Proceedings of IEEE First International Conference on Algorithms and Architectures for Parallel Processing (ICA^3PP), Brisbane, Australia, April 19-21, 1995

Block-Level Prediction for Wide-Issue Superscalar Processors

Simonjit Dutta and **Manoj Franklin**
Department of Electrical and Computer Engineering
Clemson University
102 Riggs Hall
Clemson, SC 29634-0915, USA
Email: simonjd@eng.clemson.edu, mfrankl@eng.clemson.edu

Abstract

Changes in control flow, caused primarily by conditional branches, are a prime impediment to the performance of wide-issue superscalar processors. This paper investigates a block-level prediction scheme to mitigate the effects of control flow changes caused by conditional branches. Instead of predicting the outcome of each conditional branch individually, this scheme predicts the target of a sequential block of instructions, thereby allowing the superscalar processor to go past multiple branches per cycle. This approach is evaluated using the MIPS architecture, for 8-way and 12-way superscalar processors, and an improvement in effective fetch size of approximately 15% and 25%, respectively, over identical processors that use branch prediction is observed. No appreciable difference in the prediction accuracy was observed, although block-level prediction predicted one out of four outcomes.

1 Introduction

Superscalar processors are becoming very popular now. Almost all of the recent microprocessor releases have applied the superscalar technique in some way or other [1] [12]. As more and more transistors can be placed in a chip, the issue width of these processors also keeps increasing steadily. To sustain a high issue rate, a superscalar processor fetches and decodes multiple instructions per cycle from a single flow of control. Needless to say, the sustained instruction issue rate of a superscalar processor will never exceed the average number of (useful) instructions it fetches per cycle.

0-7803-2018-2/95/$4.00 © 1995 IEEE

Fetching multiple instructions in a cycle is not difficult so long as the fetched instructions are from a straight line piece of code. Conditional branches are the primary impediment to fetching a large number of instructions per cycle, because they are frequent (roughly one in every 5 instructions, on the average), and can potentially alter the flow of control. Researchers realized this problem fairly early on, and the solution proposed was to predict the outcome of the branch before it was resolved, and to fetch and execute instructions in a speculative manner from the predicted target. If the prediction is later found to be incorrect, the effects of the instructions executed after the branch are nullified, and execution continues along the correct path. Branch prediction techniques have now improved to the point of getting about 90%-96% prediction accuracies for non-numeric programs [9] [14].

One problem that cannot be solved by high-accuracy predictions alone is that a branch's identity and target are not known until the branch instruction is decoded. If the prediction is fall-through, then this is not a problem; if the branch is predicted to be taken, then this causes a 1-cycle bubble in the fetch pipeline. Furthermore, if at most a single branch prediction is made per cycle, the instruction fetch mechanism can fetch no more than 5 instructions per fetch attempt on the average. Both these factors severely limit the the instruction level parallelism that can be exploited by superscalar processors. It is clear that to overcome the above bottleneck, either (i) the number of branches encountered by the hardware has to be reduced (through techniques such as *guarding* [6] [11]) or (ii) the target information has to be associated with higher-level blocks (as opposed to associating them with the branches), and the outcomes of multiple branches need to be predicted in a cycle.

This paper focuses on the second approach.

The approach considered in [10], called *control flow prediction*, is to predict the target of a subgraph of the control flow graph of the executed program. By predicting one of the targets, the control flow predictor specifies the next subgraph to be (speculatively) executed, and thereby goes past one subgraph a cycle. In the type of control flow prediction considered in [10], a subgraph can have an arbitrary number of branches encompassed within, and can therefore have arbitrary structure, length, and number of targets. Therefore, the instructions in a subgraph are most likely fetched sequentially. Such a control flow prediction scheme, with unrestricted subgraph structures, is more apt for execution models that pursue multiple flows of control, such as the *multiscalar processor* [2], and is not the subject of this paper.

For processors that follow the superscalar model of execution, we need to restrict the subgraph structure, and identify a path within the subgraph to fetch instructions from. The approach considered in [13] is to use tree-like subgraphs, and select a path through the tree-like subgraph by performing 2-3 branch predictions in a cycle, using a high accuracy, 2-level branch predictor [14]. For a given subgraph address, the prediction mechanism performs these predictions before even determining the addresses of the intermediate conditional branches encompassed within the subgraph. Based on the prediction values, a *Branch Address Cache* provides the intermediate fetch addresses. The instruction fetch mechanism then fetches instructions from these multiple targets. Because the intermediate branch addresses are not known when the predictions are made, this scheme uses the condensed history of all branches to make its decisions.

This paper considers a sequential block of instructions (with multiple branches encompassed within) as a subgraph. Furthermore, instead of predicting the outcome of each conditional branch in the block, a single prediction is made about which of the possible targets will be taken, and instruction fetch is continued from that target in the next cycle. The advantage of this scheme is that the predictor and the instruction fetch mechanism need not be any more complex than those in a superscalar processor implementing branch prediction.

This paper is organized as 5 sections. The introduction has highlighted the need to go beyond predictions at the branch level and to predict the outcome of multiple branches per cycle. Section 2 details the operation of block-level prediction, and discusses techniques for carrying out this scheme. Section 3 calculates the hardware cost of the block-level prediction scheme. Section 4 presents an experimental evaluation of block-level prediction using the MIPS architecture. Section 5 presents a summary and the conclusions.

2 Block-Level Prediction

2.1 General Operation

The central idea of the new predictor is to make control flow predictions at the block level, where a block is a sequential block of instructions. That is, instead of predicting the outcome of each branch (to decide the speculative execution path), predict the target of a sequential block of instructions.

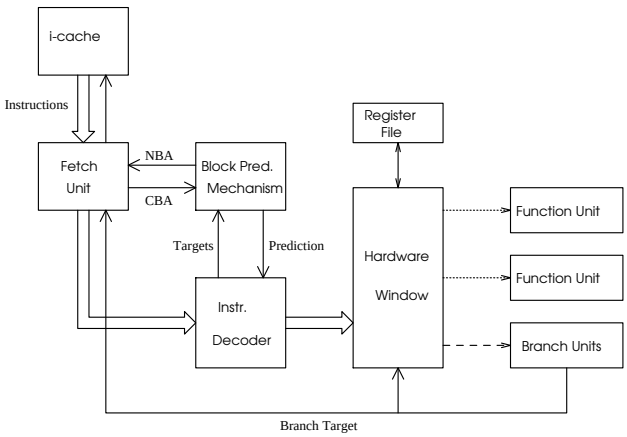

Figure 1: Block Diagram of a Processor with Block-Level Prediction

Figure 1 gives a block diagram showing the relationship between the instruction cache, the block predictor, the instruction fetch unit, the decode unit, the hardware window, and the functional units in a superscalar processor. The fetch unit has the address of the next block to be fetched, and sends this address to the block predictor. The block predictor makes a prediction to select an outcome and the corresponding target, and sends the outcome and the target to the fetch unit. The fetch unit fetches this block from the multi-ported i-cache, and forwards the fetched instructions to the decoder. If the fetch unit encounters a particular block for the first time, it also tells the decoder to send that block's targets to the prediction mechanism for storage.

The decoder decodes the instructions fetched in the previous cycle by the fetch unit, taking into con-

sideration the prediction made earlier (by the predictor) on the outcome of that block. It discards any instructions that succeed a conditional branch that is indirectly predicted to be taken. The decoded instructions are sent to the hardware window structure, which serves as a platform for performing dynamic scheduling among the decoded instructions. Multiple functional units are provided to execute multiple instructions per cycle. Among these, the branch units carry out the execution of control-changing instructions. When a control-changing instruction is executed, a check is made to see if the actual outcome is in line with the earlier block prediction; if there is a discrepancy, the hardware window discards the instructions following the point of discrepancy. It also sends the misprediction information to the fetch unit, which then starts fetching from the correct target.

2.2 How Many Targets Per Block?

Before proceeding to study how exactly the block predictions are made, we shall pause briefly to examine some restrictions that we need to impose on the size of our blocks. We can consider 2 attributes when discussing the size of a sequential block: (i) the number of control flow outcomes a block can have, and (ii) the number of instructions in a block.

The first attribute is decided by the number of branches encompassed in a block. If a block has only 2 outcomes, then only one branch is contained in a block, and only one branch is effectively predicted per cycle. If a single branch is effectively predicted per cycle, the average number of useful instructions fetched per cycle can never exceed the average number of instructions between consecutive branches, which, for non-numeric codes, is only about 5. On the other hand, if a block has a large number of possible outcomes, then the prediction mechanism has to choose one outcome from many, resulting in potentially lower prediction accuracies. Each time a misprediction is detected, the subsequent instructions (the ones after the mispredicted branch) in the hardware window have to be discarded. Thus, although a larger block is fetched at a time, the poor prediction accuracy could potentially reduce the performance compared to the case when a block has fewer outcomes. As a trade off, for our studies in this paper, we allow up to 4 outcomes per block. This will allow up to 3 conditional branches to be enclosed within a block[1]. The outcomes of a block are encoded as follows: The conditional branches in a block are assigned numbers {1, 2, 3}, depending on their sequential order of appearance in the block. The outcome obtained when conditional branch i is taken is encoded as *outcome* i. The outcome obtained if all conditional branches follow the fall-through path is encoded as *outcome* 0. Associated with each possible outcome, there is a single target. Multiple outcomes can have the same target, however. Figure 2 illustrates the encoding scheme used for the outcomes of a block. In the figure, CBi represents the i^{th} conditional branch in a block, and the numbers {0, 1, 2, 3} denote the possible outcomes of a block.

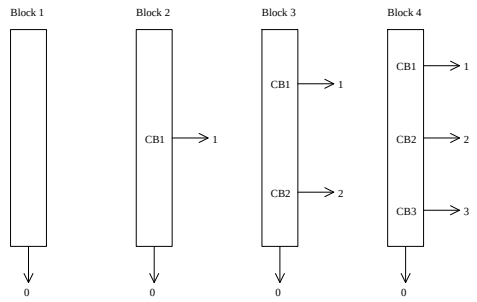

Figure 2: Encoding of Block Outcomes for Blocks with 1, 2, 3, and 4 Outcomes

Now consider the second attribute, namely the number of instructions in a block. Ideally, the fetch mechanism should be able to fetch all instructions from a block in one cycle. This will allow one block prediction to be carried out every cycle. Thus, the length of a block cannot exceed f, where f is the maximum fetch size of the superscalar fetch mechanism.

2.3 Prediction Mechanism

Let us now return to our discussion of the means for carrying out block predictions. The block predictions are made with the help of a 2-*level block predictor*[2], similar in spirit to the 2-level branch predictor described in [14].

Figure 3 shows the block diagram of the various parts within our 2-level block predictor. It consists of two main structures—the *Block History Table (BHT)* and the *Pattern History Table (PHT)*. The BHT is

[1] Restricting the number of conditional branches in a block to 3 seems to be a good choice, as indicated by the results given later in Table 3.

[2] It is not mandatory to use a 2-level prediction scheme for implementing block prediction. We describe an implementation using the 2-level approach, as it tends to give high prediction accuracies.

a cache-like structure that stores information relevant to the blocks that have been encountered in the recent past. Each BHT entry has 3 fields—Tag, Targets, and Block History Pattern. The Tag field is for uniquely identifying the block currently mapped to that entry. The Targets field stores the targets of the block, and the Block History Pattern field stores the last p outcomes ($p = 6$ in the figure) of the block as a $2p$-bit pattern. Because there are 4 possible outcomes for a block, 2 bits are required to store each outcome. The secondary level of the predictor is the PHT. For each possible $2p$-bit pattern, a condensed history of the previous outcomes corresponding to the pattern is recorded in the PHT by means of 4 independent up/down counter values $\{C_0, C_1, C_2, C_3\}$.

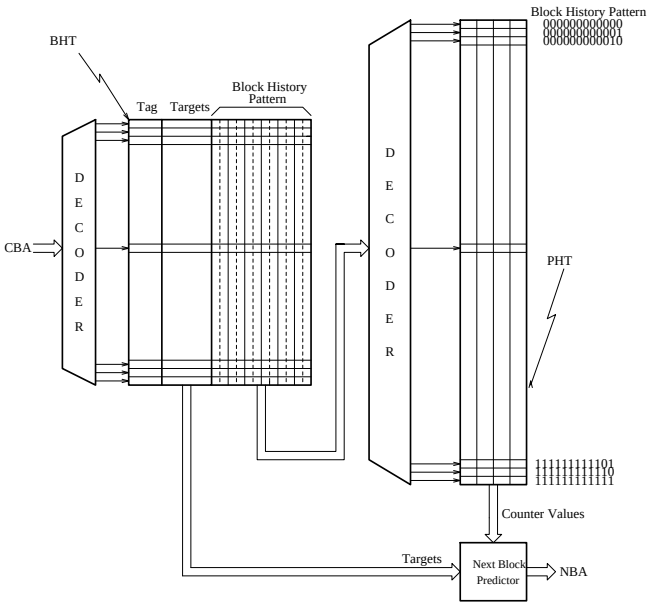

Figure 3: Block Diagram of 2-Level Block Predictor

When the instruction fetch unit makes a prediction request for a particular block address, the appropriate entry in the BHT is selected. The tag value of the entry is checked to determine if the entry corresponds to the current block address. If so, the entry's block history pattern is used to select one of the entries from the PHT. The PHT entry contains 4 count values, and the prediction mechanism determines the maximum count value. The outcome corresponding to the maximum count value is selected as the next prediction value. (If there is a tie between two or more count values for the maximum position, the selector could choose the last outcome as the next prediction, if the last outcome is also stored in each PHT entry.) The BHT entry also contains the targets corresponding to the current block. Depending on the prediction value selected, a target is chosen from the targets available in the selected BHT entry, and sent to the instruction fetch unit as the next block address.

When a prediction is made, in addition to sending the next target of a block, the predictor also updates the history of that block. The PHT entry's counter values are updated in the following manner. The count value corresponding to the predicted outcome is incremented by 3 (or less if incrementing leads to the counter value reaching its maximum allowed value), and all other count values are decremented by 1 (if their count values are not already 0). Finally, the block history pattern corresponding to that block address is shifted left by 2 bit positions, and the latest outcome is entered in the 2 rightmost bit positions (*i.e.*, the bit positions left vacant by the left shift). Notice that these updates to the BHT and PHT entries, done immediately after making a prediction, are done in a speculative manner. By doing so, up-to-date history is available for making future predictions. (As the prediction accuracy of 2-level prediction schemes is very high, using the speculatively updated history would result in better prediction accuracy than using obsolete history to make the predictions.) If a prediction is later found to be incorrect, the speculative updates done to the BHT and PHT entries corresponding to the blocks discarded by the recovery mechanism are setback.

2.3.1 Handling New Blocks

When a block B is encountered by the fetch unit for the first time, or if its entry in the BHT was replaced by another block, the BHT will not have an entry for B. If the fetch unit requests a prediction for B in such a situation, the predictor selects a prediction value of 0, and returns a target value equal to B's address $+ f$, where f is the maximum fetch size, to the fetch unit. When this block is fetched, the fetch unit requests the decoder to send the correct targets to the BHT for storage.

2.4 Determination of New Target at Times of Recovery

When a prediction is made for a block, the outcomes of the conditional branches contained in the block also get indirectly predicted. When the predicted outcome of a conditional branch is later found to be incorrect, recovery actions are initiated. These actions include clearing from the instruction window all instruc-

tions that have been fetched after the mispredicted branch. The recovery process also includes finding a new block prediction for the block that contains the mispredicted branch. If the branch was predicted to be fall-through and ends up being taken, then finding the new block prediction is straightforward; the new block prediction is given by the *branch number* b of the mispredicted branch within its block. This case is illustrated in Figure 4, for $b = 2$. The first figure shows a block whose outcome was originally predicted to be 3, as shown by the thick line. The corresponding predictions for the conditional branches CB1, CB2, and CB3 are fall-through, fall-through, and taken, respectively. Assume that CB2 when executed ended up being taken. The new prediction for the block is given by 2, as shown by the thick line in Figure 4(ii).

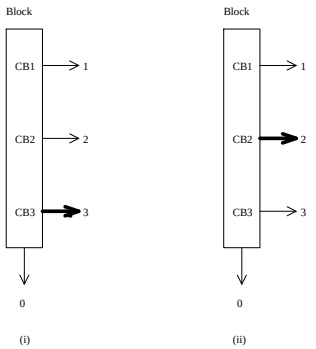

Figure 4: Recovery when Branch CB2 was Predicted to be Fall-through

Now consider the case when a branch was predicted to be taken, but ends up being fall-through. Finding a new prediction in this case is not so straightforward, as there could be more than one possible outcome for the block when the fall-through path of that branch is taken (if there are one or more conditional branches in the fall-through path). In such a scenario, the recovery mechanism requests the prediction mechanism to give a prediction greater than b or equal to 0 (corresponding to all conditional branches within the block taking the fall-through path). This is illustrated in Figure 5, for $b = 2$. The first figure shows a block whose outcome was originally predicted to be 2, as shown by the thick line. The corresponding predictions for the conditional branches CB1 and CB2 are fall-through and taken, respectively. Assume that CB2 when executed ended up being fall-through. The new prediction for the block could be either 3 (as shown by the thick line in Figure 5(ii)), or 0 (as shown by the thick line in Figure 5(iii)).

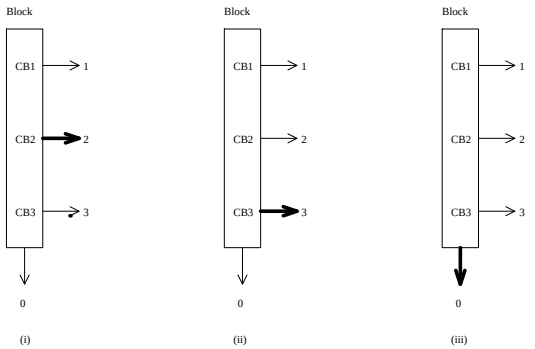

Figure 5: Recovery when Branch CB2 was Predicted to be Taken

2.5 Advantages of Block Prediction

Block-level prediction, as described here, has several advantages. First, it helps to predict the outcome of multiple branches per cycle. This is essential to improve the performance of wide-issue superscalar processors for non-numerical programs. Second, the fetch mechanism has to fetch only a sequential block of instructions. This is considerably easier than simultaneously fetching variable number of instructions that start from multiple addresses. Third, only a single prediction is made per cycle. Thus, the prediction mechanism need not be any more complex than an ordinary branch prediction scheme. One potential limitation of this technique (as will become evident in Section 4, where we present the experimental results) is that the *effective fetch size* (i.e., the number of (useful) instructions entered into the hardware window from a block) can be somewhat smaller than the maximum fetch size because of branches that are predicted to be taken. However, the compiler can attempt to modify the branch opcodes or schedule the instructions in such a manner that the conditional branches in the beginning of frequently-executed blocks most likely take the fall-through path (as done in [3] for the IBM RS 6000). This will cause most of the instructions of those blocks to be executed, thereby resulting in large effective blocks. Techniques for doing this is beyond the scope of this paper.

3 Hardware Cost

Before proceeding to do an experimental evaluation of the performance of block-level prediction, let us calculate the hardware cost for implementing the 2-level block predictor. The two hardware-intensive components of the block predictor are the BHT and the

PHT. Let us calculate the hardware costs for each of these components.

Let the number of entries in the BHT be N. The Tag field of a BHT entry requires as many bits as to uniquely identify a block address, and depends on N, the block address width (W), and the type of mapping used to select a BHT entry. In any case, the number of bits required for the Tag field will not exceed W. The Targets field of a BHT entry needs to store 4 target addresses (each requiring W bits). Lastly, the Block History Pattern field of a BHT entry stores the past p outcomes of the block mapped currently to that BHT entry, and requires $2p$ bits. Thus, the maximum number of bits required to implement the BHT is $N \times (5W + 2p)$ (which works out to be 10.75Kbytes for $N = 512$, $W = 32$, and $p = 6$).

Next, let us look at the number of bits required for the PHT. Because each Block History Pattern has $2p$ bits, the PHT has 2^{2p} entries. Each PHT entry needs to store 4 count values, each requiring, say c bits. The total number of bits for implementing the PHT is therefore given by $4c \times 2^{2p}$ (which works out to be 8Kbytes for $c = 4$ and $p = 6$). Alternate methods that reduce the hardware cost is a subject for future research.

4 Experimental Evaluation

The previous sections described block-level prediction, its implementation, and hardware cost. This section presents the results of an empirical study of its performance. For comparison purposes, we also evaluate the 2-level branch prediction scheme [14], augmented with a BTB to avoid the 1-cycle penalty due to branches that are predicted to be taken.

4.1 Experimental Setup

The performance studies are conducted using the MIPS instruction set architecture (ISA) [4], a representative of the class of streamlined ISAs that have emerged recently [5]. There are several implementations of the MIPS architecture, namely, R2000, R3000, R4x00, R6000, R8000, and R10000. (Of these, R8000 and R10000 are superscalar processors.) An important aspect of the MIPS architecture is the one cycle delay slot for all control-changing instructions such as branches, jumps, and calls; i.e., the instruction following a control-changing instruction is always executed if the control-changing instruction is executed. This architected delay slot presents a problem while deciding the block boundaries. A delay slot instruction must be filled in the same block that contains its corresponding control-changing instruction. If the last instruction in a fetch block A is a control-changing instruction C, then there is no vacant slot to accommodate its delay slot instruction in the same block. In such a situation, it is difficult to consider C's target to be one of the targets of block A, because control does not go to that target after the execution of block A. On the other hand, this target cannot be attached to the block containing the delay slot, say block B, because block B could potentially be entered through a different path. To avoid this problem, if the last instruction in a fetch block is a control-changing instruction, then it is not included in that block; instead that instruction is grouped together with its delay slot instruction in the subsequent block.

4.1.1 Simulation Tool and Benchmarks

All data reported in this paper are gathered with a simulator that accepts programs compiled for a MIPS-based DECstation and simulates their execution, keeping track of relevant information on a cycle-by-cycle basis. *The simulator models speculative execution, and is not trace-driven.* Because the entire processor is modeled in the simulator, the simulation results reflect reality to a very close extent. System calls made by the simulated programs are handled with the help of traps to the operating system. The collected results therefore exclude the code executed during system calls, but include all other code portions, including the library routines.

For benchmarks, we use 5 programs from the SPEC '92 integer suite: *compress* with input file *in*, *eqntott*, *espresso* with input file *bca*, *gcc* with input file *stmt.i*, and *xlisp* with input file *li-input.lsp*, which are all integer-intensive programs written in C. All programs were compiled using the MIPS C compiler; the SPEC benchmarks were compiled using the optimization flags distributed in the SPEC benchmark makefiles. The benchmarks were simulated to completion or up to 400 million instructions, depending on whichever occurred first.

4.1.2 Performance Metrics

For measuring performance, execution time is the sole metric that can accurately measure the performance of an integrated software-hardware computer system. However, this metric requires many implementation assumptions, including the exact hardware configu-

ration, functional unit latencies, etc. Other metrics, such as instruction issue rate and number of instructions in hardware window, also require implementation assumptions that limit the utility of results. Further, these metrics may not give much insight about the performance of the fetch mechanism. Therefore, for this study, we use the following (indirect) metrics, which provide good information about the fetch mechanism's performance: (i) *effective fetch size (EFS)*, (ii) *block prediction accuracy (BlPA)*, (iii) *branch prediction accuracy (BPA)*, and (iv) *block termination cause (BTC)*.

The first metric indicates the number of *useful* instructions fetched in each fetch attempt by the fetch unit. That is, instructions that are fetched but discarded because of a conditional branch being predicted to be taken, are not counted in this metric. Furthermore, useless instructions that are discarded due to incorrect speculative execution are also not counted while calculating this metric. Notice also that, if a block had to be fetched in 2 (or 3) attempts because of mispredictions, then it is counted as 2 (or 3) fetch attempts. Thus, EFS takes into account the effect of the inaccuracies in block prediction. This is an important metric because increasing its value is the primary goal of block-level prediction. Fetching more instructions per cycle gives the superscalar hardware window more opportunities to look for and exploit instruction-level parallelism.

The second metric of interest to us is the BlPA. It indicates the fraction of times a block prediction gives the correct target. Although the first metric completely captures the effect of block prediction inaccuracies, this metric throws more light on why the performance of the predictor is better or worse. We are also interested in knowing if making a 4-way block prediction instead of a 2-way branch prediction results in a poor prediction accuracy. The third metric, namely the BPA, indicates the fraction of times a branch (indirectly) gets correctly predicted when block prediction is performed. This metric is highly dependent on the second metric, but is helpful in carrying out comparisons with the BPAs obtained with other prediction techniques.

The last metric, namely the BTC, is highly instructive in that it helps researchers to chart directions for future research.

4.1.3 Caveat

It is important to make note of a caveat in our evaluation. The performance of any computer system is highly dependent on the compiler and the type of static scheduling done. All our studies are carried out with code compiled for a single-issue processor. The major implication of this decision is: no block-level prediction-specific optimizations were performed on the code; the code is scheduled for a single-issue processor, which can have serious implications on the performance of block-level prediction. In that sense, the performance results presented here should only be viewed as a realistic starting point. Further doctoring of the code, especially to make branches take the fall-through path more often than the taken path as in [3], will certainly improve the performance of block-level prediction further.

4.1.4 Default Parameters for the Study

We have attempted to do a reasonable study by varying the important parameters independently. When a parameter is varied, the rest of the parameters are kept fixed at their default values, given below:

- *Maximum Fetch Size (f)* The default value is 12 instructions.
- *Branch Predictor:* This scheme is simulated with a PAg 2-level branch predictor, with a pattern size (p) of 6. The BHT has 1024 entries, and is direct mapped. The PHT entries consist of 3-bit up/down saturating counters. The predictions are done based on the start address of each fetch block, so that the predictions can be done before decoding the instructions. *Therefore, strictly speaking, this scheme is a block-level predictor, with each block restricted to 2 outcomes.*
- *Block Predictor:* This scheme is simulated with a PAg 2-level block predictor. The BHT is also direct mapped, and the default value for the number of BHT entries is 1024. Each PHT entry has four 4-bit up/down saturating counters. The default pattern size is 6.

4.2 Experimental Results

4.2.1 Effective Fetch Size

Table 1 shows the EFS obtained with the default branch prediction scheme and the block prediction scheme, for $f = 8$ and $f = 12$. The first 2 columns of numbers present the EFS obtained with branch prediction for $f = 8$ and $f = 12$, respectively, and the

subsequent 2 columns present the same for block prediction. The last 2 columns present the percentage improvement in EFS when using block-level prediction, compared to the branch prediction case, for $f = 8$ and $f = 12$, respectively.

Program	*Br. Pred.*		*Block Pred.*		*% Incr.*	
	f=8	*f=12*	*f=8*	*f=12*	*f=8*	*f=12*
compress	4.72	5.74	5.84	7.22	23.73	25.78
eqntott	3.54	3.54	4.60	4.98	29.94	40.68
espresso	4.54	5.08	4.96	5.84	9.25	14.96
gcc (cc1)	4.55	5.12	4.75	5.62	4.40	9.77
xlisp	4.59	4.88	5.01	5.92	9.15	21.31

Table 1: Effective Fetch Size for $f = 8$ and $f = 12$

Let us look at the results of this table in some detail. The percentage improvement in EFS for $f = 8$ varies from 4.4% (for gcc) to 29.9% (for eqntott). When f is increased to 12 instructions, the percentage improvement for block-level prediction over branch prediction has improved, and varies from 9.8% (for gcc) to 40.7% (for eqntott). For all benchmarks (except compress), the percentage improvement in EFS has improved significantly, when f was increased from 8 to 12 instructions. This is because the branch predictor is fundamentally limited by the basic block size, and therefore does not improve much when f is increased. The block-level predictor, on the other hand, tends to gain when f is increased[3].

4.2.2 Prediction Accuracy

Next, let us look at the prediction accuracies obtained. Table 2 shows the prediction accuracies that we obtained for $f = 12$. The first column of numbers gives the BPA obtained with the branch prediction scheme. The second column of numbers gives the BlPA obtained with the block-level prediction scheme. The last column of numbers gives the BPA that was obtained when block-level prediction was carried out. Let us look at these results closely. The first thing to notice is that the two-level prediction scheme has worked very well for both branch prediction and block prediction schemes. Second, the BPA obtained for the block-level prediction scheme is very close to the BPA obtained for the branch prediction case. This result is very encouraging, because it shows that predicting one out of 4 outcomes has provided more or less the same accuracy as predicting one out of 2 outcomes.

[3]This increase is likely to taper off at some point, due to conditional branches that cause control to flow out of blocks from the middle (c.f. Table 3).

(In the block prediction scheme results for compress, the BlPA is higher than the BPA because BlPA includes blocks that do not have any branches encompassed within.)

Program	*Br. Pred.*	*Block Predictor*	
	BPA	*BlPA*	*BPA*
compress	86.99%	87.63%	86.96%
eqntott	94.82%	92.15%	94.64%
espresso	95.36%	91.79%	94.68%
gcc (cc1)	84.94%	81.16%	83.21%
xlisp	92.56%	86.53%	92.40%

BPA - Branch Prediction Accuracy;
BlPA - Block Prediction Accuracy

Table 2: Prediction Accuracies for $f = 12$

4.2.3 Block Termination Cause

Table 3 shows the BTC results for $f = 12$. We have categorized the BTC into 4 categories, as indicated by the last 4 columns of Table 3. From the table, we can deduce that the main reason for the blocks to be terminated (before reaching the fetch size) is due to conditional branches that are predicted to be taken, and cause a control flow out of the block from somewhere in the middle of the block.

Program	4^{th} *CB*	*Call & Return*	*UB & Jumps*	*CB Taken*
compress	0.00%	6.70%	24.24%	69.10%
eqntott	0.15%	6.60%	7.30%	85.94%
espresso	0.21%	9.65%	5.27%	84.87%
gcc (cc1)	2.97%	20.59%	20.70%	55.74%
xlisp	0.19%	43.78%	15.49%	40.54%

CB - Cond. Branch; UB - Uncond. Branch

Table 3: Block Termination Cause for $f = 12$

4.2.4 Sensitivity Studies

Next, let us vary some of the parameters (that were fixed so far) in order to study their effect on the block predictor's performance. The parameters that are of interest to us in these sensitivity studies are: (i) maximum fetch size (f), (ii) the pattern size (p), and (iii) the number of entries in the BHT (N). Each of these parameters is varied while keeping the rest of the parameters at their default values specified in Section

4.1.4, and the metric used was the effective fetch size (which is our primary metric).

Let us start by considering the effect of f on EFS. Figure 6 plots the EFS obtained for 3 different values of f, namely 8, 12, and 16. As f is increased, EFS also increases. But, the increase is sub-linear, and is likely to taper off, as f is increased further, because of many conditional branches being predicted to be taken.

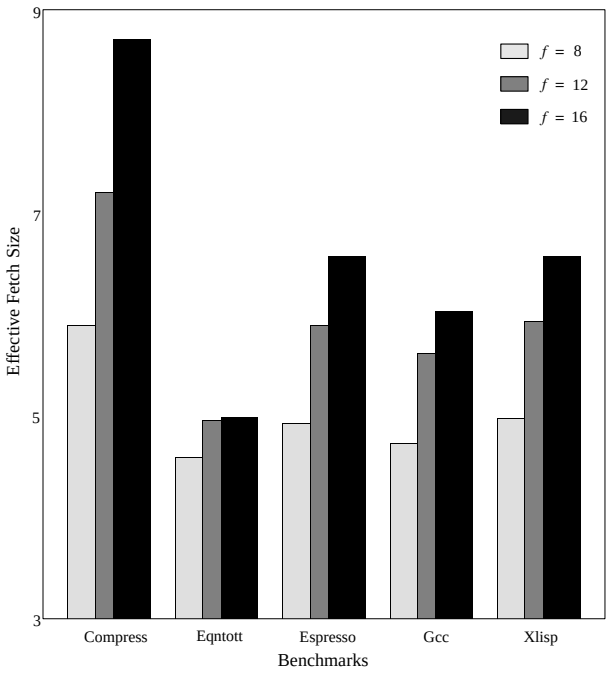

Figure 6: Effect of f on EFS for block predictor

Next, let us study the effect of varying the pattern size p. Figure 7 plots the EFS obtained for the benchmarks for $p = 4$, 6, and 8. It can be seen from the figure that a pattern size of 6 is quite adequate for the block predictor.

Lastly, let us look at the effect of the number of entries in the BHT (N) on the performance of the fetch mechanism. Figure 8 plots both the EFS and the BHT hit rates obtained for the benchmarks for different values of N. Except for gcc, all other benchmarks stand little to benefit from a value of N more than 1024. For programs that require large N, such as gcc, better hit rates can be achieved with smaller N by using set-associative mapping to select the BHT entries.

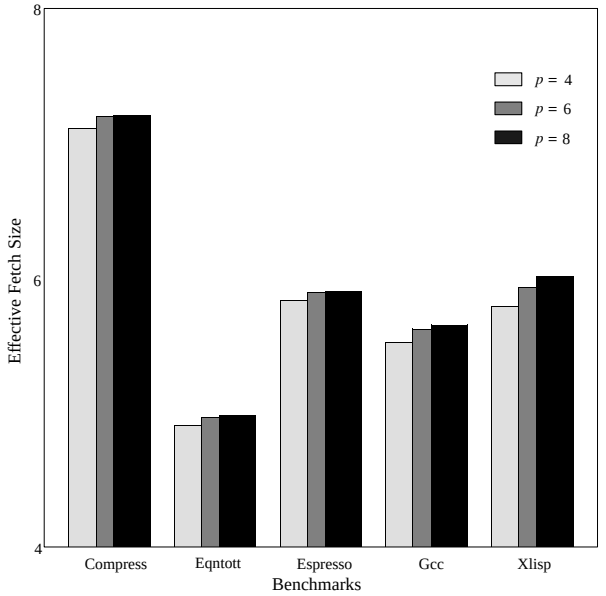

Figure 7: Effect of p on EFS for block predictor

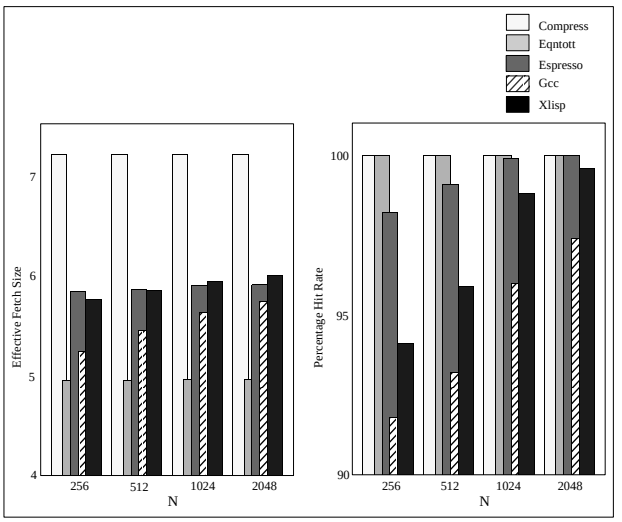

Figure 8: Effect of N on EFS and BHT Hit Rate

5 Conclusions and Future Work

This paper proposed a block-level prediction mechanism for wide-issue superscalar processors. Instead of predicting the outcome of each conditional branch, this scheme predicts the target of an entire (sequential) block of instructions (*i.e.*, it predicts where control is most likely to go after the execution of the

block). This helps to effectively predict the outcome of multiple branches per cycle, thereby enabling the fetch unit and the decoder to bring in more instructions per cycle into the superscalar hardware window. The main advantage of this scheme is the simplicity of the fetch mechanism. The fetch mechanism makes only one prediction per cycle, but could be effectively predicting the outcome of up to 3 branches.

We also conducted a thorough simulation study to verify the potential of this prediction strategy. The results of our experiments with the MIPS architecture help us to conclude that a block-level prediction approach to superscalar processors is profitable, especially when the fetch size is large. It was found to increase the effective fetch size by about 25% over that of branch prediction schemes. Our studies also showed that the main reason for the blocks being terminated prematurely was because the targets returned by the block prediction mechanism quite often forced one of the branches contained within the block to be predicted to be taken.

Future work therefore includes fetching non-straightline code sequences in a cycle so that the fetch mechanism can continue to fetch along the targets of the predicted branches as in [13]. However, we would like to do it without making several individual predictions, to retain the simplicity of our block-level prediction mechanism. Another observation that we made in our experiments was that there were many short branches due to short 'if-then' and 'if-then-else' code sequences. It might be a good idea to encompass these short branches completely within a block, so that those branches do not add to the block's targets.

Acknowledgements

This work was funded by the Department of Electrical and Computer Engineering of Clemson University, and by the NSF Research Initiation Award Grant CCR 9410706.

References

[1] *Byte*, November 1994.

[2] M. Franklin, "The Multiscalar Architecture," *Ph.D. Thesis, Technical Report TR 1196*, Computer Sciences Department, University of Wisconsin-Madison, 1993.

[3] M. C. Golumbic and V. Rainish, "Instruction Scheduling Beyond Basic Blocks," *IBM Journal of Research and Development*, Vol. 34, No. 1, pp. 93-97, January 1990.

[4] G. Kane and J. Heinrich, *MIPS RISC Architecture.* Englewood Cliffs, NJ: Prentice-Hall, 1992.

[5] J.L. Hennessy and D.A. Patterson, *Computer Architecture A Quantitative Approach.* San Mateo, CA: Morgan Kaufmann, 1990.

[6] P. Y. T. Hsu and E. S. Davidson, "Highly Concurrent Scalar Processing," *Proceedings of 13th Annual Symposium on Computer Architecture*, pp. 386-395, 1986.

[7] M. Johnson, *Superscalar Microprocessor Design.* Englewood Cliffs, NJ: Prentice-Hall, 1991.

[8] J. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *IEEE Computer*, pp. 6-22, January 1984.

[9] S-T. Pan, K. So, and J. T. Rahmeh, "Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation," *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pp. 76-84, 1992.

[10] D. N. Pnevmatikatos, M. Franklin, and G. S. Sohi, "Control Flow Prediction for Dynamic ILP Processors," *Proceedings of the 26th Annual International Symposium on Microarchitecture (MICRO-26)*, pp. 153-163, 1993.

[11] D. N. Pnevmatikatos and G. S. Sohi, "Guarded Execution and Branch Prediction in Dynamic ILP Processors," *Proceedings of the 21st International Symposium on Computer Architecture*, 1994.

[12] S. Weiss and J.E. Smith, *POWER and PowerPC.* San Francisco: Morgan Kaufmann, 1994.

[13] T-Y Yeh, D. T. Marr, and Y. N. Patt, "Increasing Instruction Fetch Rate via Multiple Branch Predictions and a Branch Address Cache," *Proceedings of the 7th ACM International Conference on Supercomputing*, pp. 67-76, July 1993.

[14] T-Y Yeh and Y. N. Patt, "Alternative Implementations of Two-Level Adaptive Branch Prediction," *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 124-134, 1992.