Caching and Predicting Branch Sequences for Improved Fetch Effectiveness *

Soner Onder
Dept. of Computer Science
Michigan Technological Univ.
Houghton, MI 49931-1295

Jun Xu
CRHC, Univ. of Illinois
at Urbana-Champaign
Urbana, IL 61801

Rajiv Gupta
Dept. of Computer Science
The University of Arizona
Tucson, AZ 85721

Abstract

A sequence of branch instructions in the dynamic instruction stream forms a branch sequence *if at most one non-branch instruction separates each consecutive pair of branches in the sequence. We propose a branch prediction scheme in which* branch sequence history *is explicitly maintained to identify frequently encountered branch sequences at runtime and when the first branch in the sequence is encountered, the outcomes of the all of the branches in the sequence are predicted. We have designed an implementation of a branch sequence predictor which provides overall misprediction rates that are comparable with the* `gshare` *single branch predictor. Using this branch sequence predictor we have devised a new instruction fetch mechanism. By saving the instructions following the first branch belonging to a branch sequence in a* sequence table*, the proposed mechanism eliminates fetches of nonconsecutive instruction cache lines containing these instructions and therefore delays associated with their fetching is avoided. Experiments comparing the proposed fetch mechanism with a simple fetch mechanism based upon a single branch prediction for* `Spec95` *benchmarks demonstrate that the total number of I-cache lines fetched during execution decreases by as much as 15%, the number of useful instructions per fetched cache line increases by as much as 18%, and the overall IPCs achieved on a superscalar processor increase by as much as 17% for some benchmarks.*

Keywords - *branch sequence prediction, sequence table, fetch bandwidth, speculative execution.*

*This work was done when the authors were at the University of Pittsburgh and was supported by a NSF grants CCR-9704350, CCR-9808590, and EIA-9806525, and Intel Corporation.

1 Introduction

In order to sustain a high instruction issue rate in superscalar processors, one of the problems that must be addressed is that of providing an effective instruction fetch mechanism which can fetch multiple instructions in each cycle. Delays in the fetching of instructions to be executed directly leads to delays in their issuing and therefore a loss of performance. Therefore branch prediction techniques are employed to predict the branch outcome and then fetch those instructions that are likely to be executed next. Significant amount of research has been conducted on developing increasingly accurate branch prediction algorithms [1, 12, 6, 8, 10, 14]. However, there are situations in which even if the branch predictor predicts the outcomes of branches accurately, and the targets of the taken branches are correctly known, delays in the fetching of instructions may be observed.

In the example shown in Figure 1, assume that during the execution of a loop iteration both the conditionals evaluate to false, that is, both the conditional branches are taken, and finally the code in the else part is executed. The two conditional branches are said to form a sequence of length two as their execution is separated by a single instruction that performs a compare (`c == EOLN`). Assuming that the instructions corresponding to the three cases of the *if-statement* begin at cache line boundaries, a simple fetch mechanism that predicts a single branch in a cycle and fetches a single cache line in each cycle can at best fetch the relevant instructions from each of the cases in successive cycles. Thus, the fetching of multiple useful instructions in each cycle may not be possible using a simple fetch mechanism since the instruction blocks fetched contain few useful instructions (a compare and a conditional branch).

In this paper we introduce the notion of a *branch sequence* which is defined as a sequence of branches in

```
while true loop
    c = getchar();
    if (c == EOF)
        ....
    else if (c == EOLN)
        ....
    else
        ....
    end if;
end loop
```

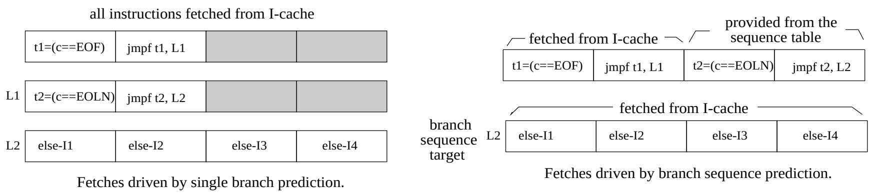


Figure 1. An example of a branch sequence.

the dynamic instruction stream where the consecutive branches in the sequence are separated by at most one non-branch instruction. Thus, branch sequences can be expected to most adversely effect the fetch effectiveness since each fetched cache line can be expected to contain a couple of useful instructions (typically a compare instruction and a branch instruction). Therefore in this paper we propose *branch sequence prediction* in which when the first branch in the sequence is encountered we predict the outcomes of all the branches in the sequence and start fetching instructions from the *branch sequence target*, that is, the target of the final branch instruction in the sequence. *Branch sequence histories* are explicitly maintained for this purpose. Moreover the need for fetching the instructions between the first branch in the sequence and the branch sequence target is eliminated by saving these instructions in a *sequence table*. We view branch sequence prediction as an extension of traditional branch prediction. Instead of always predicting the outcome of a single branch, our technique predicts the outcomes of more than one branch if and only if the branch at hand marks the beginning of a predictable branch sequence.

In the example of Figure 1 when the *first* conditional branch in the sequence is encountered, we predict that the outcome of the branch sequence by predicting the outcomes of *both* conditional branches as taken and start fetching instructions from the branch sequence target which is the *else* part. The need for fetching of instructions corresponding to the second conditional branch `if (c == EOLN)` is eliminated by saving these instructions in the sequence table. As we can observe from Figure 1, branch sequence prediction enables us to provide a greater number of useful instructions in each cycle. While in this example the branch sequence contained two conditional branches, in general a sequence may contain several branches, including unconditional as well as indirect branches. When a sequence is predicted, all the instructions in the sequence are introduced for speculative execution in the processor pipeline. Thus, if the predictions are correct the effectiveness of speculative execution mechanism is greatly enhanced.

In order to evaluate the potential for performance benefit and the cost of branch sequence prediction we conducted some experiments. The percantage of total branch instructions that can benefit from branch sequence prediction, which includes all branches in a branch sequence excluding the first branch, are shown in Figure 2. On an average nearly 34% of all executed branches in `Spec95 integer` and a somewhat lower, yet still significant, 12% number of executed branches in floating-point `Spec95` benchmarks can benefit from our technique. In Figure 3 we show the comparison between a perfect branch predictor and a perfect branch sequence predictor. For various line sizes the perfect branch sequence predictor results in the fetching of 9-39% fewer I-cache lines than the perfect branch predictor for the `Spec95 integer` benchmarks. We found that for the `Spec95 integer` benchmarks the number of instructions that form the branch sequences varies from a minimum of 729 instructions for `compress` to a maximum of 15436 instructions for `gcc` (see Figure 4). For all but four `Spec95` benchmarks this number is less than 2500 instructions. Thus the number of instructions that must be stored in the sequence table is small.

In this section we have demonstrated that branch sequences are encountered very frequently and the potential for improving fetch effectiveness by handling them in a suitable fashion is significant. In section 2 we present a study and analysis of characteristics of branch sequences. In section 3, based upon the characteristics of sequences observed in section 2, we design the hardware for the branch sequence prediction. In section 4 we present the detailed performance evaluation of our fetch mechanism based upon branch sequence prediction. Related work is discussed in section 5.

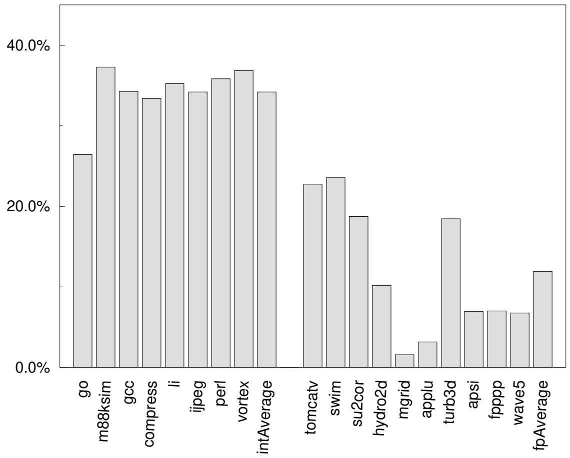


Figure 2. Dynamically executed branches that can benefit from branch sequence prediction.

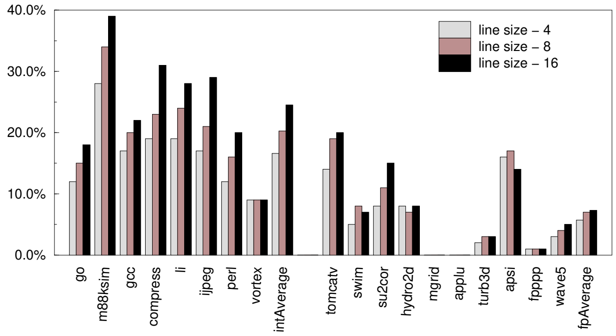


Figure 3. Percentage reduction in cache lines fetched.

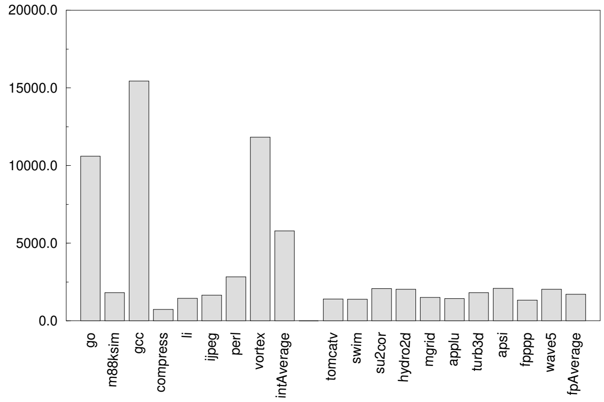


Figure 4. Maximum number of instructions stored in the sequence table.

2 Characteristics of Branch Sequences

Before we design the branch sequence prediction hardware, let us first consider the characteristics of branch sequences that are encountered in practice. There are three main characteristics of branch sequences that will influence the implementation of a branch sequence prediction hardware: (a) sequence lengths; (b) size of sequence history; (c) the types of branch instructions that form the branch sequences.

Sequence lengths. First let us consider the lengths of branch sequences. The length of a branch sequence is defined to be the number of branches in the sequence. This characteristic is important because in a practical hardware implementation it is not possible to maintain histories of branch sequences of arbitrary length. Instead an upper limit must be placed on sequence lengths. Our experiments with `Spec95` benchmarks show that the lengths of the vast majority of branch sequences vary from two to eight. However, the majority of the sequences are of lengths two and three. In Figure 5 the percentage of all branches executed that belong the sequences of lengths two, three and lengths greater than three are shown. Of the 60.6% of total dynamically executed branches in `Spec95 integer` benchmarks that belong to branch sequences, 29.5% of the total branches belong to sequences of length two, 16.6% of the total branches belong to sequences of length three, and 14.5% of the total branches belong to sequences of lengths greater than three. Thus in practice, in addition to tracking the history of individual branches, it may be sufficient that the branch sequence prediction hardware detect and track execution history for branch sequences of lengths two and three.

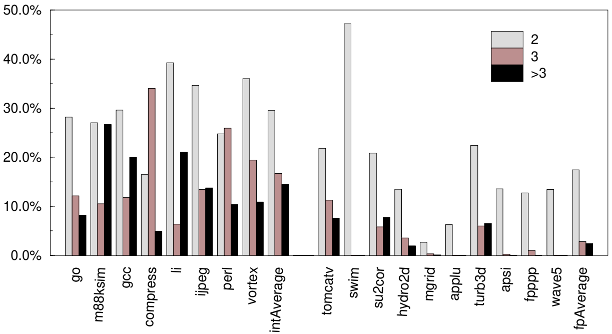


Figure 5. Percentage of branches in sequences of length 2, 3 and >3.

In order to confirm that limiting the sequences lengths to a maximum of three will not lead to significant performance degradation, we ran another experiment in which we computed the percentage of executed branches in sequences of lengths two and three. The results show that there is an average of less than 2% drop in the number of branches covered by the sequences. The reason for such a small drop is that once a limit of length three has been placed on the sequences, the branch sequences of lengths greater than three will automatically be broken down into subsequences of shorter lengths. **In conclusion we will design an implementation of branch sequence predictor that will only handle sequences of lengths up to three.**

Number of sequences. The next characteristic that we consider to be important is the number of sequences encountered since maintaining their histories explicitly would require providing additional branch history tables (BHTs). Limiting the sequence lengths to three branches, our basic strategy is to organize the *branch sequence history* into three BHTs. All individual branch histories will be stored in the first BHT as is typically done by a traditional branch predictor such as gshare [6]. The two additional BHTs will strictly store histories corresponding to branch sequences. The second BHT will contain entries for the histories of branches that are encountered as being the second branch in a sequence while the third BHT will be used to store histories of branches that appear as the third branch in a sequence.

Given the above three level organization, the sizes of the additional second and third level BHTs depend upon number of branch sequences that are encountered. In particular, the second level BHT would have an entry for each sequence of length two or three while the third level BHT will simply have entries for sequences of length three. In order to determine suitable relative sizes of the three tables we conducted an experiment which identified the number of unique branches, number of unique sequences of lengths two or three, and number of sequences of length three for the Spec95 benchmarks. In Figure 6 the sizes of second and third level BHTs relative to the first level BHT are plotted. The largest table sizes were observed for the gcc benchmark since it contains a very large number of branch sequences. The size of second level BHT was observed to be 0.41 of the first level and the size of the third level BHT was found to be 0.17 of the first level. While it may appear that the additional cost of these BHTs is significant, we view them as being constructed using part of the budget for the on-chip instruction cache. This is because these additional tables will be used to store instructions in branch sequences, in addition to branch sequence history, and thus will relieve the pressure that the branch sequences impose on the instruction cache. **In conclusion we will consider sizes for second and third level BHTs that are one-half and one-fourth the size of the first level BHT.**

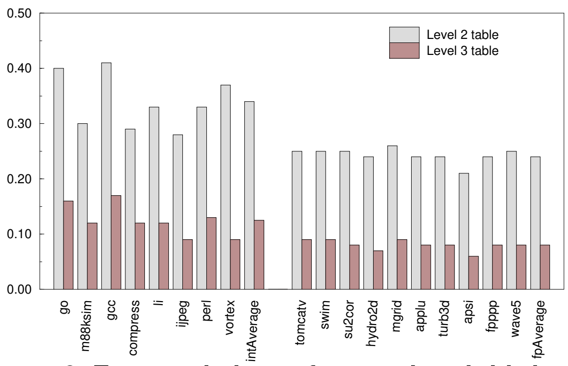


Figure 6. Expected sizes of second and third level tables relative to the first level table.

Profile of branches in branch sequences. Finally we analyzed the profile of branch instructions in terms of their types. The branches that were encountered during execution were classified into the following categories: UI - unconditional indirect; UD - unconditional direct; CD - conditional direct; and others - all other categories. The reason for using the above classification is that the first three categories account for nearly all the branch instructions executed in Spec95 benchmarks. The results of above analysis are shown in Figure 7. While conditional direct branches account for a large fraction of branches that are executed, there are a significant number of unconditional, direct as well as indirect, branches in branch sequences. Since the unconditional direct branches in sequences will be saved as part of the branch sequence history, and the targets of these branches can be typically correctly predicted using a branch target buffer, the harmful effects of these branch instructions on instruction fetching will be almost entirely eliminated through sequence prediction. In addition, their high predictability will also assist in achieving a high prediction rate for branch sequences. Unconditional indirect branches are encountered as part of sequences quite frequently. They account for an average of approximately 11% of those branches that are executed as part of branch sequences in Spec95 benchmarks. Often indirect branches are generated during the translation of switch statements which also happen to be a frequent source of branch sequences. Therefore to achieve high branch sequence prediction rates we must accurately predict targets of unconditional indirect branches. **In conclusion we**

will provide a target cache [1] type mechanism in the design of our branch sequence predictor to accurately predict the targets of unconditional indirect branches.

Program	UI%	UD%	CD%	others%
go	9.35	10.10	76.52	4.03
m88ksim	14.62	6.03	67.99	11.36
gcc	6.65	14.45	73.93	4.97
compress	15.90	12.30	51.70	20.11
li	13.06	8.68	70.51	7.75
ijpeg	6.32	8.01	80.02	5.65
perl	9.69	27.02	58.33	4.96
vortex	16.04	7.58	76.26	0.12
intAverage	11.45	11.77	69.41	7.37
tomcatv	7.88	39.61	51.65	0.86
swim	24.99	25.01	50.00	0
su2cor	9.08	36.79	53.78	0.35
hydro2d	10.54	33.73	54.88	0.85
mgrid	1.46	11.71	86.54	0.29
applu	0.01	0.06	99.93	0
turb3d	8.87	26.38	64.49	0.26
apsi	19.30	23.26	56.84	0.60
fpppp	15.10	23.65	60.90	0.35
wave5	21.78	26.13	51.58	0.51
fpAverage	11.90	24.63	63.06	0.41

Figure 7. Distribution of branch instruction types in branch sequences.

3 Implementation of a Branch Sequence Predictor

In order to exploit branch sequences we have developed a mechanism which is able to dynamically detect sequences, maintain the history of the detected sequences, and predict sequences. Whenever a branch instruction is fetched, the branch sequence history is consulted for prediction. The result of prediction is either simply the prediction of the branch, as in traditional single branch predictors, or if the branch is the first branch of a predictable branch sequence, then the entire sequence is predicted. In the latter case the remaining instructions belonging to the sequence are also provided by the branch sequence predictor from its sequence cache and these instructions are speculatively introduced in to the execution pipeline. The fetch mechanism resumes fetching of instructions from the branch sequence target, that is, the target of the last branch in the sequence. A simple state machine is implemented to detect branch sequences. This state machine examines the retiring instructions to detect branch sequences and then updates the branch sequence history appropriately.

Branch Sequence Predictor. The branch sequence prediction hardware is shown in greater detail in Figure 8. The branch history is organized in multi level table. Each level corresponds to the position of the branch in a sequence. When a branch instruction B is fetched, it is predicted using the information in the first level BHT. The second and third level BHTs provide the information corresponding to the second and third branches in branch sequence beginning with B assuming that there is such a sequence. We only provide three levels because the results in the preceding section indicated that this is enough to capture most of the branch sequences. Also the relative sizes of tables are chosen in accordance with the observations presented in the preceding section, that is, second and third level BHTs are one-half and one-quarter the size of the first level BHT.

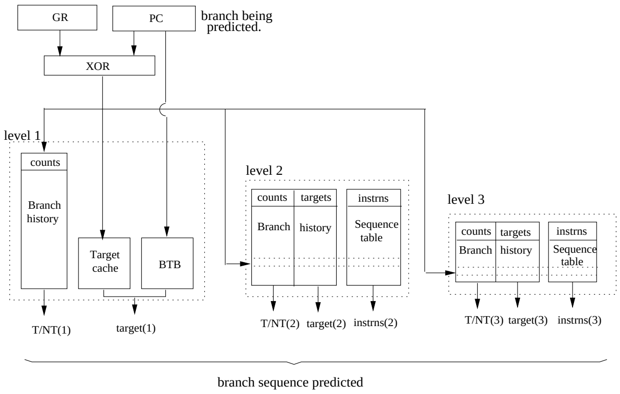


Figure 8. Branch sequence predictor.

This implementation we propose is based upon the generalization of the approach used by the gshare single branch predictor. The first level BHT is similar to the BHT used in the gshare implementation. It is indexed using exclusive-or of nine bits of global branch history kept in register GR and the PC of the branch B being predicted. The first level branch history BHTs is also provided with an accompanying target cache and branch target buffer (BTBs). The former is used for predicting the targets of indirect branches while the latter predicts the targets of direct branches. It should be noted that while BTB is accessed using an index derived from the branch's PC, the target cache is indexed using derived from exclusive-or of GR and the branch's PC. This is in accordance with their typical implementations. The contents of the first level BHT are similar to the traditional gshare single branch predictor, that is, two bit saturating counters are supported by the first level `BHT`. The value of the counter determines whether to predict the branch as taken or not-taken (values 2 and 3 correspond to the taken prediction and the values 0 and 1 correspond to the not-taken prediction).

The second and third levels also provide their own `BHT`s. The index used to access the first level table, with its most significant bit (two bits) dropped, is used to access the second (third) level `BHT`. In other words all three tables are accessed using indices directly derived from the branch `B`, the first branch in the sequence being predicted. This approach has the advantage that all three tables can be accessed in parallel and therefore branch sequence prediction takes no more time than single branch prediction. For target prediction at the second and third level, we do not use separate `BTB` or target cache structures. Instead an extended `BHT` entry which saves the taken target of the branch works well in practice. The second and third level `BHT`s also contain two bit saturating counters to predict their respective branches. These counters capture the history of a branch when it appears in a branch sequence as the second or third branch in the sequence.

The instructions in the sequence that follow the first branch instruction are stored in the *sequence tables* provided at the second and third levels. A sequence table is indexed using the same index as the corresponding `BHT` at the same level. Therefore we assume that the number of lines in the sequence cache is the same as the number of entries in the corresponding `BHT`. The second level sequence table provides instructions following the first (predicted) branch `B` and up to the second branch in the sequence (marked as `instrns(2)` in Figure 8). The third level sequence table provides instructions following the second branch and up to the third branch in the sequence (marked as `instrns(3)` in Figure 8). In addition to the instructions, each sequence table line also contains the `PC` of the first saved instruction. Therefore the line size for a sequence table at each level is three. The saved `PC` values are matched against the predicted targets of preceding branches (i.e., `target(1)` and `target(2)` in Figure 8) to ensure that a valid sequence is being predicted.

In order to ensure that sequence prediction is highly accurate, we allow the prediction of a sequence of length two only if the confidence in the prediction of the first branch is high. Thus prediction is only allowed for the extreme values (i.e., values 0 and 3) for the counter at the first level. Sequences of length two are not predicted if the counter at the first level has a value of either 1 or 2. The value of the counter at the second level is similarly used to decide whether to extend the prediction to a sequence of length three. In addition, the branch at the third level is only included if the confidence in its prediction is high (i.e., its counter value is 0 or 3). Thus, the counters at the first two levels not only provide predictions for the branches at their respective levels, but they also serve as a confidence mechanism for deciding whether or not a sequence should be predicted. When a frequently occurring predictable sequence is encountered we predict the outcome of the entire sequence. On the other hand if no sequence is predicted the predictor naturally falls back to predicting the branches individually.

Branch Sequence Detection and History Update. In order to maintain the branch sequence histories, we take the following approach. A state machine that examines retiring instructions and detects branch sequences is provided (see Figure 9). Depending upon the nature of the recently retiring instruction, a branch instruction or a non-branch instruction, appropriate state transitions occur. The initial state of this state machine is state 0 and after one branch instruction is seen it moves to state 1. When a sequence of two branches has been seen the machine is in state 2 and finally when a sequence of three branches has been detected the machine is in state 3.

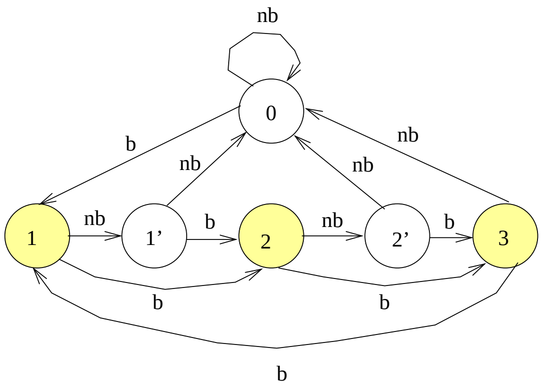


Figure 9. Sequence detection and history update.

Our history update strategy is as follows. The first level `BHT` contains histories of all branches. Therefore it is updated for the most recently seen branch in states 1, 2 and 3. The level two `BHT` is updated in state 2 and level three `BHT` is updated in state 3. It should be noted that when the history of the recently seen branch is being updated in state 2(3), the index at which the update is performed in level two(three) `BHT` must be computed from the `GR` and `PC` of the first branch in the sequence. On the other hand when the history of the same branch is being updated in the level one

BHT the index must be computed by appropriately constructing the contents of GR for that branch. A new sequence is not allowed to evict an existing sequence if the latter is highly predictable and frequent (i.e., all of its counters are saturated to the values of 0 or 3). Instead the counter value of 0 is incremented and 3 is decremented so that in the future the replacement may become possible. Using this technique we prevent sporadically appearing sequences from evicting frequently occurring ones. When a sequence is predicted by our sequence predictor, the updates to the history are performed speculatively since doing so provides better performance. This trend is not surprising as it has been observed for other branch predictors [3].

4 Performance Evaluation

In order to evaluate the performance of branch sequence prediction we conducted experiments with the following goals. First we demonstrate that the practical implementation of branch sequence prediction, proposed in the preceding section, provides low overall branch misprediction rates. Next we evaluate the effectiveness of the fetch mechanism based upon sequence prediction by measuring the average number of useful instructions fetched from the I-cache per cycle. We also estimate the reduction in cache lines fetched from the cache. Finally we study the impact of the improved fetch mechanism on overall execution times.

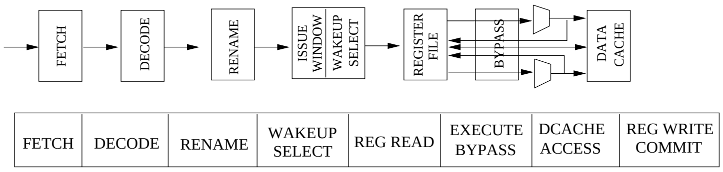
As in the case of all earlier experiments, Spec95 benchmark suite is used in the evaluation The benchmarks were compiled using the gcc 2.7.2 compiler with -O3, the highest, level of optimization. All benchmarks were executed on the test inputs. The execution of long running benchmarks was terminated once they had executed 40 million branch instructions. Relative performances of architectures that are identical in all respects except their branch predictors are presented. The simulators used in this study have been automatically generated from ADL specifications of superscalar architectures by the FAST system [7]. Our simulated architecture is based upon the MIPS-I instruction set and machine model shown in Figure 10.

We considered two different issue-widths in our experiments (4 and 8). Further it is assumed that the I-cache line size is equal to the issue width and in each cycle a single cache line can be fetched. In comparing the performance of the gshare and bsp predictors, we considered configurations in which the number of BHT entries in the gshare predictor is equal to the sum of the number of BHT entries in all three tables of the bsp predictor. In the first configuration the gshare has 4K entries while the bsp predictor has 2K, 1K, and 1K entries in the first, second and third level BHTs respectively. In the second configuration the gshare has 8K entries while the bsp predictor has 4K, 2K, and 2K entries in the first, second and third level BHTs respectively. According to our earlier observation we could have used smaller third level tables. However, we used larger third level tables so that the gshare and bsp predictors have the same total number of BHT entries.

Branch Prediction Rates. In comparing the prediction rates of the gshare and bsp predictors, we consider a branch to be correctly predicted if not only is its outcome correctly predicted, but so is its target. As we can see from the tables in Figures 11 and 12, the branch prediction rates of the two predictors are comparable. In some cases bsp performs marginally better while in others the gshare performs slightly better.

Fetch Effectiveness. In order to compare the fetch effectiveness of gshare and bsp two criteria are used. We measured the number of cache lines fetched by the bsp predictor and the gshare predictor to determine the reduction in cache traffic resulting from using bsp. The results in Figures 11 and 12 show that bsp reduces the amount of cache lines fetched by 9% on an average for integer benchmarks and a maximum of around 15% for perl and wave5 benchmarks. We also compared the average number of useful instructions contained in cache lines fetched by the bsp predictor with that of gshare. The results in Figures 11 and 12 show that bsp increases the average number of useful instructions per fetched cache line by 9% on an average for integer benchmarks and a maximum of 17% for perl. In the hydro2d and wave5 floating point benchmarks, over 10% improvement was also observed for 8K BHT sizes. The only benchmarks which showed a slight performance loss were m88ksim and tomcatv.

Execution Times. The overall improvement in instructions executed per cycle (IPCs) that results from using bsp instead of gshare was also found. As indicated by the tables in Figures 11 and 12, the bsp predictor increases the IPCs by over 10% for gcc, perl and ijpeg. Smaller gains were observed for other integer benchmarks. For 8K BHT sizes the floating point benchmarks gave an average improvement of 8% and a maximum of 17% for wave5. Only tomcatv showed a slight performance loss. We assumed a perfect I-cache in this experiment. Therefore the improvements we report are entirely because of I-cache fetches that are avoided. If a non-perfect I-cache is simulated it will only favor bsp because bsp fetches fewer I-cache lines and is therefore likely to suffer less from I-cache misses.



`BTB` - is four-way associative with 512 entries.
Target cache - untagged with 256 entries.
Recovery from branch misprediction - future file [5, 11].
Load speculation - based upon store set mechanism [2].
Latency - 2 cycles for Loads and unit latencies for others.

Figure 10. Superscalar machine model.

Program	Prediction rates		Lines fetched		Useful instrns per line		IPCs	
	$\frac{bsp2+1+1K}{gshare4K}$	$\frac{bsp4+2+2K}{gshare8K}$	$\frac{bsp2+1+1K}{gshare4K}$	$\frac{bsp4+2+2K}{gshare8K}$	$\frac{bsp2+1+1K}{gshare4K}$	$\frac{bsp4+2+2K}{gshare8K}$	$\frac{bsp2+1+1K}{gshare4K}$	$\frac{bsp4+2+2K}{gshare8K}$
`go`	1.00563	1.00613	0.970854	0.967406	1.03003	1.03369	1.02648	1.02957
`m88ksim`	0.985577	0.985739	0.971719	0.971792	1.0291	1.02902	1.01497	1.01356
`gcc`	1.00773	1.00801	0.907015	0.895872	1.10252	1.11623	1.08975	1.1026
`compress`	0.986148	0.985699	0.933367	0.934926	1.07139	1.0696	1.0443	1.04226
`li`	0.990177	0.989605	0.896972	0.894299	1.11486	1.11819	1.0339	1.03283
`ijpeg`	1.00145	1.00526	0.910929	0.889783	1.09778	1.12387	1.08886	1.11583
`perl`	1.00198	0.998921	0.894696	0.868787	1.1177	1.15103	1.10721	1.1357
`vortex`	0.998945	1.0125	0.876862	0.897337	1.14043	1.11497	1.07005	1.10145
`intAverage`	0.997205	0.998982	0.920301	0.915026	1.08798	1.09458	1.05944	1.07173

Figure 11. Relative performance of bsp vs gshare for: 4-issue and line size of 4.

Program	Prediction rates		Lines fetched		Useful instrns per line		IPCs	
	$\frac{bsp2+1+1K}{gshare4K}$	$\frac{bsp4+2+2K}{gshare8K}$	$\frac{bsp2+1+1K}{gshare4K}$	$\frac{bsp4+2+2K}{gshare8K}$	$\frac{bsp2+1+1K}{gshare4K}$	$\frac{bsp4+2+2K}{gshare8K}$	$\frac{bsp2+1+1K}{gshare4K}$	$\frac{bsp4+2+2K}{gshare8K}$
`go`	1.0052	1.0067	0.967656	0.962886	1.03342	1.03855	1.02647	1.03171
`m88ksim`	0.98607	0.98652	1.01337	1.01208	0.986803	0.988064	0.971849	0.974229
`gcc`	1.00933	1.00743	0.903822	0.901147	1.10641	1.1097	1.08902	1.09008
`compress`	0.996503	0.996489	0.912144	0.912165	1.09632	1.09629	1.07508	1.07504
`li`	0.992928	0.992836	0.907336	0.906954	1.10213	1.10259	1.0219	1.0222
`ijpeg`	1.00418	1.00532	0.907873	0.90431	1.10148	1.10581	1.09396	1.10186
`perl`	1.00365	1.00192	0.849906	0.853124	1.1766	1.17216	1.15674	1.15189
`vortex`	1.02096	1.01337	0.879692	0.890796	1.13676	1.12259	1.12372	1.10294
`intAverage`	1.00235	1.00132	0.917725	0.917933	1.09249	1.09197	1.06984	1.06874
`tomcatv`	0.986147	0.994232	1.00024	0.994007	0.999761	1.00543	0.977915	0.993601
`swim`	0.988678	0.999679	0.94892	0.920363	1.05383	1.08653	1.0417	1.08351
`su2cor`	0.987627	0.994995	0.963702	0.923489	1.03766	1.08228	1.00408	1.08118
`hydro2d`	0.99707	0.998392	0.988936	0.882861	1.01119	1.13295	1.00503	1.12347
`applu`	1.00286	1.00022	0.990729	0.972273	1.00935	1.02851	1.01011	1.05106
`turb3d`	1.00065	1.00662	0.968783	0.931552	1.03222	1.07348	1.02531	1.0736
`apsi`	0.998202	0.999121	0.991406	0.943565	1.00867	1.05977	1.00567	1.05821
`wave5`	0.999388	1.01287	0.979844	0.845578	1.02056	1.18262	1.01531	1.17209
`fpAverage`	0.995079	1.00076	0.97907	0.92671	1.02165	1.08145	1.01064	1.07959

Figure 12. Relative performance of bsp vs gshare for: 8-issue and line size of 8.

5 Related Work

More recently two aggressive fetch mechanisms have been proposed based upon multiple branch prediction: the *branch address cache* [13] and the *trace cache* [9].

The mechanism based upon branch address cache [13] generates targets of multiple subsequent basic blocks in each cycle and fetches them simultaneously from the I-cache. To generate multiple targets, outcomes of multiple branches must be predicted in each cycle. The complexity of the hardware for predicting the targets increases exponentially with each branch. Moreover the above mechanism requires for the instruction cache to be capable of providing multiple nonconsecutive lines in each cycle. In contrast our mechanism based upon branch sequence prediction only fetches a single cache line from the I-cache in each cycle and its hardware complexity grows linearly as additional branches are added to the sequences.

In the trace cache based fetch mechanism traces are formed from consecutively executed dynamic instruction sequences when they are encountered first, saved in the trace cache, and fetched from the trace cache during future executions. Traces are formed for all executed instructions and instructions lying along multiple paths may belong to multiple traces. here are several important differences between our fetch mechanism and one based upon the trace cache. In the trace cache, traces are constructed first and then the outcomes of the multiple branches that fall in the trace are predicted. In contrast our approach identifies highly predictable branch sequences first and then only predicts the outcomes of multiple branches that correspond to these branch sequences.

The branch sequences stored in the sequence table may be viewed as very specialized traces which are short (between 2 and 5 instruction in length) and have high branch density (2 or 3 branches). Thus, among all the traces that may be constructed for a program, branch sequences represent those that are most detrimental to fetch performance.

Finally, our fetch mechanism can be very naturally extended to build a highly aggressive fetch mechanism like the trace cache if more storage is made available for larger sized sequence tables. As mentioned above, sequence table is currently used to store very short traces with very high branch density. By progressively allowing increasing number of non-branch instructions between consecutive branches we can allow branch sequences (i.e., traces) of increasing size and lower branch densities to be saved. Such flexibility of selectively saving important traces is not supported by proposed trace cache designs since they treat all traces equally.

References

[1] P-Y. Chang, E. Hao and Y.N. Patt, "Target Prediction for Indirect Jumps," *Proc. of the 24th Annual International Symposium on Computer Architecture*, pages 274-283, June 1997.

[2] G. Chrysos and J. Emer, "Memory Dependence Prediction using Store Set," *Proc. of the 25th Annual International Symposium on Computer Architecture*, pages 142-154, June 1998.

[3] E. Hao, P-Y. Chang, and Y. Patt, "The Effect of Speculatively Updating Branch History on Branch Prediction Accuracy, Revisited," *Proceedings of the 27th International Symposium on Microarchitecture*, San Jose, California, November, 1994.

[4] Q. Jacobson, E. Rotenberg, and J. E. Smith, "Path-Based Next Trace Prediction," *Proc. of the 30th Annual International Symposium on Microarchitecture*, pages 14-23, December 1997.

[5] M. Johnson, *Superscalar Microprocessor Design*, Prentice Hall Series in Innovative Technology, 1990.

[6] S. McFarling, "Combining Branch Predictors," *WRL Technical Note TN-36*, Digital, Palo Alto, CA, June 1993.

[7] S. Onder and R. Gupta, "Automatic Generation of Microarchitecture Simulators," *IEEE International Conference on Computer Languages*, pages 80-89, Chicago, Illinois, May 1998.

[8] S.T. Pan, K. So, ad J.T. Rahmeh, "Improving the Accuracy of Dynamic Branch Prediction using Branch Correlation," *Proc. of ASPLOS V*, pages 76-84, October 1992.

[9] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace cache: A Low Latency Approach to High Bandwidth Instruction Fetching," *Proc. of the 29th Annual International Symposium on Microarchitecture*, December 1996.

[10] J.E. Smith, "A Study of Branch Prediction Strategies," *Proc. of 8th International Symposium on Computer Architecture*, pages 135-148, May 1981.

[11] G.S. Sohi and S. Vajapeyam, "Instruction Issue Logic for High-Performance Interruptable Pipelined Processors," *Proc. of the 14th Annual Symposium on Computer Architecture*, pages 27-34, June 1987.

[12] T-Y. Yeh and Y.N. Patt, "A Comparison of Dynamic Branch Predictors that use two levels of branch history," *Proc. of the 20th International Symposium on Computer Architecture*, pages 257-266, May 1993.

[13] T-Y. Yeh, D. Marr, and Y.N. Patt, "Increasing the Instruction Fetch Rate via Multiple Branch Prediction and Branch Address Cache," *Proc. of the 7th ACM International Conference on Supercomputing*, July 1993.

[14] T-Y. Yeh and Y.N. Patt, "Two-level Adaptive Branch Prediction," *Proc. of the 24th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 51-61, 1991.