predictor is presented next for table sizes of 64, 256 and 512, and prediction levels of 1, 2 and 3. All entries in the mBTB's fully associative table are compared in parallel to the address, just as in an associative cache or BTB. The prediction accuracy for the fully-associative predictor is shown in Figure 9 as a function of table size and predictor level (see size-level caption). It is clear that prediction accuracy drops with decrease in table size and increase in the number of levels. Overall, however, a 256-entry, 2-level predictor does quite well and the results exceed those for larger, direct-mapped predictors, except for gcc.

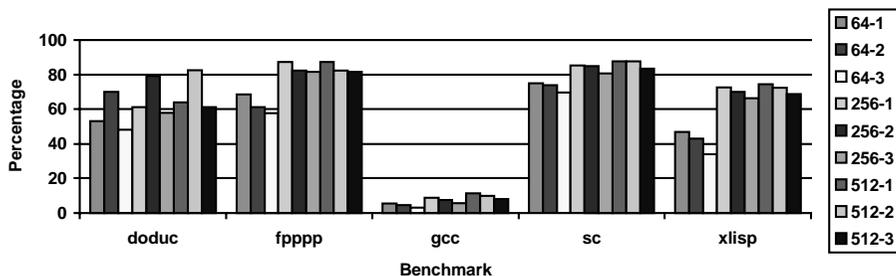


Figure 5 Fully-associative predictor accuracy for 1, 2, and 3-level prediction

## 7 History-Based Prediction

Everything discussed so far used the MLBP unit indexed by the (branch) program counter. The predictor works well, as shown above, but is not using a hardware table entry efficiently for unconditional branches. In addition, it has been shown elsewhere that (standard) history-based branch prediction can be more accurate than PC-based prediction. In this section a different MLBP unit using past branch history for predictor indexing is studied and compared to the PC-based unit described in previous sections. An additional advantage of this unit is that it can be extended to handle more

levels at a lower cost since predictor entry size is independent of the number of prediction levels.

## 7.1 History-Based Prediction Method

In a standard, history-based predictor branch history is used to index an array of counters. A similar method can be applied to MLBP. In order to do so, we extend the Branch History Register (BHR in Figure 1) to  $N+k$  bits (to the left), where  $k$ =number of prediction levels and  $N=\log_2\text{PredictorSize}$ . The resolved branches will continue to update the BHR and shift it to the left. The leftmost  $N$  bits will be used as the index into the predictor table with  $2^N$  entries.

Let us again concentrate on a 2-level predictor. A single-lookup target address predictor, given a certain branch history in the  $N$ -bit BHR, directly looks up and predicts the outcome and the two target addresses, the taken and not taken, of a branch 2 levels past the “Current Branch”. Note that none of the “ $k$ ” intervening branch outcomes are even considered. The reasoning is that previous history already sufficiently differentiates between different paths.

Each predictor entry thus contains two addresses and a counter. The counter predicts whether this future branch is taken or not. And, therefore, which of the two addresses to predict as next. The address is passed to the next prefetch unit as described above. Only a direct-mapped, 2-level predictor is reported in this paper. However, a much larger number of levels, up to 8, has been predicted using same predictor.

## 7.2 Predictor accuracy

Four benchmarks with some of the worst branch prediction behavior are used to present the accuracy of the new predictor. Only predictor sizes of 1K and 4K entries are used, skipping smaller-size predictors. In addition, one larger predictor size of 16K entries is presented to investigate the effect of size. The results are shown in Figure 6.

As before, increasing predictor size helps significantly. For the largest predictor size even the worst benchmarks, gcc and Oracle, are beginning to show reasonable accuracy. The results can be directly compared with PC-based predictor in Figure 7. Fpppp shows an improvement of over 30%, gcc of over a 100% for smallest predictor. The improvement decreases with size of predictor. An interesting event occurs in the case of sc, where a 4K-entry, history-based predictor has worse accuracy.

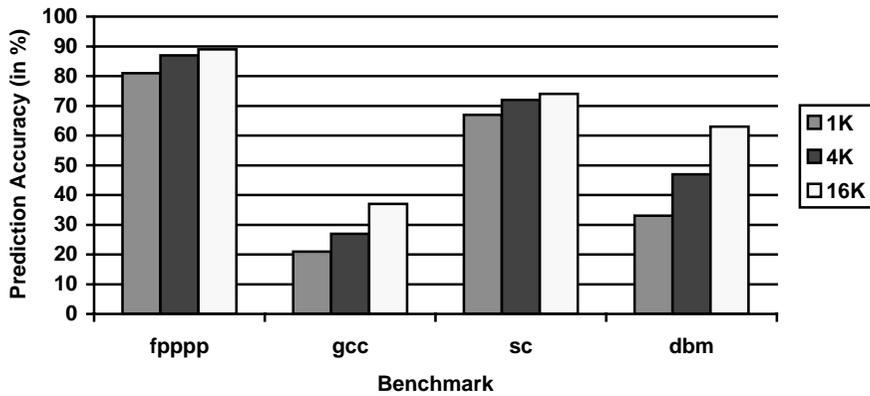


Figure 6. Prediction accuracy of history-based predictors

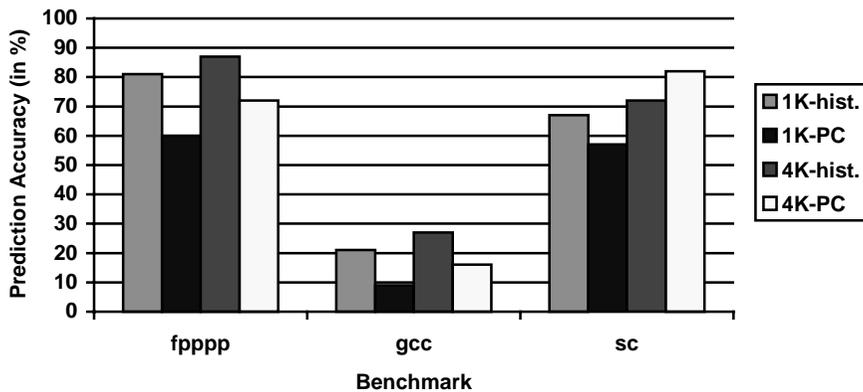


Figure 7 Comparison of 2-level PC- and History-based Predictors

## 8 Prefetching Results

As shown above, multi-level branch prediction works fairly well in predicting the branch target addresses. Next, the MLBP is used to provide an address prediction which is then used to initiate instruction prefetch. We start by simulating several I-cache organizations to assess the baseline performance, followed by the I-cache prefetching mechanisms described above. Finally, the effect of instruction fetch on execution time and issue rate is investigated.

## 8.1 Instruction Cache Miss Rates

I-cache sizes of 8 and 16KBytes and line sizes of 32 and 64Bytes are studied. The size and associativity of the I-cache are kept small to guarantee high clock rates. **Table 2** shows the results. The miss rates range from below 1 to 10%. While doubling the associativity or line size leads to a miss rate reduction, overall the miss rates remain unacceptably high.

The percentage of cache misses caused by branch targets is shown in Table 3. These misses will be targeted by MLBP prefetching. It shows that the branch target misses can be a very high fraction of I-cache misses. But they are not very sensitive to associativity and cache size, except for fpppp. Fpppp has approximately 2% of instruction cache misses caused by branch targets.

### GCC

Cache Size	Line Size	Associativity	Miss Rate	Cache Size	Line Size	Associativity	Miss Rate
8K	32	1	7.26	16K	32	1	4.86
8K	32	2	6.65	16K	32	2	4.87
8K	64	1	3.89	16K	64	1	2.59
8K	64	2	3.57	16K	64	2	2.59

### XLISP

8K	32	1	1.24	16K	32	1	1.15
8K	32	2	0.40	16K	32	2	0.12
8K	64	1	1.05	16K	64	1	0.73
8K	64	2	0.36	16K	64	2	0.28

### SC

8K	32	1	1.71	16K	32	1	1.07
8K	32	2	1.14	16K	32	2	0.58
8K	64	1	1.27	16K	64	1	0.80
8K	64	2	0.81	16K	64	2	0.42

### FPPPP

8K	32	1	10.51	16K	32	1	6.92
8K	32	2	10.30	16K	32	2	6.66
8K	64	1	5.35	16K	64	1	3.53
8K	64	2	5.24	16K	64	2	3.40

### DODUC

8K	32	1	3.36	16K	32	1	1.48
8K	32	2	2.75	16K	32	2	1.31
8K	64	1	1.96	16K	64	1	0.85
8K	64	2	1.59	16K	64	2	0.73

### DBM

8K	32	1	10.9	16K	32	1	9.9
8K	32	2	10.7	16K	32	2	9.5

**Table 2** Instruction cache miss rates

Organization	doduc	fpppp	gcc	sc	xlisp
8K, Direct-mapped	14.5	2.38	9.86	47.7	42.7
8K, 2-way assoc.	12.4	2.21	9.44	48.1	44.3
16K, Direct-mapped	14.9	2.05	9.25	46.3	42.8
16K, 2-way assoc.	13.0	1.72	7.50	49.4	45.4

**Table 3** Percentage of branch target misses for 32Byte line

## 8.2 The Performance Impact of Prefetching

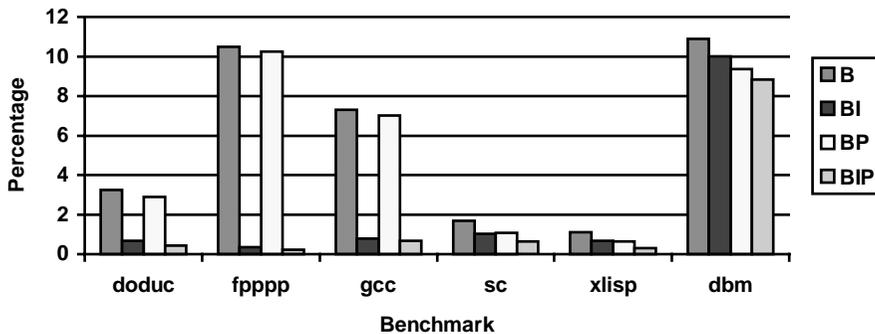
The impact of prefetching on system performance is measured by its effect on I-cache miss rate and the reduction in CPU stall cycles caused by cache misses. The baseline architecture (B) does no instruction prefetching, the second architecture (BI) adds a one-line instruction stream buffer, the third architecture (BP) adds a 2-level MLBP-based prefetch generator and two 1-line prefetch buffers, and the last architecture (BIP) adds both a stream and a MLBP-based prefetch units.

First, results for the direct-mapped, 4K-entry predictor are presented, followed by a fully-associative, 256-entry MLBP. The results are shown for an 8KB, direct mapped instruction cache organization with 32Byte lines, unless otherwise specified. A cache configuration is represented by a triplet (cache size, line size, associativity). First, the miss rate change is shown followed by the CPU time analysis for the four architectures.

### 8.2.1 Direct-mapped MLBP Implementation

#### 8.2.1.1 Prefetch Effect On I-Cache Miss Rate

Recall that with prefetching, a miss occurs when neither the I-cache or a prefetch buffer contains the requested line. The case when a prefetch buffer has issued a request but a line is not yet available is counted as a hit. The stall cycles spent waiting in this case will be shown in the next section. Figure 8 summarizes the I-cache miss rate for the four architectures and Figure 9 shows the I-cache miss rate change.



**Figure 8.** I-cache miss rates

For *doduc*, *fpppp*, and *gcc*, the stream buffer is more efficient than multilevel branch prediction in removing cache misses. For *sc* and *xlisp*, the effect of the two organization is quite close. The stream buffer sequential prefetching reduces the miss rate by 80 to 97% for 3 benchmarks, but for the other three it only removes 40% or fewer of the misses. Branch target prefetching removes close to an additional 40% of the misses in the latter two benchmarks, while giving 3 to 11% improvement in the former three benchmarks. When the two prefetching methods are used together, the effect is very close to purely additive. For *dbm* the miss rate reduction is not very large in all the cases, but very important as we will show in the next section.

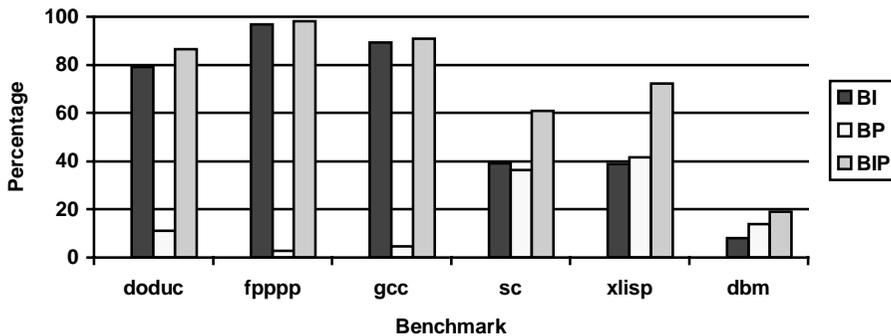


Figure 9. Percentage of baseline architecture (B) misses removed

Stream (BI) and stream plus direct-mapped MLBP (BIP) prefetching results for an additional three cache configurations are shown in Table 4. The first row repeats the results shown in Figure 9. The effect of associativity and size on cache miss rate reduction from prefetching is quite small in this range of cache configurations.

Cache config.	doduc		fpppp		gcc		sc		xlisp		dbm	
	BI	BIP	BI	BIP	BI	BIP	BI	BIP	BI	BIP	BI	BIP
8K,32,1	79	86	97	98	89	91	39	61	39	72	8.1	19.0
8K,32,2	82	88	97	98	89	91	39	60	43	64	8.2	18.3
16K,32,1	78	86	97	98	90	91	41	60	38	74	8.4	19.0
16K,32,2	80	87	97	98	91	92	37	59	46	66	8.6	18.7

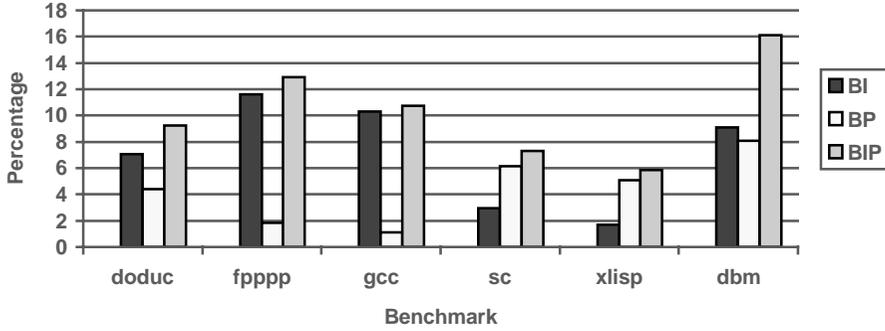
Table 4. I-cache miss rate reduction from baseline architecture

### 8.2.1.2 The Effect on Execution Time

The stream buffer significantly reduced the instruction cache miss rates for *doduc*, *fpppp*, and *gcc* while MLBP-based prefetching significantly helped *sc* and *xlisp*. The combination of two methods produced additive results. Next, the actual execution time change from prefetching is evaluated. Figure 10 summarize the base execution time reduction due to prefetching. The range of improvement from stream buffer prefetching is 1.7 to 10.3%, 1 to 8% from MLBP prefetching, and 5.8 to 16% for combined prefetching.

The effect is smaller than one might expect from the miss rate reduction. As shown in the next section, there is still a significant stall component in stream buffer prefetching

indicating it may not be started early enough and may need more bandwidth. The relative performance of stream and MLBP-based prefetching also changes. For *sc* and *xlisp*, multilevel branch prediction based prefetching becomes more efficient than stream buffer, for *dbm* they are very close and completely additive, and for *doduc* the difference is significantly reduced. This reflects two facts. First, in *fpppp* and *gcc* sequential misses dominate, while in *sc*, *xlisp* and *dbm* a large fraction of instruction cache misses, up to 40%, are caused by branch targets (see Table 3). Second, MLBP allows prefetches to be issued earlier than in stream buffer prefetching.



**Figure 10.** Relative CPU time decrease

CPU time reduction from instruction stream buffer alone and in conjunction with MLBP-based prefetching for BI and BIP for the other cache configurations is summarized in Table 5. The associativity and cache size make a much bigger difference in the CPU time reduction as compared to their effect on miss rates. Larger caches and higher associativity reduce the miss rate but benefit less from the prefetching improvement. However, except for *xlisp*, there is still a noticeable improvement from combined prefetching methods for all cache configurations.

Cache	doduc		fpppp		gcc		sc		xlisp		dbm	
Config.	BI	BIP	BI	BIP	BI	BIP	BI	BIP	BI	BIP	BI	BIP
8K,32,1	7.1	9.2	11.6	12.9	10.3	10.8	3.0	7.3	1.7	5.9	9.1	16.1
8K,32,2	6.3	9.6	11.6	12.8	9.7	10.2	2.1	5.2	1.1	2.1	9.1	15.7
16K,32,1	4.1	6.0	10.1	11.1	8.2	8.5	2.2	4.7	1.5	5.6	8.9	15.1
16K,32,2	3.7	5.6	10.0	10.8	8.1	8.4	1.2	2.8	0.5	0.8	8.8	14.6

**Table 5.** Percent CPU time reduction from stream buffer and MLBP prefetching

### 8.2.1.3 CPU Cycle Breakdown

The following approximation was used to derive benchmark execution times above. The processor stalls on an I-cache miss (6 cycles), any taken branch (1 cycle), and a branch misprediction (5-cycles). Other stalls are not modeled, e.g L2 cache misses, conflicts with primary D-cache misses, or execution unit stalls. (Equation 1) gives a total benchmark execution time accounting for the stalls we model.

$$T_{cpu} = T_{hb} + T_{tb} + T_{cm} + T_{mp} + T_{pr} - T_{ovlp} \quad (\text{Equation 1.})$$

Where:

**T<sub>hb</sub>**- time to fetch all four-instruction blocks (not all are executed due to branches)

**T<sub>tb</sub>** - taken branch stall time when the branch predictor predicts branch taken

**T<sub>cm</sub>** - stall time for servicing instruction cache misses from the L2 cache

**T<sub>mp</sub>** - branch misprediction stalls

**T<sub>pr</sub>** - stall on an issued prefetch which is not serviced when a I-cache miss occurs

**T<sub>ovlp</sub>** - cache miss and mispredicted branch overlap

Event counts for the **T<sub>cpu</sub>** components are collected during simulation, multiplied by the corresponding delay time, and added up to obtain the total. **T<sub>ovlp</sub>** is the time over-charged for a misprediction and a cache miss at the same time, thus **T<sub>ovlp</sub>** is subtracted. Table 6 summarizes the results, showing the **T<sub>cpu</sub>** and **T<sub>pr</sub>** directly while showing event counts for other categories. For an event 'xx', the stall time **T<sub>xx</sub>** is found by multiplying the 'xx' column of the table by the stall duration. For example, the cache miss stall time **T<sub>cm</sub>** can be found by multiplying the 'cm' column of the table by the L2 latency of 6 cycles.

The results clearly demonstrate the need to reduce I-cache misses. The two components dominating the CPU time are the half-block fetches followed by cache miss or taken branch stalls. Recall that a 6-cycle cache miss stall is used. An 8-cycle stall would make cache misses the second largest CPU time component. At the same time, the programs where stream buffer prefetching had a large effect on miss rate show a large amount of prefetch stalls. The MLBP-based prefetching alone has no prefetch stalls and demonstrates the effectiveness of prefetching across branches, even in the data base manager.

#### DODUC

System	Tcpu	hb	tb	cm	mp	Tpr	ovlp
B	736124	391027	67422	43997	4385	0	1282
BI	692759	391027	67422	9201	4384	165244	1217
BP	709123	391027	67422	39101	4384	0	1235
BIP	668144	391027	67422	5948	4384	159132	1176

#### FPPPP

B	1907158	547121	12639	224268	1248	0	782
BI	1685369	547121	12639	7279	1248	1080541	1080
BP	1878085	547121	12639	219030	1248	0	726
BIP	1660889	547121	12639	4464	1248	1072525	1072

#### GCC

B	375184	146921	3070	34815	4182	0	943
BI	336562	146921	3070	3749	4182	147525	989
BP	371776	146921	3070	33472	4182	0	849
BIP	334838	146921	3070	3230	4182	148192	913

#### SC

B	44442	26355	9378	1212	458	0	152
BI	43343	26355	9378	737	458	1750	147
BP	42165	26355	9378	772	458	0	147
BIP	41737	26355	9378	473	458	1755	144

#### XLISP

B	657086	424711	95440	12994	13444	0	721
BI	647899	424711	95440	7941	13444	20334	623
BP	629181	424711	95440	7593	13444	0	684
BIP	625054	424711	95440	3613	13444	25543	615

**DBM**

B	46691	13785	3352	4658	1075	0	737
BI	42439	13785	3352	4281	1075	7559	726
BP	42896	13785	3352	4009	1075	0	714
BIP	39184	13785	3352	3769	1075	7390	707

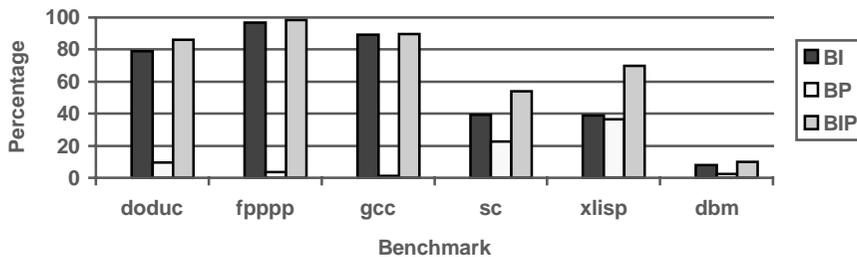
**Table 6. CPU time components for direct-mapped predictor ( in thousands)**

### 8.2.2 Fully-Associative Implementation

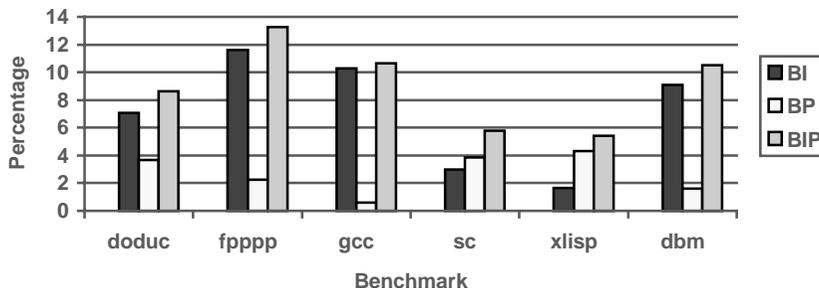
A fully-associative MLBP implementation uses a tagged table of 256 entries, otherwise it is identical to the direct-mapped case.

Figure 11 and Figure 12 show the percent reduction in the miss rate and CPU time, respectively. Table 9 shows the CPU time breakdown.

The results are very close to the CPU times for the 4K-entry direct-mapped implementation (usually within 1%), except for the dbm benchmark. The dbm benchmark suffers a significant reduction in its ability to perform MLBP prediction and address generation. This indicates that a tagless table may work better as it will make a prediction even if the history belongs to another reference. Insufficient size to hold the predictions is another reason. The miss rate reduction is also seriously affected in sc but it still generates a noticeable decrease in the CPU time.



**Figure 11. Misses removed by a fully-associative predictor**



**Figure 12.** CPU time decrease with a fully-associative predictor

Table 7 and Table 8 summarize the effect of instruction stream buffer and MLBP prefetching on cache miss rate and CPU time for the other three I-cache configurations. Again, the miss rate reduction is not very sensitive to associativity or size. The CPU time is affected in about the same way as in the direct mapped case.

Cache config.	doduc		fpppp		gcc		sc		xlisp		dbm	
	BI	BIP	BI	BIP	BI	BIP	BI	BIP	BI	BIP	BI	BIP
8K,32,1	79	86	97	98	89	90	39	54	39	70	8.1	10
8K,32,2	82	87	97	99	89	90	39	49	43	62	8.2	9.6
16K,32,1	78	82	97	98	90	90	41	51	38	71	8.4	9.8
16K,32,2	80	82	9	99	91	91	38	45	46	68	8.6	9.4

**Table 7.** Cache miss rate reduction (%) for stream buffer and MLBP prefetching.

Cache config.	doduc		fpppp		gcc		sc		xlisp		dbm	
	BI	BIP	BI	BIP	BI	BIP	BI	BIP	BI	BIP	BI	BIP
8K,32,1	7.1	8.7	11.3	13.3	10.3	10.7	3.0	5.8	1.7	5.4	9.1	10.5
8K,32,2	6.3	8.5	11.6	13.1	9.7	10.1	2.1	3.5	1.1	2.0	9.1	10.0
16K,32,1	4.1	5.3	10.1	11.3	8.2	8.4	2.2	3.5	1.5	5.2	8.9	9.8
16K,32,2	3.7	4.5	10.0	11.0	8.1	8.3	1.2	1.7	0.5	0.8	8.8	9.3

**Table 8.** CPU time reduction (%) from stream buffer and MLBP

DODUC							
System	T <sub>cpu</sub>	hb	tb	cm	mp	T <sub>pr</sub>	ovlp
B	736123	391027	67422	43997	4385	0	1282
BI	692759	391027	67422	9201	4384	165244	1217
BP	713482	391027	67422	39838	4384	0	1235
BIP	672463	391027	67422	6165	4384	162509	1171

FPPPP							
System	T <sub>cpu</sub>	hb	tb	cm	mp	T <sub>pr</sub>	ovlp
B	1907158	547121	12639	224268	1248	0	782
BI	1685369	547121	12639	7279	1248	1080541	1080
BP	1871264	547121	12639	217865	1248	0	687
BIP	1654249	547121	12639	3627	1248	1070563	1070

GCC							
System	T <sub>cpu</sub>	hb	tb	cm	mp	T <sub>pr</sub>	ovlp
B	375184	146921	3070	34815	4182	0	943
BI	336562	146921	3070	3749	4182	147525	989
BP	373388	146921	3070	34412	4182	0	872
BIP	335231	146921	3070	3557	4182	147022	932

SC							
System	T <sub>cpu</sub>	hb	tb	cm	mp	T <sub>pr</sub>	ovlp
B	44442	26355	9378	1212	458	0	152
BI	43343	26355	9378	737	458	1750	147
BP	43020	26355	9378	938	458	0	150
BIP	42295	26355	9378	556	458	1842	147

XLISP							
System	T <sub>cpu</sub>	hb	tb	cm	mp	T <sub>pr</sub>	ovlp
B	657086	424711	95440	12994	13444	0	721

BI	647899	424711	95440	7941	13444	20334	623
BP	633549	424711	95440	8236	13444	0	697
BIP	627535	424711	95440	3915	13444	25846	607
<b>DBM</b>							
B	46686	13785	3352	4658	1075	0	738
BI	42440	13785	3352	4281	1075	7552	728
BP	45949	13785	3352	4548	1075	0	732
BIP	41802	13785	3352	4186	1075	7550	722

**Table 9.** CPU time breakdown for fully-associative predictor (in thousand cycles)

## 9 Related work

“One-level” branch prediction algorithms have been widely researched. High prediction accuracy has been achieved using many innovative ideas. Some fundamental methods for doing this are described next. A branch predictor using a table of 2-bit saturating up-down counters and addressed by PC was proposed in [Smit81] to keep track of dynamic branch information. A two-level branch predictor proposed in [YePa91] uses two levels of branch history to predict branch direction. Hybrid branch predictors composed of several single scheme predictors and a way to select one of them at a particular time have been proposed [McFa89, ChHP95].

Instruction prefetching has been addressed in the past primarily through sequential prefetch or code layout techniques [Smit82, DEC82, SmHs92, HwCh89, McFa89, Joup90, ERPR95, UNMS95, XiTo96, LBCG95 Intel93]. Sometimes instruction prefetch was initiated along both possible branch paths [Intel93]. Compiler assistance can help by code layout or by identifying the end of a basic block to stop prefetching [HwCh89, McFa89, XiTo96]. The main improvement comes from adding a sequential prefetcher as has been done in many existing machines. The problem in existing approaches is that prefetching *stops* when a branch instruction is encountered and the predicted address is non-sequential.

An approach to prefetch speculatively along both paths while waiting for a branch to be resolved has been used (Intel Pentium™), but was aimed at getting the instructions from the I-cache to the Decode unit. [ULMS95] consider sequential prefetching for small I-caches. “Optimistic” prefetch policy is used in [LBCG95] to predict a branch and prefetch down the predicted path.

Multilevel instruction prefetching can be accomplished using the Lookahead Program Counter [BaCh91], which advances forward one instruction per cycle using a standard predictor. For small-size predictors we have shown this multiple-lookup prediction to be less accurate than single lookup [Sham98]. True multilevel branch prediction has been proposed in order to speculatively fetch instructions along the most likely path and to “collapse” them into a contiguous sequence [YMP93, DuFr95, CMMP95]. Our multi-level prediction follows these techniques. An interesting solution using a history-based predictor proposed in [RoBS96] uses multiple history table lookups for 3 levels of prediction. A path-predicting approach is used in [MeSC97] predicts a complete path through N branches using an automaton similar to the standard predictor but with  $2^N$  states.

Finally, a related approach [SJSM96] was independently developed to fetch the next two cache line even if they contain branches. Our approach differs in its intent to

prefetch to the I-cache and eliminate cache misses. It also allows any lookahead distance to be used.

## 10 Conclusions

A new, non-sequential instruction prefetching method using multi-level branch target prediction (MLBP) is developed in this work and its effectiveness studied. It predicts the branch direction and a branch target across K branches for a K-level prediction. We concentrate on a 2-level prediction to balance hardware complexity and performance. However, the concept can be extended to more levels. The behavior of five SPEC92 benchmark programs with highest instruction cache miss rates and of Oracle tpm-C benchmark is analyzed. Integer and data base management benchmarks showed the largest improvements, up to 15% CPU time reduction, from prefetching. Benefits are greatest for small, fast caches.

The MLBP prediction accuracy was found to be quite close for a fully-associative 256-entry and direct-mapped, 4K-entry predictors, about 70% on average. This is low by branch prediction standards but may be sufficient for prefetching. Part of the reason for low accuracy is our implementation choice of tracking only taken branches. It was done to make the MLBP implementation simpler in order for it to keep up with the CPU and require less hardware.

Another approach using a novel history-based predictor was described which uses simpler hardware and is easier to extend to more levels. Its accuracy, especially for smaller predictors, was shown to be typically much higher. Up to 8-level prediction and prefetching study has been presented in [Sham98].

The average fraction of cache misses caused by branch targets in the benchmarks studied is about 25% for an 8KB I-cache with 32 byte line size. Their elimination can lead to a significant miss rate reduction when MLBP is used. The effect on CPU performance is smaller, up to 8%. For a larger I-cache miss service time the effect on CPU performance will increase.

Traditional sequential prefetching works well in some programs and our additional prefetching hardware provides only a small improvement. In other benchmarks the MLBP-based prefetching produces a larger effect than sequential prefetching. Its success is due in part to the longer lookahead afforded by the multi-level prediction.

The MLBP and sequential instruction prefetching complement each other. They can be used together to effectively remove both sequential and branch target instruction cache misses. The sequential prefetch is often only partly effective because it is not initiated early enough. A larger stream buffer can help but will consume a lot of L2 cache bandwidth and, given the frequency of branches, may not help after all. MLBP prefetching does not have this problem. Combined improvement reaches 16%.

Overall, MLBP-based prefetching is a new, unique way to perform non-sequential instruction prefetch for an arbitrary number of branch levels. Its accuracy and resulting performance improvement are very encouraging and warrant further investigation.

## 11 References

- [**BaCh91**] Jean-Loup Baer and Tien-Fu Chen. "An effective on-chip preloading scheme to reduce data access penalty", *Supercomputing'91*, pp. 176--186. November 1991.
- [**CaGr95**] B. Calder and D. Grunwald, "Next Cache Line and Set Prediction", *International Symposium on Computer Architecture*, pp.287--296, May 1995.
- [**ChHP95**] P. -Y. Chang, E. Hao, and Y. N. Patt.: *Alternative Implementations of Hybrid Branch Predictors*. In: *28th ACM/IEEE International Symposium on Microarchitecture*, Nov. 1995.
- [**CHKW86**] Fred Chow, A. M. Himelstein, Earl Killian and L. Weber, "Engineering a RISC Compiler System," *IEEE COMPCON*, March 1986.
- [**CMMP95**] T.M. Conte, K. N. Menezes, P.M. Mills, and B.A. Patel, "Optimization of Instruction Fetch Mechanism for High Issue Rates", *International Symposium on Computer Architecture*, pp.333--344, May 1995.
- [**DuFr95**] Simonjit Dutta and Manoj Franklin, "Control Flow Prediction with Tree-Like Subgraphs for Superscalar Processors". *International Symposium on Microarchitecture (Micro-28)*, pp. 258--263, November 1995.
- [**ERPR95**] J. H. Edmondson, P. R. Rubinfeld, Ronald Predton, and Vidya Rajagopalan. "Superscalar Instruction Execution in the 21164 Alpha Microprocessor". *IEEE Micro*, Vol. 15, No. 2, April 1995
- [**DEC82**] *VAX Hardware Handbook*, Digital Equipment Coporation, 1982.
- [**EvCP96**] M. Evers, P-Y Chang, and Y. N. Patt, "Using Hybrid Branch predictors to Improve Branch Prediction Accuracy in The Presence of Context Switches", *International Symposium on Computer Architecture*, pp. 3--13, May 1996.
- [**HePa96**] John L. Hennessy and David A. Patterson, "Computer Architecture, a Quantative Approach", 2nd edition, pp. 465, 1996.
- [**HwCh89**] W.-M. Hwu and P. Chang, "Achieving High Instruction Cache Performance with an Optimizing Compiler", *International Symposium on Computer Architecture*, pp. 242-251, May 1989.
- [**Inte93**] *Pentium Processor User's Manual, Vol.1: Pentium Processor Data Book*. Intel, 1993.
- [**JoGr97**] D. Joseph and D. Grunwald, "Prefetching using Markov Predictors," *Proceedings of the ACM*, 1997
- [**Joup90**] Norman P. Jouppi. "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers", *International Symposium on Computer Architecture*, pp. 364--373, May 1990.
- [**JoWi94**] Norman P. Jouppi and Steven J.E. Wilton, "Trade-offs in Two-level On-chip caching", *International Symposium on Computer Architecture*, pp. 34-45, April 1994.
- [**LBCG95**] D. Lee, J.-L. Baer, B. Calder, D. Grunwald "Instruction Cache Fetch Policies for Speculative Execution", *International Symposium on Computer Architecture*, pp. 357-367, May 1995.
- [**McFa89**] S. McFarling, "Program Optimization for Instruction Caches", *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 183-191, 1992.
- [**McFa93**]. S. McFarling, "Combining Branch Predictors," *Technical Note TN-36*, DEC-WRL, June 1993
- [**MeSC97**] K. N. Menezes, S. W. Sathaye and T. M. Conte, "Path Prediction for High Issue-Rate Processors," *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques (PACT'97)*, Nov. 1997.

- [PaSR92]** S-T Pan, K. So, and J.T. Rameh, "Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation", International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 76-84, October 1992.
- [RoBS96]** Eric Rotenberg, Steve Bennett, and James E. Smith, "Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching", 29th Annual International Symposium on Microarchitecture, pp. 24-34, December 1996.
- [SaPN96]** Ashley Saulsbury, Fong Pong and Andreas Nowatzyk. "Missing the Memory Wall: the Case for Processor/Memory Integration". Computer Architecture News, Vol. 24, No. 2, pp.90-101, May, 1996.
- [SeLM96]** S. Sechrest, C-C Lee and T. Mudge. "Correlation and Aliasing in Dynamic Branch Prediction", International Symposium on Computer Architecture, pp. 22--32, May 1996.
- [Sham98]** A. Shameer, "Multi-level Branch Target Buffer for Prefetching Across Branches", M.S. Thesis, EECS Dept., University of Illinois at Chicago, August 1998
- [SJSM96]** A. Sez nec, S. Jourdan, P. Sainrat, P. Michaud. "Multiple-Block Ahead Branch Prediction", International Symposium on Computer Architecture, pp. 116-127, May 1996.
- [SmLet84]** A. J. Smith and J. Lee, "Branch prediction strategies and branch-target buffer design," Computer, January 1984.
- [Smit81]** J. E. Smith. "A Study of Branch Prediction Strategies." Proceedings of the 8th International Symposium on Computer Architecture, pp.135-148, May, 1981.
- [SmHS92]** J.E. Smith and W.-C. Hsu, "Prefetching in Supercomputer Instruction Caches", International Supercomputing Conference, pp. 588-597, July 1992
- [UNMS95]** R. Uhlig, D. Nagle, T. Mudge, S. Sechrest, and J. Emer, "Instruction Fetching: Coping with Code Bloat", International Symposium on Computer Architecture, pp. 348--356, May 1995.
- [Veid97]** A. Veidenbaum, "Instruction Cache Prefetching using Multilevel Branch Prediction," International Symposium on High Performance Systems, October 1997.
- [XiTo96]** C. Xia and J. Torrrillas, "Instruction Prefetching of Systems Codes with Optimized Layout for Reduced Cache Misses", International Symposium on Computer Architecture, pp. 271--283, May 1996.
- [YePa91]** T.-Y. Yeh and Y. N. Patt. "Two Level Adaptive Branch Prediction." 24th ACM/IEEE International Symposium on Microarchitecture, Nov. 1991.
- [YeMP93]** T-Y Yeh, D.T. Marr, and Y. N. Patt, "Increasing Instruction Fetch Rate via Multiple Branch Predictions and a Branch Address Cache", International Conference on Supercomputing, pp. 67-76, July 1993.
- [Zhao96]** Q. Zhao, "Performance evaluation of instruction prefetching using multi-level branch prediction", M.S. Thesis, EECS Dept., University of Illinois at Chicago, October 1996.