Cost-Efficient Branch Target Buffers

Jan Hoogerbrugge

Philips Research Laboratories, Prof. Holstlaan 4, 5656 AA Eindhoven, The Netherlands
jan.hoogerbrugge@philips.com

Abstract. Branch target buffers (BTBs) are caches in which branch information is stored that is used for branch prediction by the fetch stage of the instruction pipeline. A typical BTB requires a few kbyte of storage which makes it rather large and, because it is accessed every cycle, rather power consuming. Partial resolution has in the past been proposed to reduce the size of a BTB. A partial resolution BTB stores not all tag bits that would be required to do an exact lookup. The result is a smaller BTB at the price of slightly less accurate branch prediction. This paper proposes to make use of branch locality to reduce the size of a BTB. Short-distance branches need fewer BTB bits than long-distance branches that are less frequent. Two BTB organisations are presented that use branch locality. Simulation results are given that demonstrate the effectiveness of the described techniques.

1 Introduction

Branch target buffers (BTBs) play an important role in many pipelined processors in which branch prediction is employed to provide a steady flow of instructions in the presence of branch instructions [1]. A BTB is a cache in which branch targets are stored. In its most elementary form, a BTB is a table indexed by the lower part of the pc (program counter) with entries consisting of a tag and a branch target. In the fetch stage of the pipeline the BTB is indexed by the lower part of the pc. The tag of the returned entry is compared with the remaining bits of the pc. If these pc bits match, the instruction that is fetched from the instruction cache will be a branch instruction. In that case the target returned by the BTB will be assigned to the pc so that instructions will be fetched from that address in the next cycle. This scheme works fairly well, since most branches are taken (65-70% on average) and branches are stored in the BTB when they are taken.

A first improvement on this scheme is to make the BTB set-associative to reduce conflicts between branches mapping onto the same entry. Instead of indexing one entry from the BTB, a set of n entries is indexed and n parallel tag compares determine the presence of a branch and, if present, which entry it contains.

A further improvement is to predict the direction, i.e. taken or not taken, of a branch not by the presence of it in the BTB but in a more sophisticated way. A common method is to include a two bit saturating counter in a BTB entry [2]. This counter is incremented on a taken branch and decremented on a not-taken branch. A branch will be predicted to be taken if its corresponding counter has its highest bit set. Very often these counters are decoupled from the BTB and stored in another table which has typically more entries than the BTB and is often accessed differently to exploit branch correlation [3, 4, 5].

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 950–959, 2000.

Another improvement is to employ a return address stack (RAS) [6]. Function return branches are indirect jumps with varying targets. This makes that the target stored in the BTB, which is simply the target of the last execution of the branch, is a worse target predictor. A RAS can improve this by letting function call branches push the return address on it and letting function return branches pop values of it and use them as target prediction. For this to work, the fetch stage must know whether branches are function calls, function returns, or neither. This sort of type information can be stored in the BTB. If the BTB reports that the fetched instruction is a function call it pushes the address of the next sequential instruction on the RAS. If the BTB reports that the fetched instruction is a function return it pops a value of the RAS and uses it for target prediction.

The objective of the work described in this paper is to improve the cost-efficiency of BTBs by using fewer bits per BTB entry. This will make BTBs smaller and reduce power dissipation. The latter can be quite high since a BTB is a table of typically a few kbyte in size that is accessed every cycle[1]. Alternatively, with the same area and power budgets one can make a BTB that delivers more performance with the techniques described in this paper.

We obtain a reduction in storage requirements by making the tag and target fields of a BTB entry smaller. Both reductions are shown in Fig. 1. Reduction of tag fields is known as partial resolution and has been proposed by Fagin and Russell [8]. The main contribution of this paper is target field size reduction. We make use of branch locality, i.e., most branch distances are short. Two schemes are presented that exploit branch locality with comparable performance. In one scheme BTB entries can store a short branch and two entries are required to store a long branch, which occurs less frequently. In the other scheme only one entry per set of a set-associative is able to store long branches. The other entries can only store short branches and are therefore smaller.

The remainder of the paper is organised as follows. Section 2 describes the simulation environment we used for our experiments. Section 3 discusses partial resolution. Section 4 presents the two BTB organisations that exploit branch locality. Section 5 concludes the paper.

2 Simulation Environment

We used the SimpleScalar v2.0 tool set for our experiments in which we modified the BTB simulation code [9]. SimpleScalar resembles the 32-bit MIPS architecture. We used a decoupled system in which the BTB detects branches and provides branch targets and type information, and a separate predictor provides direction predictions. To expose the effect of different BTB organisations, we use a fairly aggressive hybrid branch predictor consisting of a 4096 entry gshare predictor with 12 bits history, a 2048 entry bimodal predictor, and a 2048 entry meta predictor [3, 4, 5]. We used a 32 entry RAS. The BTBs used are 4-way set-associative with LRU replacement with sizes varying from 32 to 1024 entries.

[1] According to Musoll [7], the Intel Pentium Pro dissipates 5% of its total power dissipation in its 512-entry BTB. This percentage becomes higher for a less aggressive processor or a processor with less complex decoding. The percentage will also be higher for 64-bit processors.

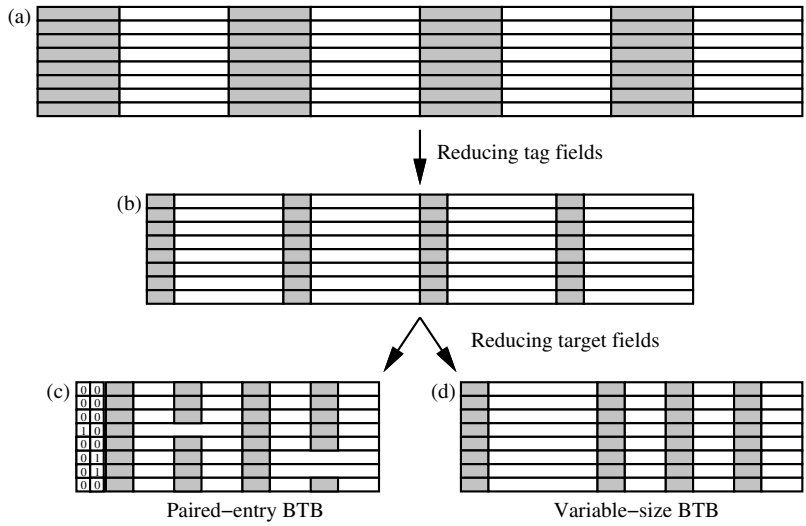

Fig. 1. Organisations of four types of BTBs. The gray parts represent tag fields while the white fields represent target and type information fields. The picture is approximately on scale. (a) shows a traditional 4-way set-associative BTB with 32 entries. (b) shows a BTB with partial resolution in which tag fields have been reduced. (c) and (d) show two organisations in which branch locality is exploited. (c) shows a paired-entry BTB (PE-BTB) in which long branches occupy two BTB entries. (d) shows a variable-size BTB (VS-BTB) in which only one BTB entry per set can store a long branch.

As a benchmark set we used the eight benchmarks from SPECint95. Simulation was limited to 100 million instructions.

3 Partial Resolution

BTBs with partial resolution do not store all the address bits that are not used for indexing in tags [8]. This reduces both the number of storage bits as well as the size of the comparators that perform the tag compare. The result of partial resolution is so-called false-hits; the BTB hits for an instruction which is not a branch instruction or is a branch instruction that does not correspond to the information returned by the BTB. If the branch predictor predicts taken, then the control flow is very likely to be directed in the wrong direction. This will be detected and corrected in a later pipeline stage. The effect is a cycle penalty comparable to the misprediction penalty. How often this occurs will depend on how many tag bits are used. Fagin and Russell conclude that 2 tag bits are necessary to obtain 99.9% of the accuracy of full resolution and 3 tag bits for 99.99%.

Our experiments however showed that more tag bits are required. Figure 2 shows the results of these experiments. We used two benchmarks with a large branch working set, *gcc* and *go*, and two benchmarks with a small branch working set, *compress* and

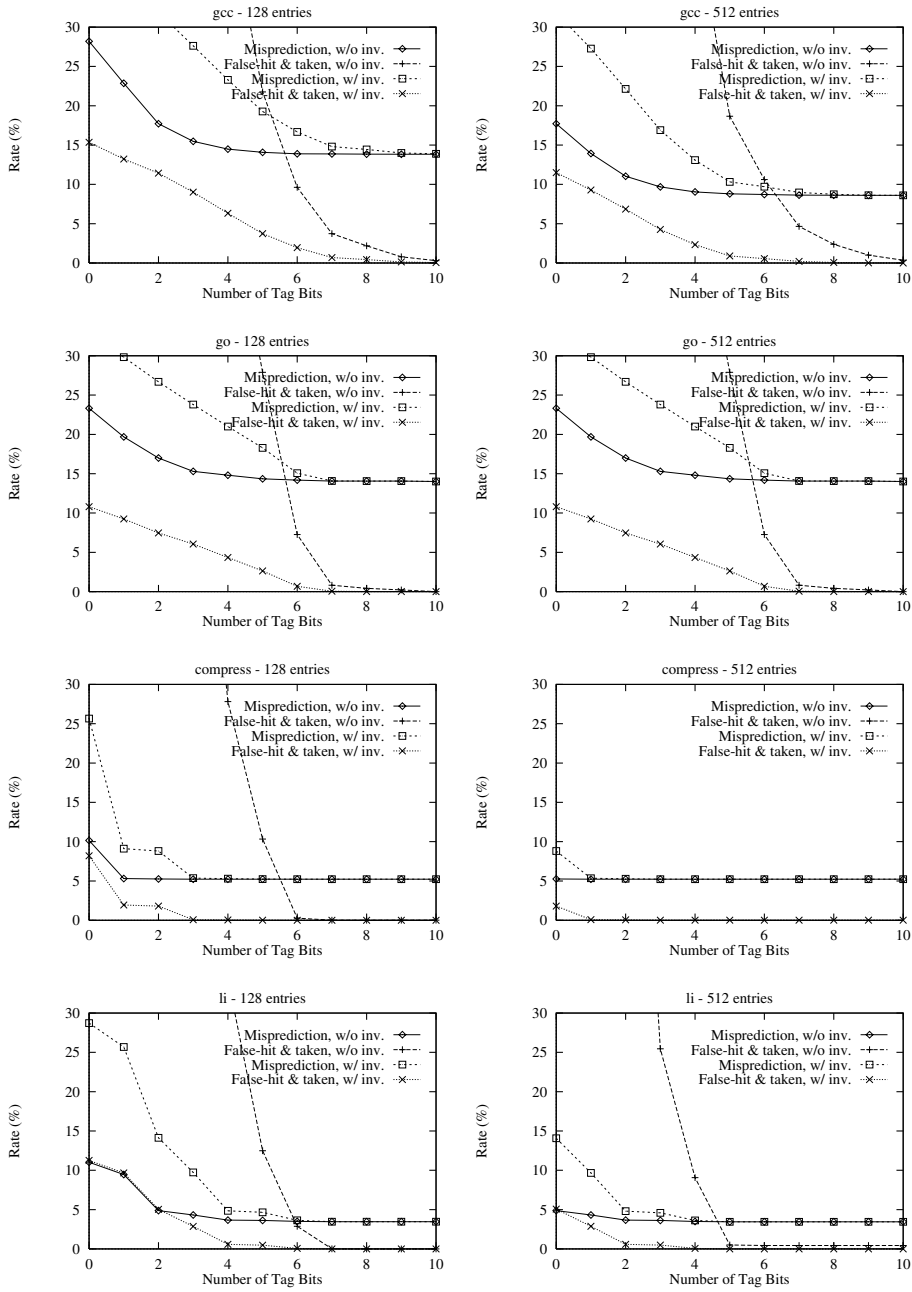

Fig. 2. Misprediction and false-hit rates for *gcc, go, compress*, and *li* for 128 and 512 entry BTBs as a function of the number of tag bits. The false-hit taken rate of *compress* for 512 entries without invalidation is so high (more than 430%) that it lies beyond the vertical range of the graph.

li. We used BTBs of 128 and 512 entries. Two policies were used to handle false-hits: invalidation and no invalidation. With invalidation a BTB entry that causes a false-hit is invalidated to prevent it continuing to cause false-hits[2]. This is achieved by a special type information value in the type information field of the BTB entry. Alternatively, one can write a random value in the tag field. The graphs of Fig. 2 show two values: the misprediction rate and the false-hit taken rate. The latter is the ratio of the number of non-branch instructions that cause a false-hit that are predicted taken relative to the number of branch instructions. Obviously, this value can be more than one when only a few tag bits are used.

Several conclusions can be drawn from the results shown in Fig. 2. First, both the misprediction and the false-hit taken rate decrease as the number of tag bits increases. More tag bits reduces the probability of branches being mixed up with other branches (which decreases the misprediction rate) and that of branches being mixed up with non-branch instructions (which decreases the false-hit taken rate). Invalidation reduces the false-hit taken rate significantly but increases the misprediction rate because BTB entries of branches are invalidated. Because the decrease in false-hit taken rate is much higher than the increase in misprediction rate, it is a good design decision to invalidate BTB entries that cause false-hits.

The number of tag bits required for a misprediction rate close to the misprediction rate for full resolution and a false-hit taken rate close to zero depends clearly on the branch working set size of the application. For the small branch working sets of *compress* and *li*, one to four tag bits will be sufficient. For the larger working sets of *gcc* and *go*, seven to eight tag bits will be required. This is significantly more than the two to three tag bits advised by Fagin and Russell. Because *gcc* and *go* are more representative of real-world applications than *compress* and *li*, we propose to use at least eight tag bits. Note that BTBs with fewer entries will have a higher false-hit rate and will therefore need more tag bits to reduce this to close to zero.

4 Exploiting Branch Locality

The tag and target fields are the largest fields of a BTB entry. Partial resolution shortens the length of tag fields. In this section we show how target fields can be shortened by making use of branch locality, i.e., the property of most programs that most branch distances are short. Branch locality is already being used for pc-relative branches in most architectures to reduce code size. Storing pc-relative offsets in the BTB is likely to be infeasible since a BTB lookup and addition would have to be performed sequentially in one clock cycle. A solution is to use page-relative addressing. Short branches are branches in which only the lowest n significant[3] pc bits change. The other branches are long branches. The idea is to choose a small value for n such that most branches are short branches and to develop a BTB such that short branches are stored in fewer

[2] Fagin and Russell do not mention invalidation in their paper [8]. It is therefore not clear whether they apply this.

[3] In many architectures instructions are aligned on 2^m byte addresses. In that case the lowest m bits are always zero and therefore not significant and do not have to be stored in the BTB.

bits than longer branches. First we determine a value for n. Figure 3 shows the percentage of short branches as a function of the number of addressing bits for pc-relative and page-relative addressing for *gcc*. Function return branches were excluded in this measurement since their targets are stored in the RAS and not in the BTB. The measurement shows that pc-relative addressing would be a little bit more effective for our purpose than page-relative addressing. The reason is that a short-distance branch across a boundary that is a high power of two is a long branch. Nevertheless, pc-relative addressing is not applicable for the reason mentioned above. The graph of Fig. 3 starts to level off at $n = 10$, at which 86% of the branches are short branches. Therefore, $n = 10$ will be a suitable value.

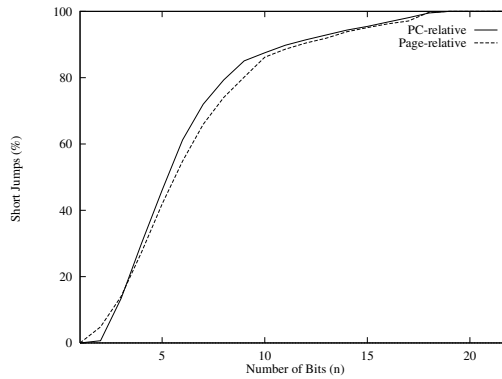

Fig. 3. Percentage of short branches as a function of the number of offset bits n. In pc-relative addressing, a branch from a to b is a short branch if $-2^{n-1} \leq b - a < 2^{n-1}$. In page relative addressing, a branch from a to b is a short branch if only the n lowest significant bits of a and b differ in value.

4.1 Paired-Entry and Variable-Size BTBs

We propose two BTB organisations for exploiting branch locality: paired-entry and variable-size BTBs.

—A *paired-entry BTB (PE-BTB)* is a set-associative BTB in with which an entry can contain one short branch and a pair of two entries from one set is needed to store a long branch. For efficient implementation, entry pairs are required to be adjacent and aligned in the set. A *pair-bit* is required per entry pair to determine whether entries are paired. For optimal resource utilisation, the total storage requirements of two short branches should be equal to the storage requirements of one long branch. This limits the range of possible tag and target sizes; the sum of 2 tags and 2 short targets should be equal to 1 tag and 1 long target (ignoring type information fields). To obtain better matching, one can decide to use different tag sizes for long and short branches or for different ways.

—A *variable-size BTB (VS-BTB)* is a set-associative BTB with which some part of the entries of each set can store both long and short branches while the remaining set entries can store only short branches.

Using the target information from the BTB is straightforward. The lookup returns on a hit whether a short or a long branch is hit. In the case of a short branch and a taken prediction, only the n lowest significant bits of the pc are updated; the upper bits retain their value. In the case of a PE-BTB, the pair-bits are used in the tag comparison. If a long branch is stored in two BTB entries, the tag compare of the entry whose tag bits are used to store the long target has to be disabled.

Some design options exist in updating the BTB. In the case of updating a PE-BTB with a long branch: (1) one can victimise the pair containing the least recently used entry, or (2) one can victimise the pair which has been least recently used, where a pair is used when one of its entries is used. To understand the difference, consider the following set of four entries showing the latest access times ($T_1 < T_2 < T_3 < T_4$):

T_1	T_4	T_2	T_3

In the case of the first alternative, the first pair would be victimised (because it contains the first entry which is the least recently used entry), whereas in the case of the second alternative the second pair would be victimised (because this pair was used least recently). We opted for the first alternative because of its simplicity and because the differences in prediction accuracy are negligible. For VS-BTBs one has to decide whether short branches are allowed in long entries or whether long entries are reserved exclusively for long branches. We chose the first alternative, which yields a slightly better prediction accuracy.

4.2 Evaluation

To evaluate the cost-efficiency of the discussed techniques we defined the four BTB types that in Table 1. All types are 4-way set-associative and use LRU replacement. In the entry size calculations we neglected the bits for the LRU administration, which are the same for all types. In the case of partial resolution, we used 8 bit tags. For short branches we used 10 bit page-relative addressing. We assumed a 32-bit RISC architecture in which instructions were 32-bit aligned, so absolute addresses were 30 bits long. Two bits of type information were used per entry, which is sufficient to distinguish four cases: (1) function call branches, (2) function return branches, (3) invalidated entries, and (4) the normal case. We varied the number of entries between 32 and 1024 in powers of two ($2^n, 5 \leq n \leq 10$).

Figure 4 shows BTB size vs. prediction accuracy curves for the four BTB types for the eight SPECint95 benchmarks. Figure 5 shows the average of the eight benchmarks. The curves obtained for the benchmarks corresponding to the four BTB types run in parallel until the BTB reaches a size at which it no longer constrains the prediction accuracy. The distances between the curves indicate the improved cost-efficiency of partial resolution, VS-BTBs, and PE-BTBs. The point at which the curves meet depends on the branch working set size of the benchmark. This point shifts to the right as the

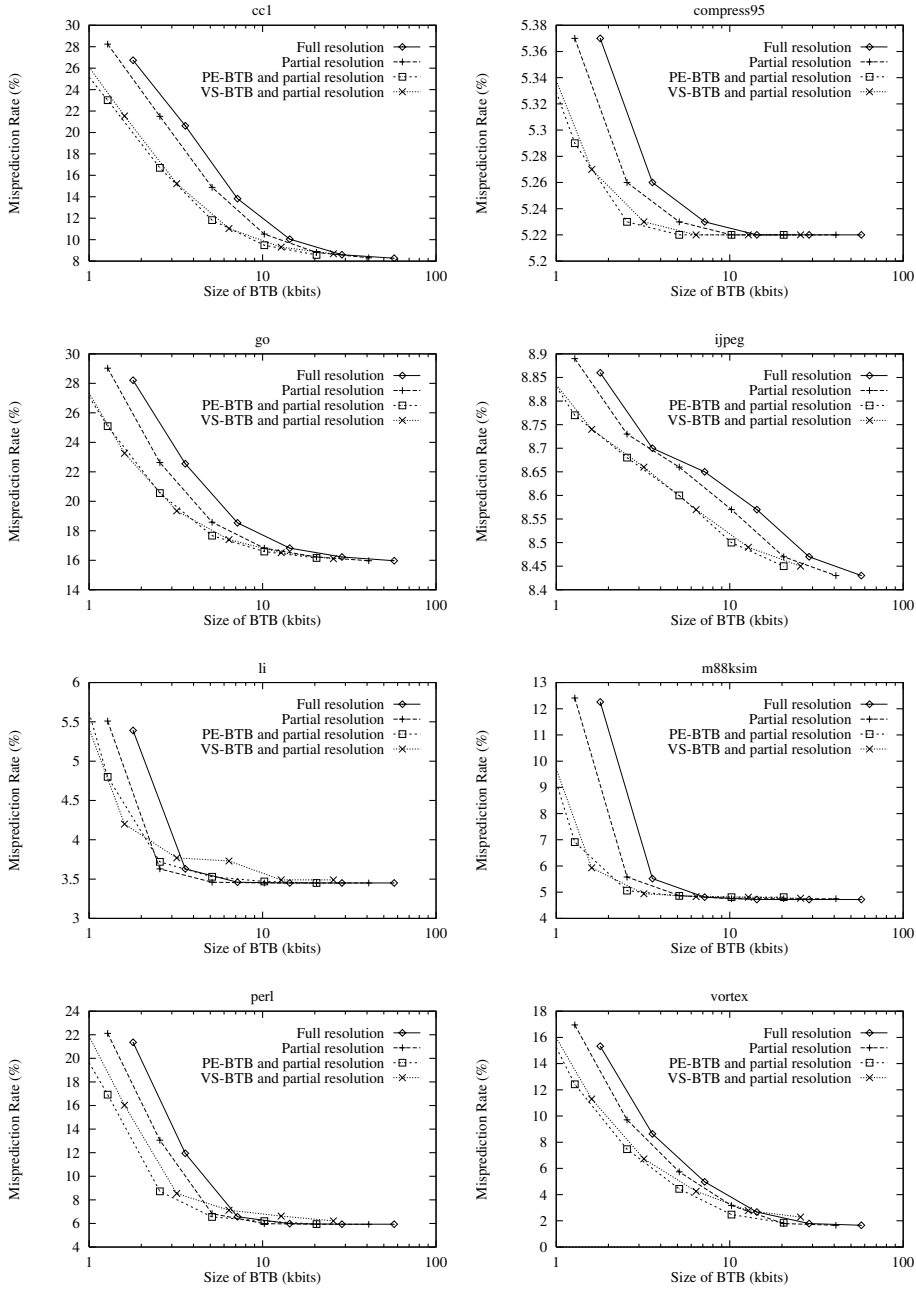

Fig. 4. BTB size vs. mispredict rate for the BTB types listed in Table 1 and shown in Fig. 1.

Type	Resolution	Target size	Type info size	Entry size	Fig. 1
Traditional	Full – 23-27b	4×30	2	220-236	(a)
Traditional	Partial – 8b	4×30	2	160	(b)
PE-BTB	Partial – 8b	$2 \times 30/4 \times 10/1 \times 30 + 2 \times 10$	2	82	(d)
VS-BTB	Partial – 8b	$4 \times 10/1 \times 30 + 3 \times 10$	2	100	(c)

Table 1. Four types of BTBs corresponding to the types shown in Fig. 1.

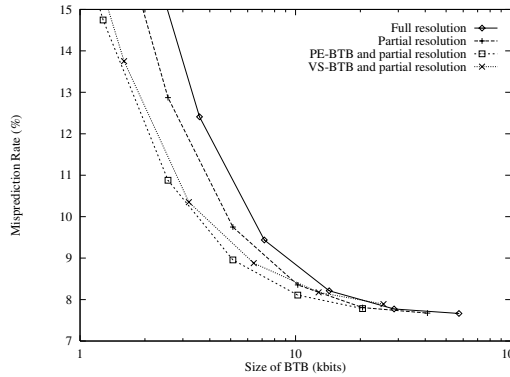

Fig. 5. BTB size vs. average mispredict rate for the BTB types listed in Table 1.

benchmarks become larger, and therefore more realistic. It will also shift to the right in a multi-tasking environment which puts more pressure on branch prediction resources.

The results show that both VS-BTBs and PE-BTBs have a better cost-efficiency than traditional BTBs, with PE-BTBs performing a little bit better than VS-BTBs. In the case of *li*, *perl*, and *vortex*, VS-BTB is performing worse than the other organisations for a large number of entries. The reason is conflicts between long branches that can only be mapped in one entry per set. This is clearly a disadvantage of VS-BTBs.

4.3 Variations

Several variations are possible on the VS-BTB and PE-BTB described above. First, one can use more than two target sizes. For example, 6 bits, 12 bits, and 30 bits. This could improve the utilisation of target fields.

PE-BTBs can be extended to three or more entries for long branches. This could be useful to match the total storage space of multiple short branches with the storage space of one long branch, especially when target size of short branches are very small, e.g., 6 bits, or the target size of a long branch is very large, e.g., 62 bits in a 64-bit architecture.

In the case of VS-BTBs, one can use several memories with a different number of entries for different branch target sizes. For example, one could use a table of 128 entries with 30 bit target fields together with a table of 512 entries with 10 bit target fields. The combination is a two-way set-associative BTB with the two ways having different numbers of entries.

5 Conclusions

Two methods for reducing the size and power dissipation of BTBs have been described: partial resolution and exploitation of branch locality. Partial resolution has been proposed before while exploiting branch locality is a contribution of this work. Partial resolution reduces the number of tag bits. This reduces the size of a BTB at the cost of a slightly higher misprediction rate due to false-hits. Two BTB organisations have been proposed for exploiting branch locality. In both organisations a distinction is made between short branches and long branches, with short branches being branches with which a certain number of the highest pc bits do not change when the branch is taken. In the VS-BTB organisation, one or more entries of a set (both proposed organisations assume set-associativity) can store both long and short branches while the others can store only short branches. In the PE-BTB organisation, entries can store only a short branch and two entries are required to store a long branch. These paired entries are adjacent and aligned for simplicity. The effectiveness of partial resolution, VS-BTBs, and PE-BTBs has been demonstrated by means of simulations. The results show a clear improvement in cost-efficiency for 32-bit architectures, with PE-BTBs being sightly more efficient than VS-BTBs. For 64-bit architectures, the improvement that can be realized with the described techniques will be even greater.

References

[1] Johnny K. F. Lee and Alan Jay Smith. Branch Prediction Strategies and Branch Target Buffer Design. *IEEE Micro*, 21(7):6–22, January 1984.

[2] James E. Smith. A Study of Branch Prediction Strategies. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 135–147, May 1981.

[3] Shien-Tai Pan, Kimming So, and Joseph T. Rahmeh. Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 76–84, Boston, Massachusetts, October 12–15, 1992.

[4] Tse-Yu Yeh and Yale N. Patt. Two-Level Adaptive Training Branch Prediction. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pages 51–61, Albuquerque, New Mexico, November 18–20, 1991.

[5] Scott McFarling. Combining Branch Predictors. Technical Report TN-36, Western Research Laboratory, Palo Alto, California, June 1993.

[6] David R. Kaeli and Philip G. Emma. Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 34–42, Toronto, Ontario, May 27–30, 1991.

[7] Enric Musoll. Predicting the Usefulness of a Block Result: A Micro-Architectural Technique for High-Performance Low-Power Processors. In *Proceedings of the 32th Annual International Symposium on Microarchitecture*, pages 238–247, Haifa, Israel, November 16–18 1999.

[8] Barry Fagin and Kathryn Russell. Partial Resolution in Branch Target Buffers. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 193–198, Ann Arbor, Michigan, November 29–December 1, 1995.

[9] Doug Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report 1342, University of Wisconsin-Madison, Computer Sciences Department, June 1997.