Power Efficient Instruction Cache for Wide-issue Processors [*]

Ana-Maria Badulescu, Alexander Veidenbaum
Information and Computer Science Department
University of California, Irvine
{*ancuta, alexv*}*@ics.uci.edu*

Abstract

This paper focuses on reducing power in instruction cache by eliminating the fetching of instructions that are not needed from a cache line. We propose a mechanism that predicts *which instructions are going to be used out of a cache line* before *that line is fetched into the instruction buffer. The average instruction cache power savings obtained by using our fetch predictor is 22% for SPEC95 benchmark suit.*

1. Introduction

Instruction cache power consumption is significant in high performance processors, ranging between 10% and 20% of the total power [4]. This paper focuses on reducing power in instruction cache by eliminating the fetching of instructions that are not needed from a cache line. These are the instructions following a taken branch and/or the instructions preceding a branch target in a cache line. We propose a mechanism that *predicts* which instructions are going to be used out of a cache line *before* that line is fetched into the instruction buffer. The prediction is used in each fetch cycle to fetch only the useful part of the cache line, and thus saving power.

Besides predicting which instructions to fetch, we also need a cache that has the ability to fetch any sequence of instructions from a cache line. This is similar to dividing the cache into subbanks [8]. A subbank consists of a number of RAM bits which in our case is equal to the width of an individual instruction. We use a control vector to activate only the subbanks that contain the instructions we want to fetch. This capability has been implemented in RS/6000 [7]: the instruction cache was organized as 4 separate arrays, each of which could use a different row address.

[*]This work is supported in part by DARPA under the PAC/C program

2. Previous Work

There have been several proposals for reducing the power consumption of on-chip caches. They aim to either reduce the width of associative set search/access or reduce the cache size in some way.

The *phased cache* [3] attempts to reduce the power consumption of a set associative cache by dividing the cache access into two phases. First, all the tags in the set are examined in parallel and no data access occur. Next, if there is a hit, a data access is performed for the hit way. This technique reduces the average power consumption by about 70%. However, since it has a longer cache-hit time, it increases the average cache-access time by 100%.

The *set-associative cache with way-prediction* [5] speculatively selects one way before it starts a normal cache access. If the way prediction is correct, the power is reduced by a factor of 4. On prediction misses, however, the cache-access time is increased due to the access to the other ways in the cache.

The *loop cache* reduces the instruction fetch power when executing a tight program loop. If a loop can fit in the small instruction cache which is called loop cache, than all the instruction requests are directed to the loop cache and the main cache is shut down completely. There is no performance degradation associated with this technique. Similarly, small L0 cache has been proposed as a low-power extra level in memory hierarchy [6].

3. Predictor algorithm

We design a fetch predictor that determines the fetch control vector for each line to be fetched. The control vector is a bit mask whose length is equal to the number of instructions in a cache line. Each bit in the mask is used to enable or disable the subbanks with the words to be fetched in the next fetch cycle.

The fetch predictor is built on top of the branch predictor, since the branch predictor provides the PC of the next instruction . We use the branch misprediction detection and recovery mechanism along with the branch predictor update phase in order to update the fetch predictor.

For eliminating the fetch of the instructions preceding a branch target, there is no need for a new mechanism. We use the existing BTB to determine the target address of the branch that is taken. Based on the position of the target in the next cache line, we build a *target-mask* with the first bits till the target disabled, and the remaining bits till the end of the line enabled.

For eliminating the fetch of the instructions following a taken branch, we do need a new mechanism. It will predict whether or not the next cache line to be fetched contains any taken branch instructions. To accomplish this, we build a *predicted-mask* table, with a one-to-one mapping between its entries and the cache lines. The information associated with each line contains the prediction in the form of a bit mask with the first bits till the taken branch enabled, and with the remaining bits till the end of the line disabled. The *predicted-mask* is continuously updated.

There are situations when both a target and a taken branch may occur in the same cache line. In this case, the *target-mask* and the *predicted-mask* need to be combined.

For example, consider the following two cache lines:

$lineA : i_1, branch_A\_to\_target_A, i_2, i_3$

$lineB : i_4, target_A, branch_B, i_5$

$i_j, j = 1, 5$ and $target_A$ are non-branching instructions. Assume that line A is the current line being executed. The branch from line A is taken and its target is $target_A$ from line B. If the branch from line B is also taken, then only the second and the third instruction from line B must be fetched. The other two are not used. In this case, *target-mask* $= 0111$ and *predicted-mask* $= 1110$ (1 = enable the subbank/fetch, 0 = disable the subbank/do not fetch). The *fetch-mask* $= 0110$ that results after combining these two masks by a logical AND will be used for fetching line B.

The fetch predictor works in the following way:

1. All the masks in the predicted-mask table are initialized to all 1's, equivalent to fetching all the instructions in a cache line.

2. When a cache miss occurs and a cache line is replaced, the mask associated in the predicted-mask table is reset to all 1's.

3. In the fetch stage

```
1. IF there is a taken branch
               in the current line
2.   THEN use the branch predictor
            and the BTB to compute
            the target-mask;
3.   ELSE set target-mask to all 1's;
4. get the predicted-mask associated
     with the next line to be fetched
     from the mask-table;
5. fetch-mask =
      target-mask AND predicted-mask;
6. IF fetch-mask equals 0
7.   THEN fetch-mask  = target-mask;
```

The last test above is necessary the following case. Consider two consecutive lines:

$lineA : i_1, branch_A\_to\_target_A, i_2, i_3$

$lineB : branch_B, target_A, i_5, i_6$

If the first time when line A is executed, $branch_A$ is not taken, the program will continue with line B. If at this point $branch_B$ is taken, than the *predicted-mask* for line B in the mask-table will be $predicted\_mask = 1000$. If the second time when line A is executed $branch_A$ is taken to $target_A$, then the masks for line B are $target\_mask = 0111$ and $predicted\_mask = 1000$, so *fetch-mask* will be zero. As per steps 6 and 7 above, the *fetch-mask* used for fetching line B in this case will be equal to $target\_mask$, which is the correct mask to use.

4. In case of a branch misprediction, reset the *predicted-mask* in the mask-table corresponding to the cache line containing the mispredicted branch to all 1's.

5. When updating the branch predictor, also update the *predicted-mask* in the mask-table for the cache line that contains the branch instruction. If the branch that is being updated will be predicted taken next time it is going to be executed, than disable all the bits from the position of the branch to the end of the line. Otherwise, set all the bits to 1. The update to the mask-table is performed only if the line containing the branch is still in the cache.

This algorithm guarantees that the fetch predictor is never going to fetch less instructions than necessary from a cache line. This is very important for several reasons.

Firstly, fetching fewer instructions is caused by predicting a wrong mask for a particular line. In such a case, the current PC and the next PC will be on the same line and the wrong mask must not be used in the next fetch cycle. Therefore, there is a need for an additional mechanism for detecting such cases, which is not trivial. It has to verify if the last instruction executed was a branch or not, and if it was taken or not. Moreover, all these tests have to be performed before the next fetch cycle begins, in order to invalidate the wrong mask.

Secondly, the update of the masks is more complex. The masks have to be updated not only on a branch misprediction, but also on a mask misprediction.

Lastly and most important, it introduces execution overhead which results in performance penalty. The fetch cycle is critical for the execution speed and must not be extended in time.

Our fetch predictor uses the information collected from previous execution of a line to determine which instructions to fetch from the same line next time. If there is no information previously collected for a cache line, it might fetch more instructions than needed, but it will never fetch fewer. The prediction is done in the cycle previous to the fetch. Therefore, the overall performance is not affected.

4. Experimental Results

4.1. Experimental Setup

For computing the amount of saved power, we use the power model from [2]. The power consumption is modeled on the following stages: decoder, wordline drive, bitline discharge, and output drive (sense amplifier). The fetch predictor reduces the power in bitline and output drive stages. In these two stages, the power is proportional with the number of bits fetched from the memory.

We use version 3.0 of the *SimpleScalar* [1] tool suite. SimpleScalar is a cycle-accurate execution-driven simulator. It models a derivative of the MIPS architecture and uses binaries compiled to a MIPS-like target. Out of the five processor simulators included in the SimpleScalar release, we used the most detailed one, sim-outorder. It models an out-of-order issue, superscalar processor that supports non-blocking caches and speculative execution.

We added two modifications to sim-outorder. First, we implemented the algorithm of the fetch predictor. Second, we extended the simulator fetch cycle to be aware of the instruction cache organization and not to fetch instructions across cache line boundaries.

Our simulations are executed on a subset of SPEC95 benchmarks. All the statistics are collected after the execution of the first 3 billion instructions for each benchmark. The branch predictor type is bimodal, with a 2K-entry counter table and a 512-entries 4-way associative BTB. We use a direct mapped instruction cache of 256 lines with 8 instructions per line. The 8 instructions are fetched in 2 accesses to the cache, 4 instructions in each.

The size of the fetch predictor in bits is equal to the product between the number of cache lines and the number of instructions that fit into a cache line. For an instruction cache of 256 lines with 8 instructions per line, the fetch predictor size is 256 Bytes. The power overhead of such a small table in insignificant compared to the power consumed by a branch predictor, a BTB table or an instruction cache, all of which are accessed every clock cycle as well.

4.2. Results

The instruction cache power saved by using the fetch predictor varies with each benchmark. The lower bar in Figure 1 represents the power savings relative to the total cache power, obtained by using the fetch predictor. The savings for the integer benchmarks are noticeably larger than for the floating point benchmarks. For the integer benchmarks, the number of branches per instruction (BPI) is high. Therefore, the probability of having branch on a cache line is also high, which means there is more potential for saving power.

The average instruction cache power savings using the proposed fetch predictor is 22% – ranging between 4% and %31. The overall processor power savings is 4.4%, assuming that the instruction cache consumes 20% of the total.

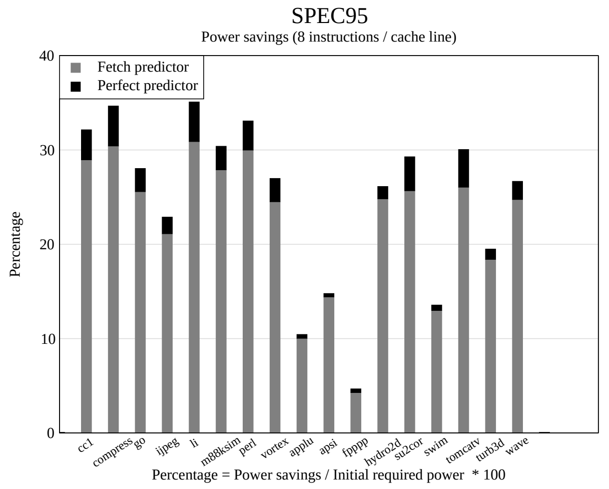

Figure 1. Instruction cache power savings

The whole bar in Figure 1 shows the maximum amount of instruction cache power that can be saved for each benchmark by using a perfect fetch predictor. This predictor knows exactly which instructions will be executed and never mispredicts. In terms of power, the average power savings for a perfect predictor is 24%, ranging between 5% and 35%. Assuming that the instruction cache consumes about 20% of the total processor power, the overall average savings obtained by using a perfect predictor is 4.8%.

The differences in savings between the perfect predictor and the fetch predictor are due in part to the branch prediction accuracy. A branch misprediction results in reseting the mask for a line to all 1's, which lowers the accuracy of the fetch predictor. The instruction cache miss rate has a negative impact as well. Each time a line is replaced in the cache, the corresponding mask is reset to all 1's.

5. Conclusions

In this paper we propose saving power by fetching only the useful instructions from the instruction cache. We propose a fetch predictor for determining which words to fetch in each cycle. The hardware overhead of the fetch predictor is a table of bit masks, one per cache line. The length of each mask is equal to the number of instructions in a cache line. The power consumption of such a table is insignificant compared to the total power of the processor.

The experiments we did for SPEC95 show that the savings obtained by using our fetch predictor and a fetch predictor are very close. The relative difference between them is only 8%.

Lastly, but very important, there is no performance penalty associated with our mechanism. The instruction cache access time remains the same.

References

[1] D. Burger and T.M. Austin. The SimpleScalar Tool Set, version 2.0. Technical Report 1324, University of Wisconsin, June 1997.

[2] M. Martonosi D. Brooks, V. Tiwari. Wattch: A framework for architectural-level power analysis and optmizations. In *ISCA*, 2000.

[3] A. Hasegawa et al. High code density, low power. In *IEEE Micro*, pages pp 11–19, 1995.

[4] J. Montanaro et al. A 160-MHz, 32-b, 0.5-W CMOS RISC Microprocessor. In *IEEE ISSCC*, 1996.

[5] K. Inove et al. Way-predicting set-associative cache for high perfomarnce and low energy consumption. In *ISLPED*, pages pp 273–275, 1999.

[6] N. Bellas et al. A new scheme for I-cache energy reduction in high-performance processors. In *Power Driven Microarchitecture Workshop, held in conjunction with ISCA*, 1998.

[7] Architecture Manual. IBM RISC System/6000 Processor.

[8] C. Su and A. Despain. Cache design tradeoffs for power and performance optimization: A case study. In *International Symposium on Low Power Design*, pages pp 63–68, 1995.