Simultaneous Way-footprint Prediction and Branch Prediction for Energy Savings in Set-associative Instruction Caches

Weiyu Tang Rajesh Gupta Alexandru Nicolau
Alexander V. Veidenbaum
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425
{wtang,rgupta,nicolau,alexv}@ics.uci.edu

Abstract

Caches are partitioned into subarrays for optimal timing. In a set-associative cache, if the way holding the data is known before an access, only subarrays for that way need to be accessed. Reduction in cache switching activities results in energy savings.

In this paper, we propose to extend the branch prediction framework to enable way-footprint prediction. The next fetch address and its way-footprint are predicted simultaneously for one-way instruction cache access. Because the way-footprint prediction shares some prediction hardware with the branch prediction, additional hardware cost is small.

To enlarge the number of one-way cache accesses, we have made modifications to the branch prediction. Specifically, we have investigated three BTB allocation policies. Each policy results in average 29%, 33% and 62% energy savings with normalized execution time 1, 1, and 1.001 respectively.

1 Introduction

With advances in semiconductor technology, processor performance continues to grow with increasing clock rates and additional hardware support for instruction level parallelism. The side effect is that power dissipation also increases significantly. With the maturity of IC techniques for power management, architectural and compiler techniques hold significant potential for power management [2]. These techniques decrease power dissipation by reducing the number of signal switching activities within the microprocessor.

High utilization of the instruction memory hierarchy is needed to exploit instruction level parallelism. Thus power dissipation by the on-chip instruction cache is also high. On-chip L1 instruction cache alone can comprise as high as 27% of the CPU power[7].

In this paper, we exploit cache way partitioning in the set-associative caches for instruction cache energy savings. For a cache access, if the way holding the instructions is known before the access, then only that particular way needs to be accessed. To know which way holds the instructions before an access, a way-footprint prediction mechanism can be used. There are similarities between the branch prediction and the way-footprint prediction. One predicts the next fetch address based on current fetch address; the other predicts the way-footprint of the next fetch address based on current fetch address. Thus we can extend the branch prediction framework to enable way-footprint prediction, which can significantly reduce the hardware cost for the way-footprint prediction.

The rest of this paper is organized as follows. In Section 2, we present the motivation for this research. Section 3 describes the implementation of way-footprint prediction. The experimental results are given in Section 4. Section 5 compares way-footprint prediction with related techniques for instruction cache energy savings. The paper is concluded with future work in Section 6.

2 Motivation

For optimal timing, caches are partitioned into several subarrays so that wordline and bitline lengths are short. In high-performance processors, all the data subarrays and tag subarrays in a set-associative cache are accessed in parallel to achieve short access time. If the cache way holding the data is known before an access, only data subarrays and tag subarrays for that particular way need to be accessed. This reduces per cache access switching activities and hence results in energy savings.

One approach to predict the way-footprint for an address is to use the way-footprint of this address when it was accessed last time. For this purpose, his-

tory way-footprints should be saved. A simple implementation is to use a way-footprint cache. Each entry in the way-footprint cache is in the following format:

(addr_tag, way-footprint).

The size of the way-footprint field is equal to $\log(n+1)$, where n values are needed for one-way access in a n-way set-associative cache and one value is needed for all-way access. For a 4-way set-associative cache, the size of the way-footprint field is 3 bits. For a 2k-entry way-footprint cache and 4-byte instruction size, the size of addr_tag field is 21 bits. The tag (addr_tag) cost is much higher than the data (way-footprint) cost in terms of area and power.

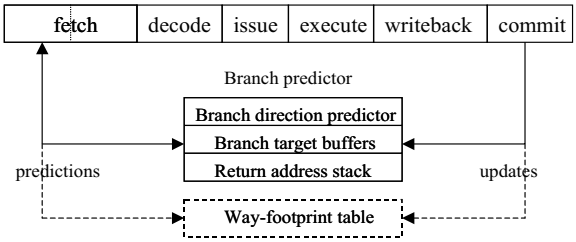

Figure 1: Pipeline architecture

To support instruction fetch across basic block boundaries, branch prediction is used in modern processors. Figure 1 shows a typical pipeline architecture with a branch predictor. For high-performance processors, multiple instructions are fetched simultaneously and it may take 2 or 3 cycles to access the instruction cache. Whether an instruction is a branch can only be determined a few stages later in the pipeline. If the branch predictor only uses a branch address to predict the next fetch address, there will be bubbles in the pipeline for instruction fetch or the branch miss prediction rate will be high. Thus in processors such as G5 [5], the branch predictor uses current fetch address to predict the next fetch address every cycle.

Generally, there are three components in a branch predictor: **b**ranch **d**irection **p**redictor (BDP), **b**ranch **t**arget **b**uffers (BTB) and **r**eturn **a**ddress **s**tack (RAS). The BDP predicts whether a branch will take the target path. The BTB predicts the target address for a taken branch. The RAS predicts the return address for a return instruction.

A BTB is organized as a RAM-based structure and is indexed by the fetch address. Each entry in the BTB is in the following format:

(addr_tag, target address).

A RAS is organized as a stack and only the top entry is accessed. Each entry in the RAS is in the following format:

(return address).

Note that the same fetch address is used in both branch prediction and way-footprint prediction. If the tag comparison in the BTB fails, then the tag comparison in the way-footprint cache will also fail. Thus the tag used in the way-footprint cache is redundant and can be eliminated to reduce hardware cost.

3 Way-footprint Prediction

To support way-footprint prediction, a way-footprint field is added to the RAS entry. As the number of entries in a RAS is small and only the top entry is accessed during the branch prediction, the RAS access is not on one of the critical path. Consequently, adding the way-footprint field to the RAS entry is unlikely to affect the processor cycle time.

Adding way-footprint fields to the BTB entry will increase the BTB capacity. The BTB access time increases with capacity. This may affect the processor cycle time because the BTB access is often on one of the critical path. Thus a separate **W**ay-**F**ootprint **T**able (WFT) shown in Figure 1 is used instead. The number of ways and the number of sets in the WFT is equal to those in the BTB. Each entry in the WFT has the following two way-footprint fields:

- target address way-footprint
- fall-through address way-footprint

The WFT access time is shorter than that of the BTB because the WFT capacity is much smaller than the BTB capacity. Thus the WFT access is not on one of the critical path.

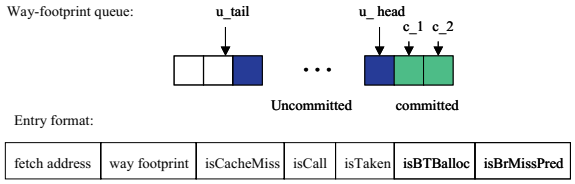

Figure 2: Way-footprint queue

Figure 2 shows the way-footprint queue needed for the WFT and the RAS update. Entries "c_1" and "c_2" are reserved for the last two committed fetch addresses. Entries from "u_head" to "u_tail" are used to keep track of the way-footprints for uncommitted fetch addresses.

When an instruction fetch finishes, the "fetch address" and "way-footprint" fields of entry "u_tail" are updated. The "isCacheMiss" field is set if this fetch has generated a cache miss.

When an instruction commits and its address matches the "fetch address" field of entry "u_head", the following fields of entry "u_head" are updated:

- "isCall" is set if it is a call instruction;
- "isBTBalloc" is set if a BTB entry is allocated for the instruction;
- "isBrMissPred" is set if the instruction is a miss predicted branch;
- "isTaken" is set if the instruction is a taken branch.

Then pointers "u_head", "c_1" and "c_2" are updated to reflect the fact that a new entry has committed. If the committed instruction is a miss predicted branch, a wrong path is taken and the way-footprints in the uncommitted entries of the way-footprint queue are useless. All the uncommitted entries are flushed. Note that this flush is done in parallel with the pipeline flush, which is required on a branch prediction miss. Other queue operations are simple. Thus queue operations are unlikely on one of the critical path.

The WFT is updated in the next cycle if one of the following conditions is satisfied:

- "isCacheMiss" of entry "u_1" is set; the way-footprint for a fetch address may change on an instruction cache miss and WFT update is necessary;
- "isBrMissPred" of entry "u_2" is set; the way-footprint for the next fetch may also change on a branch prediction miss because a different control path may be taken;
- "isBTBalloc" of entry "u_2" is set; an entry will also be allocated in the WFT so that both the target address and the way-footprint can be provided next time the same fetch address is encountered.

The "fetch address" field of entry "u_2" and the "way-footprint" field of entry "u_1" are used to update the WFT. Either the "target address way-footprint" field or the "fall-through address way-footprint" field is updated depending on whether the "isTaken" field of entry "u_2" is set.

Entries in both the BTB and the WFT can be identified using (way, set). During the branch prediction, the BTB and the WFT are accessed in parallel using the same index function. If the fetch address matches the tag of the BTB entry (w, s), then either the "target address way-footprint" or the "fall-through address way-footprint" of the WFT entry (w, s) is provided for the next fetch depending on the branch direction predicted by the BDP.

If the "isCall" field of entry "u_2" is set, the RAS update is needed. As the next fetch address is not the return address, the "way-footprint" in entry "u_1" is useless. However, if the call instruction is not on the cache line boundary, the instruction following the call instruction, which will be executed once the call returns, is also in the same cache line, Thus the way-footprint for the call instruction can be used for the return address if the call instruction is not on the cache line boundary. Otherwise, the way-footprint for all-way access will be used.

During the branch prediction, if a return instruction is predicted to be in the current fetch, then the top entry of the RAS will provide both the return address and the way-footprint for the next fetch.

Modifications to the BTB allocation policy can affect the BTB hit rate, which in turn can affect the number of successful way-footprint predictions because way-footprint prediction can succeed only when the tag comparison in the BTB succeeds. We have investigated the following three BTB allocation policies:

- **t**aken **b**ranch policy (TB): BTB allocation only for a taken branch missing from the BTB;
- **a**ny **b**ranch policy (AB): BTB allocation for any branch missing from the BTB;
- **a**ny **f**etch **a**ddress policy (AFA): BTB allocation for any fetch address missing from the BTB.

When an untaken branch or a non-branch instruction is allocated a BTB entry, the target address is the next continuous fetch address, which is the default address prediction if current fetch address misses from the BTB. The AB and AFA policies can decrease the number of entries available for taken branches and may degrade performance. Thus the TB policy is used in most processors.

4 Performance

We use the SimpleScalar toolset [3] to model an out-of-order speculative processor with a two-level cache hierarchy. The simulation parameters shown in Table 1 roughly correspond to those in a high-performance microprocessor. We have simulated 100 million instructions for all SPEC95 benchmarks except Vortex.

Parameter	Value
branch pred.	combined, 4K 2-bit chooser, 4k-entry bimodal, 12-bit, 4K-entry global 7-cycle miss prediction penalty
BTB	2K-entry, 4-way
return address stack	32
RUU/LSQ	64/32
fetch queue	16
fetch width	8
int./flt. ALUs	4/2
int./flt. Mult/Div	2/2
L1 Icache	32KB, 4-way, 32B block
L1 Dcache	64KB, 4-way, 32B block
L2 cache	512KB, 4-way, 64B block

Table 1: System configuration

For the 4-way set-associative instruction cache, we use the Cacti [11] to obtain the cache partitioning parameters with the optimal timing. The data array is partitioned into eight subarrays and the tag array is partitioned into two subarrays. One-way cache access needs only to access two data subarrays and one tag subarray. We also use the Cacti to derive the power parameters. The power per all-way access is normalized to 1. The power per one-way access is 0.2896 and the power per WFT access is 0.054.

The RAS and the way-footprint queue are small structures and the power dissipation by them is very small comparing to that of the instruction cache. Thus the power dissipation by them is not modeled.

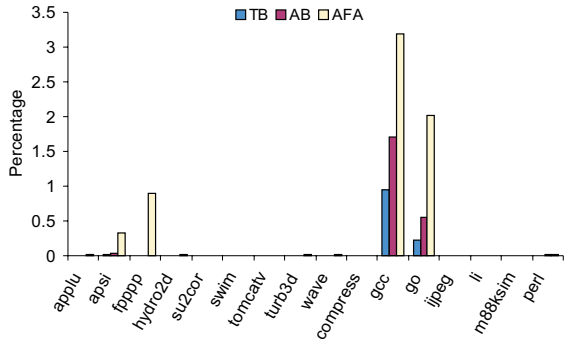

Figure 3: BTB allocation rate

Figure 3 shows the BTB allocation rate, calculated as total number of BTB allocations versus total number of instruction fetches. The BTB allocation rate is close to 0 for most benchmarks. For those benchmarks, once a fetch address is allocated an entry, it is unlikely to be replaced from the BTB because the BTB capacity is much larger than the work set size. Noticeable increase in the allocation rate can be found in *apsi*, *fpppp*, *gcc* and *gcc*. For these benchmarks, the work set size is relatively large and the number of BTB conflict misses increases, which leads to more number of BTB allocation.

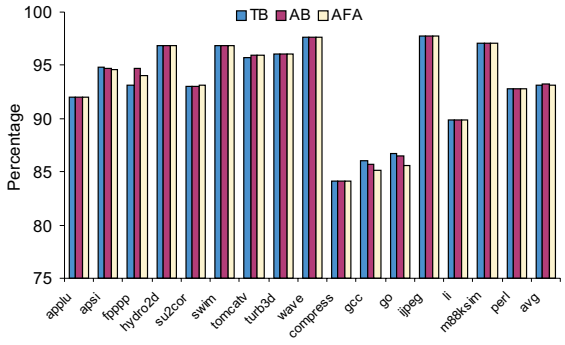

Figure 4: Branch address prediction hit rate

Figure 4 shows branch address prediction hit rate. For most benchmarks, there is virtually no difference in the hit rate with different BTB allocation policies. For *gcc* and *go*, the hit rate with the AB and AFA policies is lower than that of the TB policy. The reason is that untaken branches and non-branch instructions are allocated BTB entries. As a consequence, the effective number of BTB entries for the taken branches with the AB and AFA policies is smaller than that of the TB policy.

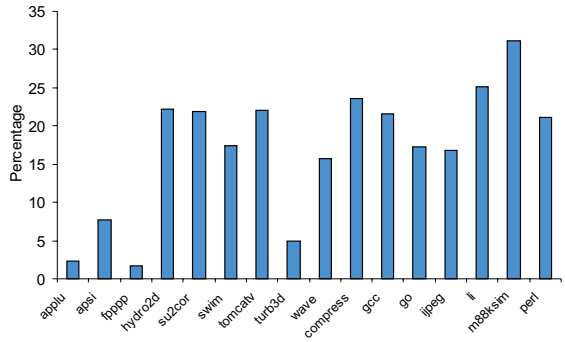

Figure 5: Dynamic branch instruction rate

However, a couple of benchmarks such as *fpppp*

show slight increase in the hit rate. The branch history is updated if an address has an entry in the BTB. The history update can somehow improve the prediction accuracy for other correlated branches.

Figure 5 shows the dynamic branch instruction rate. The dynamic branch instruction rate varies widely. The rate ranges from 15% to 35% for 12 benchmarks. For four float-point benchmarks–*applu*, *apsi*, *fpppp* and *turb3d*, the rate is lower than 10%.

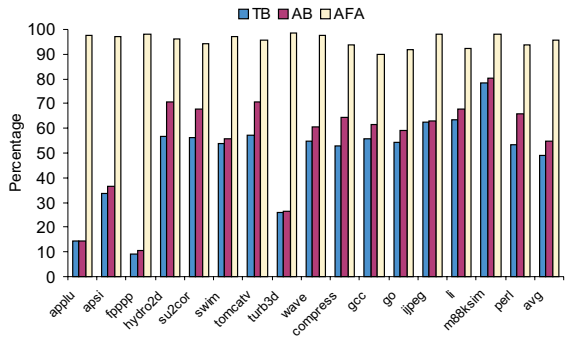

Figure 6: One-way cache access rate

Figure 6 shows percentage of instruction fetches that need only one-way access. For the AFA policy, a BTB entry is allocated for every fetch address missing from the BTB. Thus one-way cache access rate is close to 100% for every benchmark and is not affected by the dynamic instruction rate shown in Figure 5. However, high dynamic instruction rate results in high one-way access rate for the TB and AB policies. One-way access rate for the AB policy is slightly higher than the rate for the TB policy because of additional entries allocated for untaken branches.

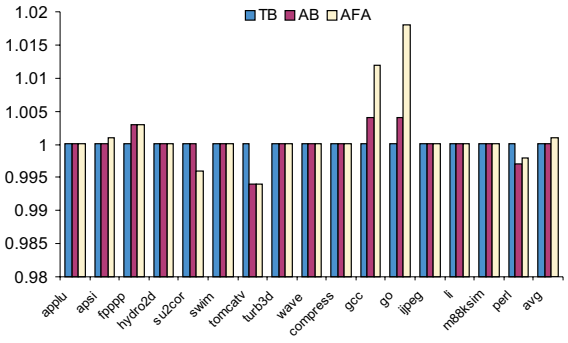

Figure 7: Normalized execution time

Figure 7 shows normalized execution time. For most benchmarks, the execution time is almost same with different policies. For *gcc* and *go*, the execution time increases because the branch address prediction hit rate decreases as shown in Figure 4. For *su2cor* and *tomcatv*, increase in branch address prediction hit rate results in decrease in execution time. For *fpppp*, although the address prediction hit rate increases, the overall instruction cache miss rate increases as well. Hence the execution time increases slightly. The average normalized execution time is 1, 1 and 1.001 for TB, AB and AFA respectively. There is virtually no performance degradation with our technique.

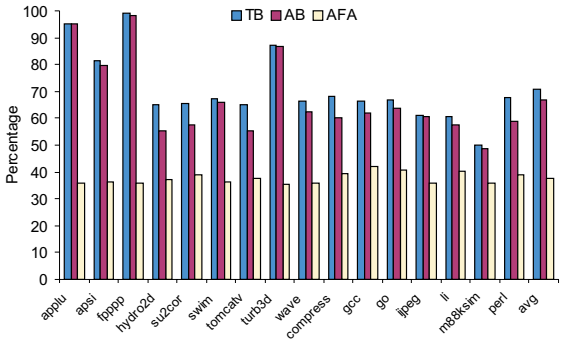

Figure 8: Normalized energy

Figure 8 shows normalized energy. As our technique can only reduce instruction cache hit energy, hit energy is used in the calculation. The relationship between the hit energy and the miss energy depends on the instruction cache miss rate. For all benchmarks except *fpppp*, the hit energy is at least ten times the miss energy. Normalized energy highly depends on the one-way cache access rate shown in Figure 6. For TB, AB and AFA policies, the average normalized energy is 70.8%, 66.7% and 37.6% respectively, which translates into 29.2%, 33.3% and 62.4% energy savings.

5 Discussion

Calder and Grunwald have proposed a coupled BTB for fast instruction fetch in a set-associative instruction cache [4]. A similar scheme is also used in the Alpha 21264 [8]. Each cache line is in the format:

(tag, data, next-line, next-way).

"Next-line" and "next-way" are used to locate the next fetch cache line. Because of the "next-way" prediction, most of the time one cache way is accessed and this can result in low instruction cache energy.

The effective number of entries in a coupled BTB is smaller than the number of instruction cache lines. In

addition, "next-line" and "next-way" can only point to one cache line. Prediction misses often occur when there are multiple branches in a cache line or a branch changes direction. On the contrary, the decoupled BTB shown in Figure 1 can provide accurate prediction in the above scenarios.

In the coupled BTB, the next fetch cache line is unknown until current instruction fetch finishes. The instruction fetch is serialized and is not scalable. On the other hand, the decoupled BTB is scalable. It can support multiple branch predictions and multiple cache line fetches in one cycle to deliver more instructions [10, 12]. The decoupled BTB can also enable a scalable front-end with asynchronous instruction fetch and BTB prediction for high rate instruction delivery as proposed by Reinman, Austin and Calder [9].

Comparing to a coupled BTB, the energy savings by our way-footprint prediction are higher because the prediction accuracy by the decoupled BTB is higher and we can predict based on RAS. And the framework for energy savings by our way-footprint prediction is scalable.

Inoue, Ishihara and Murakarni have proposed another kind of "way-prediction" [6]. For each cache set, the way-footprint for the last accessed way is stored in a table. When the same set is accessed next time, the last accessed way is speculatively accessed first. On a way-prediction miss, the remaining ways are accessed in the next cycle. Way-prediction is stored in a table and this table is accessed before the cache. Because of this kind of access serialization, the processor cycle time may be affected. In addition, the performance degradation is much higher than our approach.

Albonesi has proposed "selective cache ways" [1] to turn off some cache ways based on application requirements. He has only investigated energy savings in the data cache. "Selective cache ways" can also be used in the instruction cache. As all the active ways are accessed simultaneously, the energy savings are much lower than "way-prediction", where only one way is accessed most of the time. This technique cannot be used in applications with large work set because the number of cache misses, which can incur high energy and performance cost, may increase dramatically.

6 Conclusion

In this paper, we have proposed a way-footprint prediction technique for energy savings in set-associative instruction caches. The hardware cost is small because it utilizes existent hardware in the branch predictor. And the added hardware is not on one of the critical path. We have investigated three BTB allocation policies for their effects on performance and energy. Each of them results in 29%, 33% and 62% instruction cache energy savings with normalized execution time of 1, 1 and 1.001 respectively.

We are currently investigating the potential performance advantages of the way-footprint prediction. For one-way cache access, the access time is shorter because there is no need for way selection. It is likely to take a shorter time for the instructions to go through the pipeline. This may result in early branch miss prediction detection and reduce the branch miss prediction penalties. In addition, the average instruction cache port utilization is decreased because of shorter cache access time. Idle ports can be used by some techniques, such as tag check during the prefetching, to improve performance.

References

[1] D. H. Albonesi. Selective cache ways: on-demand cache resource allocation. In *Int'l Symp. Microarchitecture*, pages 248–259, 1999.

[2] L. Benini and G. D. Micheli. System-level power optimization: techniques and tools. *ACM Trans. on Design Automation fo Electronic Systems*, 5(2):115–192, April 2000.

[3] D. Burger and T.Austin. The simplescalar toolset, version 2.0. Technical report, University of Wisconsin-Madison, 1997.

[4] B. Calder and D. Grunwald. Next cache line and set prediction. In *Int'l Symp. Computer Architecture*, pages 287–296, 1995.

[5] M. Check and T. Slegel. Custom S/390 G5 and G6 microprocessors. *IBM Journal of Research and Development*, 43(5/6):671–680, 1999.

[6] K. Inoue, T. Ishihara, and K. Murakami. Way-predicting set-associative cache for high performance and low energy consumption. In *Int'l Symp. on Low Power Electronics and Design*, pages 273–275, 1999.

[7] J. Montanaro et al. A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor. *IEEE Journal of Solid-State Circuits*, 32(11):1703–14, 1996.

[8] R. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, 1999.

[9] G. Reinman, T. Austin, and B. Calder. A scalable front-end architecture for fast instruction delivery. In *Int'l Symp. Computer Architecture*, pages 16–27, 1999.

[10] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud. Multiple-block ahead branch predictors. In *Int'l Symp. Architectural Support for Programming Languages and Operating Systems*, 1996.

[11] S. Wilton and N. Jouppi. An enhanced access and cycle time model for on-chip caches. Technical report, Digital Western Research Laboratory, 1994.

[12] T.-Y. Yeh, D. Marr, and Y. Patt. Increasing the instruction fetch rate via multiple branch prediction and a branch address cache. In *Int'l Symp. Computer Architecture*, pages 67–76, 1993.