1998 International Computer Symposium
Workshop on Computer Architecture
December 17-19, 1998, N.C.K.U., Tainan, Taiwan, R.O.C.

Improving Branch Target Prediction with Register References

Yueh-Hung Liu and *Chang-Jiu Chen*

Department of Computer Science and Information Engineering
National Chiao Tung University

ABSTRACT

Branch instruction can interrupt the steady flow of instruction stream in the pipeline. To resolve this problem, various branch prediction schemes have been proposed and some of these schemes can achieve high prediction accuracy. However, there is a fact that the accuracy of branch target prediction is usually not as high as the accuracy of branch direction prediction.

In this paper, we propose a new branch prediction mechanism to attempt to increase the prediction accuracy of branch targets. This mechanism will refer the register file and make predictions according to the register contents.

We simulate our design using the SimpleScalar tool set and compare our scheme with a basic predictor model on some of the SPEC95 benchmarks. The simulation results show that the average improvement on prediction accuracy of branch targets are 1% ~ 9% for different benchmarks. In addition, from our experiments we obtain an advantage of this newly proposed prediction scheme that we can use less hardware to get higher accuracy.

1. Introduction

Branch instructions are always the performance bottleneck of modern pipelined superscalar processors. They can interrupt the steady flow of instruction stream in the pipeline. To resolve this problem, various branch prediction schemes have been proposed and some of them can achieve high prediction accuracy.

The problem of branch prediction can be divided into two parts: branch directions and branch targets. For branch directions, the frequently discussed two-level branch predictor has an excellent performance. For branch targets, the common BTB-based mechanism and the recently proposed target cache scheme are used. Now the accuracy of predicting branch directions can achieve 90% ~ 99% correct prediction. However, the accuracy of branch target prediction is usually not as high as branch direction prediction.

In this paper, we present a branch prediction mechanism that try to increase the branch target prediction accuracy for both direct branches and indirect branches. Our mechanism is slightly differed from most recent schemes, We do not use any branch history as its prediction information, but use the contents of registers to predict branch targets. We call this mechanism as *register reference branch predictor (RRBP).*

2. Register Reference Branch Predictor

2.1 RRBP Structure

The RRBP is a branch predictor based on registers reference and the BTB structure. It attempts to improve the prediction accuracy of branch target archived by BTB-based branch prediction schemes commonly used in the modern pipelined superscalar processors.

2.1.1 Organization

The organization of the RRBP is shown below.

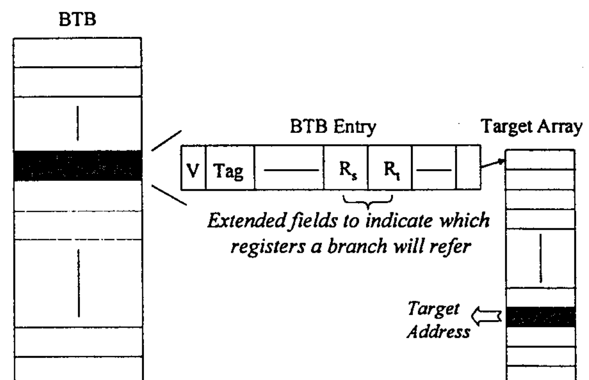

Figure 1: Logical organization of the RRBP

There is also a BTB in this branch predictor, but each entry of the BTB has been extended. In addition to the branch address and valid bit, we add fields to record registers a branch will refer. The number of these fields is dependent on the instruction set architecture (ISA), especially the format of branch instructions.

For each instruction, we only mention the source operands. If a specified instruction set is a two sources instruction set then we will allocate two fields for each BTB entry. And if the branch instructions haven't any source operand, there are some unused fields and become inevasible redundancies. Since a branch can read any register in the register file, thus each field should be long enough to distinguish different registers.

There is another extension for the BTB entries. We extend the target address storage to two or more fields instead of one field in the ordinary BTB. These target address storage fields form an array, we call it *target array*. For each BTB entry, there is one and only one target array associated with it. The number of the target array entries is not fixed, but due to the high hardware cost and in our results, two entries for each target array are enough to obtain good prediction accuracy.

2.1.2 Characteristics

The basic observation behind the RRBP is that some branches can be more accurately predicted if the values of source registers are known at the predicting time. Consider the following 3-statement code.

```
if ( a = 0 ) { r1 = r2 = 50; }      /* a1 */
else { r1 = 0; r2 = 100; }          /* a2 */

if (r1 == r2 ) { ... }              /* b1 */
```

The condition in statement *b1* is dependent on statement *a1* and *a2*. Statement *a1* and *a2* are executed exclusively; that is, there is only one of them can be executed at any time. Every time the program including these statements arrives at *b1*, the phenomenon described above will appear. If we want to predict whether the condition in *b1* is true or not, that is, whether the branch in *b1* will be taken or not, we can refer to registers r1 and r2 while arriving at *b1*. Since the RRBP allocates more than one field for each BTB entry (the target array) to store targets, thus if the branch associated with *b1* is in the BTB; we can predict one target from the target array according to the values of registers r1 and r2.

The branch history is unnecessary, we just have to know what the values of registers r1 and r2 are at the moment *b1* is encountered. However, the register value is a kind of correlation information, since while a specified instruction is encountered, the registers may usually keep some specified states. For branches, with different register states, especially the states of source registers, the behavior of branches may change. The RRBP attempts to record the relation between source registers and targets of branches, and uses such relationship to predict where branches probably go.

For indirect branches, the RRBP will have higher accuracy of target prediction than traditional BTB-based schemes because of the target array mechanism. In traditional BTB-based schemes, although we can expect that an indirect branch may jump to a different address each time it is executed, we still have only one possible target to predict. In the RRBP, there are multiple selections to predict a target for any indirect branch, and hence the probability of correct prediction is increased.

2.2 RRBP Manipulation

Manipulation of the RRBP is concerned with the issues of accessing and updating the RRBP. Good manipulation schemes can significantly improve the performance of the RRBP.

2.2.1 Accessing the RRBP

While an instruction is fetched at the IF stage of pipeline, the extended BTB of the RRBP should be indexed. As the traditional BTB, the extended BTB of the RRBP is indexed using the instruction address. The lower bits of address are formed an index into the BTB, then the tag and valid bit of indexed BTB entry will be checked. If there is an entry in the BTB associated with the instruction, we know the instruction is a branch instruction and the information contained in the indexed entry will be used to predict a target.

Otherwise, if there is no entry associated with the instruction, the next instruction address will be returned; that is, if the instruction is a branch, the branch is predicted non-taken. In addition, if there is no entry associated with the branch instruction, a new entry will be allocated in the BTB for the branch.

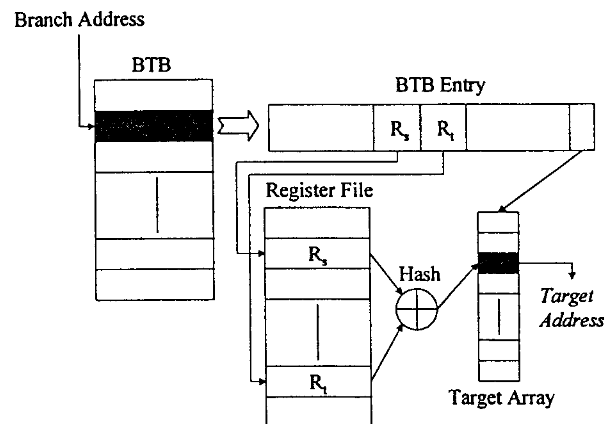

Figure 2: Accessing the RRBP to make a prediction

As shown in Figure 2, when a proper BTB entry is found, the additional fields that record which registers the associated branch will read, as illustrated in section 2.1.1, are accessed.

According to these fields, we can know which registers should be referred to make a target prediction, and then we access the register file to read the contents of the specified registers. The contents of the specified registers will be hashed together to form an index into the target array of the BTB entry associated with the predicted branch. The hash function used here is simply the logical function Exclusive-OR.

In our experiments, we use the *least significant bits (LSB)* to index the target array. We suppose that the values to be compared are not usually too large, thus the least significant bits can reflect the difference of different branch instance. The other branch instructions in the

instruction set are unconditional direct/indirect branches. We don't care the direct branches because they won't access any register. For each indirect branch, it will jump to an address stored in a register and we guess the least significant address bits of different branch instance are usually distinguishable. Thus we also use the least significant bits of register values to index the target array for indirect branches.

Finally, when we index the target array for a branch successfully, the address stored in the indexed entry will be the predicted branch target.

2.2.2 Updating the RRBP

When a branch is finished, the BTB of the RRBP must be updated with the actual outcome of the branch. Which BTB entry should be updated is decided at the time the BTB was accessed with the branch address as an index. If there is an entry associated with the specified branch, then this entry will be updated; otherwise, a new entry should be allocated in the BTB.

We can know an instruction whether it is a branch or not at the ID stage of pipeline, and if it is a branch without an associated BTB entry, we must allocate one for it at the ID stage or later. For the new BTB entry allocation, if the BTB is direct mapped, we simply replace the indexed entry with the outcome of the "new branch". But if the BTB is set-associative, we have to select a "suitable" entry for the new branch. Trivially, an "empty" entry is always a suitable entry. However, if all entries in the indexed set are in use that is, each entry had been allocated for a different branch, then we must select one to be replaced. We use a common scheme called the *least recently used (LRU)* algorithm to select an entry to be replaced.

In addition to the original BTB fields, such as the tag field (branch address) and valid bit, there are other fields should be filled. First, we have to fill the additional fields that record the register reference; that is, we should determine which registers a branch will refer to. This work can be done while allocating a new branch entry in the BTB at the ID stage of pipeline or later, because after the instruction decoding, the opcode and operands of that instruction can be known. The other fields should be filled are the entries of the target array, and we initiate each entry of the target array with the actual resolved target address. In other words, if a branch is really taken, we fill each target array entry with the address the branch jumps; otherwise, the fall-through address will be used instead. Note that for a non-taken branch, the BTB also has to be updated with the non-taken target address (i.e. the fall-through address).

This situation differs from the traditional BTB-based schemes since the RRBP doesn't have any mechanism of direction prediction. The work of filling the target array can't be done at the ID stage of pipeline since the actual target address has not been resolved yet. After the branch instruction executing at the EX stage, we can know the resolved address and can fill it into the target array.

If there is already a BTB entry for an executed branch, the only work we have to do is to update one of the target array entries. The additional fields that record the register reference need not be updated, since registers a branch will refer are fixed at compile-time. Hence, once such information exists in the BTB entry for a branch while the entry was allocated in the BTB, it can be used for each instance of the branch.

In order to update the target array, we must know the resolved target address and which entry should be updated. The resolved target address can be determined at the EX stage of pipeline as described above. Now the question is which target array entry should be updated. The entry should be updated is that was used to predict the branch target. If the prediction for a branch is correct, no updating is needed; otherwise, replacing the updated entry with the resolved target address.

The action of updating the RRBP implies a fact that the information used to accessing the RRBP at the IF stage of pipeline should be maintained through all pipeline stages or at least kept until branches finish their execution. For instance, the instruction address should be kept for allocating a new entry in the BTB at the ID stage if a branch without an associated BTB entry is encountered. Moreover, the hashing result used to index the target array should be passed to the EX stage in order to process the updating. The information is transferred between stages, thus there must be some mechanism to buffer it, such as latches.

3 Simulation Results

In this section, we will examine the prediction accuracy using the RRBP. Three types of accuracy will be shown separately: branch target prediction, branch direction prediction, and indirect branch target prediction. We will also examine the execution performance of the RRBP in terms of *instructions per cycle (IPC)*. The simulation results will be compared with a base predictor model to express the advantages of the RRBP.

3.1 Experimental Methodology

In order to simulate the RRBP design, we employ the SimpleScalar tool set (version 2.0)[3].

3.1.1 Benchmarks

To experiment for the efficiency of the RRBP design, we will use a subset of the SPEC95 benchmarks. We only select 8 of them to simulate (4 are from SPECint95 and 4 are from SPECfp95).

3.1.2 Experimental Model Configurations

The out-of-order issue, superscalar processor simulator in the SimpleScalar tool set employs a 16-entry *register update unit (RUU)* along with 4 integer ALUs and 4 floating point ALUs. It can decode 4 instructions at one

time, and can issue and execute up to 4 instructions simultaneously if there are no data dependency among these instructions to be executed.

In this paper, we use a branch predictor model **Base** as the comparison base for the RRBP, as shown in Figure 3. The **Base** model employs a two-level adaptive branch predictor that uses a global history register as its first level and a PHT *(Pattern History Table)* as its second level to predict the direction of branches. The length of the global history register is 8-bit long and the size of the PHT is 1024 entries. The hash function used to index the PHT is *gselect*, this function will concatenate the global history and the branch address bits to form an index. Moreover, there is also an ordinary BTB (without any extension) to store the branch targets in the **Base** model.

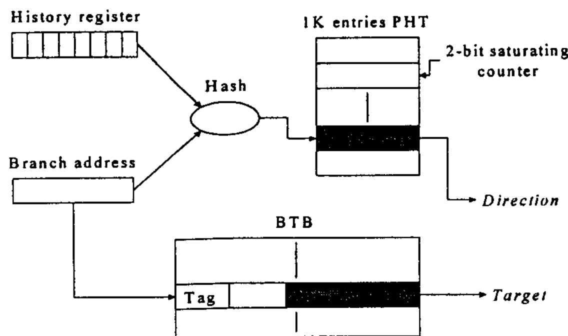

Figure 3: Logical organization of the **Base** model

We simulate the RRBP design using an extended BTB model as described in section 2. We will allocate two additional fields to indicate the register reference for each BTB entry. We use the function XOR to hash two register values for two-source branches and access the register directly (no hashing) for one-source branches to form an index into the target array. The zero-source branches are all the unconditional direct branches and therefore need not be predicted they will always jump and always jump to a single fixed location. In addition, for return instructions (a type of indirect branches), we allocate an 8-entry *return stack* for each model to store the return address of function calls, thus the targets of return instructions are obtained from the return stack, not from the BTB.

3.2 Number of Target Array Entry

In this section, we will examine the effect of target array size. The results for target arrays of 2 entries and 4 entries are plotted in Figure 4 and Figure 5, respectively, for the various benchmarks. The prediction accuracy in these figures is the average of various set-associative configurations, including 256 to 1024 sets, direct-mapped to 4-way associativity, and etc. Both Figure 4 and Figure 5 have three independent charts, these charts illustrate the accuracy of branch target prediction, branch direction prediction, and indirect branch target prediction, respectively. Note that the prediction accuracy of branch target and branch direction include both the direct branches and the indirect branches.

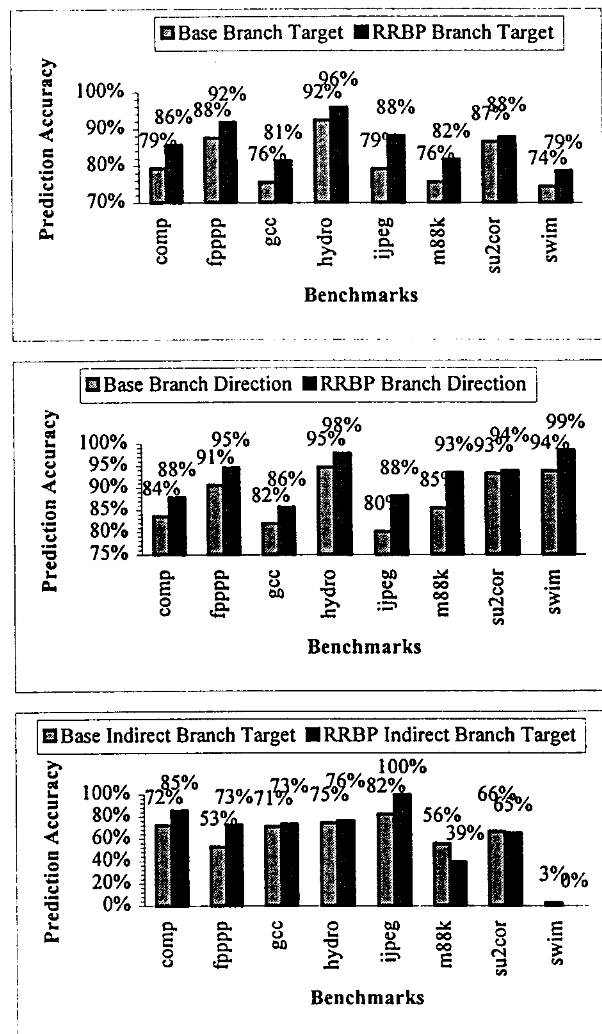

Figure 4: Prediction accuracy of branch target, branch direction, and indirect branch target using 2 target array entries

For branch target and branch direction prediction, all of the simulated benchmarks have higher accuracy than the **Base** model while using the RRBP with 2 target array entries or 4 target array entries. For indirect branch target prediction, however, most benchmarks also have higher accuracy on the RRBP except *m88ksim*, *su2cor*, and *swim*; especially for *swim*, the prediction accuracy is almost zero (0.02%). But for *swim*, even the **Base** model can only achieve a prediction accuracy of 3%. This situation tells us that the indirect branches in *swim* may have a different target each time a branch is encountered and therefore hard to predict. It is also possible that the interference for indirect branches in *swim* is serious, thus the information in the BTB could be "polluted".

For *su2cor*, the prediction accuracy of indirect branch target for the RRBP is only 1% (in fact, less than 1%) less than the **Base** model; it will not affect the whole efficiency too much and hence can be ignored. The difference between the RRBP and the **Base** model for *m88ksim* is about 17%, it probably reflects that indirect branches in *m88ksim* will jump to the last targets more often but the RRBP always take them to another locations.

Although *m88ksim*, *su2cor*, and *swim* have worse prediction accuracy of indirect branch target using the RRBP, the prediction accuracy of whole branch targets (the first chart in each figure) still outperforms the **Base** model. This point means that the advantage brought by accurate prediction of direct branch target may be greater than the damage of inaccurately predicted target of indirect branch. Note that the direction prediction accuracy is always greater than or equal to the target prediction accuracy; it is trivial since we can know an unconditional indirect branch will jump but we can't conclude where it will jump to.

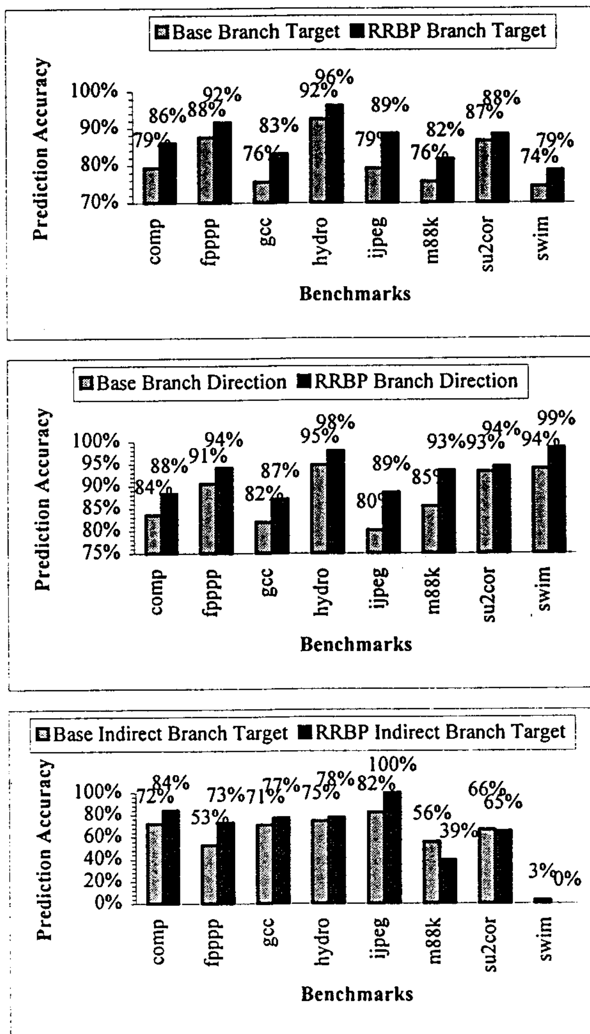

Figure 5: Prediction accuracy of branch target, branch direction, and indirect branch target using 4 target array entries (cont.)

Figure 6 compares the execution performance of the RRBP with the Base model for the experimental benchmarks in terms of instructions per cycle (IPC). Again, the IPC in the figure is the average of various set-associative configuration results. Note that while using the RRBP, each benchmark has better performance except *su2cor*, which is slightly worse than using the **Base** model. This is because the statistic is an average value, and there could be some configurations with much worse performance and others with little better performance, therefore the final result is slightly worse; this point can be verified in Table 1. In Table 1, we compare the IPC of **Base** model with the RRBP (2 target array entries) for *su2cor*, and the IPC improvement is also listed. We can see that for the set associativity of lower degree, the IPC improvement is much worse (-1% ~ -7%); but for the set associativity of higher degree, the IPC improvement is only little better (less than 1%); hence the average value is negative. This situation can be explained as that since the conflict in the RRBP is more serious due to the non-taken branch allocation policy, therefore the useful prediction information could be replaced frequently, especially when the degree of set associativity is low.

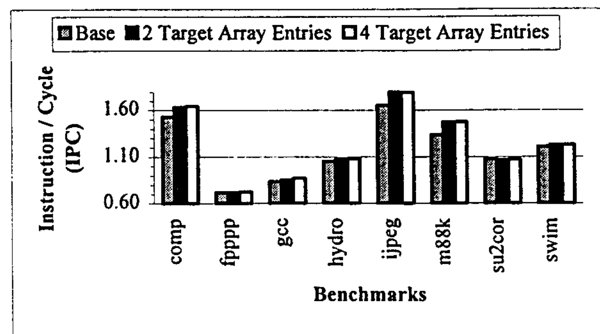

Figure 6: Execution performance comparison in terms of IPC

	Base	2 TA Entries	Improvement
Direct-mapped 256 Sets	1.028	0.9525	-7.31%
2-way 256 Sets	1.0717	1.0345	-3.47%
4-way 256 Sets	1.0889	1.0947	0.53%
Direct-mapped 512 Sets	1.0475	1.0294	-1.73%
2-way 512 Sets	1.088	1.0938	0.53%
4-way 512 Sets	1.0889	1.0988	0.91%
Direct mapped 1024Sets	1.0782	1.0566	-2.00%
2-way 1024 Sets	1.0889	1.0988	0.91%
4-way 1024 Sets	1.0889	1.0988	0.91%
average	1.0743	1.0620	-1.14%

Table 1: IPC improvement using the RRBP with 2 target array

Benchmark	Improvement of 2 TA entries	Improvement of 4 TA entries
compress	6.84%	7.58%
fpppp	0.16%	0.70%
gcc	2.21%	4.21%
hydro2d	2.51%	2.75%
ijpeg	8.73%	8.49%
m88ksim	10.47%	10.43%
su2cor	-1.14%	-0.66%
swim	1.45%	1.45%

Table 2: IPC improvement using the RRBP

For reference, the IPC improvement for each benchmark while using the RRBP instead of the **Base** model is listed in Table2. Figure 7 to Figure 9 express the relation between the number of target array entries and the prediction accuracy of branch target, branch direction, and indirect branch target, respectively, for the benchmarks *gcc*, *m88ksim*, and *swim*. Figure is the comparison of IPC on various target array sizes for *gcc*, *m88ksim*, and *swim*. We

select these three benchmarks since they have some special properties on branches. Based on the benchmarks, we observe that *gcc* and *m88ksim* have the lowest instructions per branch, and that means branches appear more frequently in *gcc* and *m88ksim*. In addition, *m88ksim* and *swim* have the highest indirect branch rate, and that means they have more possibility to "meet" a branch that is an indirect branch.

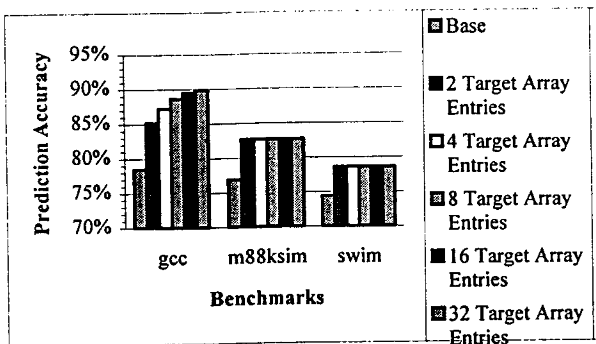

Figure 7: Branch target prediction accuracy of various target array sizes

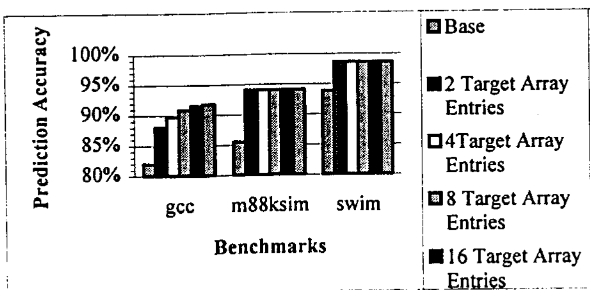

Figure 8: Branch direction prediction accuracy of various target array sizes

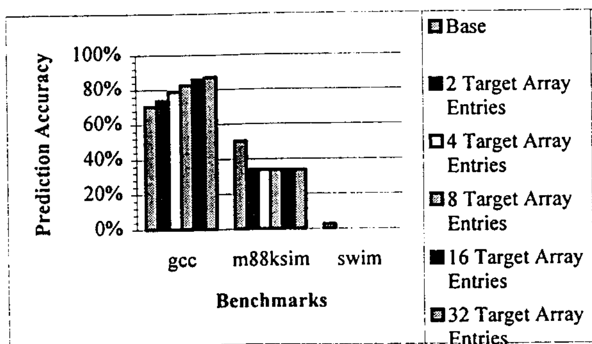

Figure 9: Indirect branch target prediction accuracy of various target array sizes

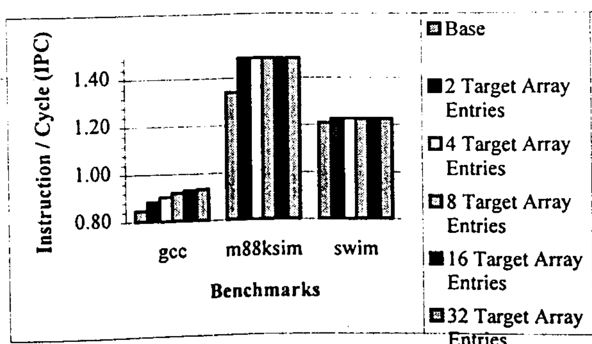

Figure 10: Execution performance of various target array sizes

As shown in these four figures above, when the number of target array entries increase, there are almost no change for the prediction accuracy or for the performance of *m88ksim* and *swim*; that is, the target array size of 2 entries is good enough to predict branches. For *gcc*, the prediction accuracy and the execution performance are increased as the target array size increasing, but the distance of improvement is under 5%, and it is disproportional to the hardware cost for the target array increment. According to this observation, we can conclude that 2 target array entries are enough for most applications.

4. Performance and Hardware Cost

In order to make a prediction for a branch, we should first index the BTB of the RRBP using the branch address. If there is an entry associated with the branch, the additional fields will be accessed to index the register file. Then, the appropriate register values will be hashed to index the target array and finally the content of the indexed target array entry is the predicted target. According to these actions, three indexing and one hashing are needed. In addition, each indexing brings an access, and hence three accesses are also necessary. These actions should be proceeded while a branch instruction is fetched into the pipeline at the IF stage and completed in one cycle.

How many actions do other BTB-based schemes take to make a prediction? For a traditional BTB combined with a two-level adaptive direction predictor, in order to predict a branch, two indexing are needed. First, the branch address is used to index the BTB and the branch history table simultaneously; second, the indexed branch history table entry is used to index the pattern history table. In these indexing processes, three tables are accessed: the BTB, the branch history table, and the pattern history table. But since the BTB and the branch history table are accessed simultaneously, we can evaluate them, as just one access is needed. Sometimes a register called *branch history register* will substitute the whole branch history table. In this situation, two indexing are also necessary, one for the BTB, and the other for the pattern history table. Before indexing to the pattern history table, the branch history is usually hashed with the branch address; thus one hashing is sometimes needed.

In the scheme just discussed, the actions taken in BTB to predict a branch are less than the RRBP does. The RRBP takes one more indexing and one more access. However, if we combine the access of the target array with the access of the BTB, and then use the hashing result of registers to decide which target array entry should be used. Just as the pattern history is used to decide whether the target address in a BTB entry should be applied or not in the above-discussed scheme, we can reduce the access and indexing actions to two times. In other words, the performance of the RRBP will not worse than the common BTB-based mechanism. However, the delay for obtaining a prediction is fixed on the time to perform two or three indexing/access and hard to be truncated. If there are multiple branches in one fetched instruction line and we

just want to know which of them are predicted taken, these branches will be predicted one by one and hence the predicting actions are difficult to finish in one cycle due to the fixed long delay. For traditional BTB-based mechanism, if there are prediction bits in each BTB entry, the above situation can be easily resolved since only one indexing/access is needed to determine whether a branch is taken or not. In our simulations, we restrict that there can be only one branch in one fetch cycle and therefore we do not consider this problem now.

For updating the RRBP, we need only update the appropriate target array entry. For the BTB with a two-level predictor, however, the history tables and the BTB entry all need to be updated. In the updating phase, the RRBP is simpler than the common BTB-based scheme.

One of the main extra hardware to implement the RRBP is for the additional fields used to indicate register reference and for the target array. If the instruction set architecture is a two-source architecture and there are at least 32 general-purpose registers, the cost of each additional filed is at least 5 bits and there are two fields for each BTB entry. That is, the cost of additional fields for each BTB entry is at least 10 bits. In addition, since the register file will be accessed (read) while making a prediction, the read ports of the register file should be increased. In the above situation, there are at least two extras read ports should be added to the register file. Moreover, increasing the read ports will bring additional complexity on implementing the register file.

However, the potential cost is for the target array. Each target array entry should have capacity to store an instruction address as the target address filed in the BTB entry. In general, there are multiple entries for a target array, and hence the cost will be times of the target address field in the BTB entry. The cost of the RRBP is more than the common BTB-based scheme (with a two-level predictor), although there are branch history table and pattern history tables in the common BTB-based mechanism. The cost of these history tables is less than the target array. For the pattern history table, each entry is a 2-bit saturating counter; for the branch history table, each entry is a shift register, and the length of each register is usually 6 to 10 bits. In addition, most recent designs of predictor use a branch history register instead of a branch history table; that is, the cost of the two-level predictor is significantly decreased.

Of course it is always a trade-off between the hardware cost and the prediction accuracy. In our experiments, the branches are more accurately predicted; thus we may conclude that we have obtained the advantage from the higher hardware cost of the RRBP.

5 Conclusions

Although the current branch predictors are effective in predicting the directions of branches, the prediction accuracy of branch targets is not as high as branch directions. In this paper, we proposed a new branch predictor, the register reference branch predictor (RRBP), to try to achieve higher prediction accuracy of branch targets.

The RRBP extends the traditional BTB and refers the register values to make prediction for branches. In our experiments, we compared the RRBP to a general (two-level branch predictor + BTB) model and the RRBP outperformed this predictor model. The RRBP could have higher hardware cost due to its target array mechanism. But fortunately, our experiments showed that a target array of 2 entries could obtain a reasonable better performance. We had examined that the RRBP with low degree of set associativity (and hence less hardware cost), for example, 256 entries and 2-way set-associative, can outperform the compared predictor model with higher degree of set associativity (ex. 1024 entries, 4-way set associative; hence more hardware cost). The selections of the indexing bits for target array were also examined. The results showed that the least significant bits are more useful than other indexing bits selections.

In addition to the content dealt in this paper, there are still other subjects should be discussed and evaluated in the future. For the branch without an associated BTB entry, the RRBP always allocates an entry for this branch at the time it is encountered, whatever it is taken or non-taken, since the RRBP has no any direction predictor. This scheme could lead to a situation that the interference would be especially serious. Therefore, we will improve our design to resolve this problem in the future.

In the RRBP, the register values are hashed together to form an index into the target array; the hash function used in this paper is XOR. Although it performed a good experimental result, however, if we attempt to further distribute different occurrence of branches on the target array, we have to design a more appropriate hash function with or without other information to be the hashed arguments. Furthermore, there are still other selections of indexing bits, such as odd bits or even bits, etc., can be investigated to probably obtain the better performance.

The problem of data dependency is still existing in the current RRBP design. It is certain that the incorrectly referred register values will degrade the prediction accuracy. For example, each time a branch is predicted, the RRBP may read a different register value, and thus a different prediction is made, but the real value may be the same each time the branch is encountered! We can expect that if the problem of data dependency is resolved, the efficiency will be risen; it is also a researchable subject.

As described in this paper, there is a restriction on the RRBP that it is hard to predict multiple branches (in one fetched instruction line) in one cycle because the prediction delay for each branch is somehow long. In actual fact, however, there is sometimes more than one branch instruction that could be fetched into the pipeline in one cycle. Therefore, to truncate the prediction delay of the

RRBP is important.

References

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

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

[3] D. Burger and T. M. Austin, "The SimpleScalar Tool Set Version 2.0," *Technical Report 1342,* Computer Sciences Department, University of Wisconsin, Madison, WI, 1997.