Energy-Effective Instruction Fetch Unit for Wide Issue Processors

Juan L. Aragón[1] and Alexander V. Veidenbaum[2]

[1] Dept. Ingen. y Tecnología de Computadores,
Universidad de Murcia, 30071 Murcia, Spain
jlaragon@ditec.um.es
[2] Dept. of Computer Science,
University of California, Irvine, 92697-3425 Irvine, CA, USA
alexv@cecs.uci.edu

Abstract. Continuing advances in semiconductor technology and demand for higher performance will lead to more powerful, superpipelined and wider issue processors. Instruction caches in such processors will consume a significant fraction of the on-chip energy due to very wide fetch on each cycle. This paper proposes a new energy-effective design of the fetch unit that exploits the fact that not all instructions in a given I-cache fetch line are used due to taken branches. A Fetch Mask Determination unit is proposed to detect which instructions in an I-cache access will actually be used to avoid fetching any of the other instructions. The solution is evaluated for a 4-, 8- and 16-wide issue processor in 100nm technology. Results show an average improvement in the I-cache Energy-Delay product of 20% for the 8-wide issue processor and 33% for the 16-wide issue processor for the SPEC2000, with no negative impact on performance.

1 Introduction

Energy consumption has become an important concern in the design of modern high performance and embedded processors. In particular, I- and D-caches and TLBs consume a significant portion of the overall energy. For instance, the I-cache energy consumption was reported to be 27% of the total energy in the StrongArm SA110 [17]. Combined I- and D-cache energy consumption accounts for 15% of the total energy in the Alpha 21264 [9]. In addition, continuing advances in semiconductor technology will lead to increased transistor count in the on-chip caches, and the fraction of the total chip energy consumed by caches is likely to go up. Other design trends such as wider issue, in case of high performance processors, or highly associative CAM-based cache organizations, commonly used in embedded processors [6][17][27], increase the fraction of energy consumed by caches. This is especially true for the I-cache, which is accessed almost every cycle. For these reasons, the energy consumption of the fetch unit and the I-cache is a very important concern in low-power processor design.

Several techniques have been proposed to reduce the energy consumption of TLBs [7] and caches in general. Many of them proposed alternative organizations, such as filter caches [13], way-prediction [11][20][24], way determination [18], way-

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 15 – 27, 2005.

memoization [15], cached load/store queue [19], victim caches [16], sub-banking [8][23], multiple line buffers and bitline segmentation [8], the use of small energy-efficient buffers [4][13], word-line segmentation [22], divided word-lines [26], as well as other circuit design techniques that are applicable to SRAM components. Some of these proposals provide an ability to access just a portion of the entire cache line (e.g. subbanking, wordline segmentation and divided wordlines), which is particularly useful when accessing the D-cache to retrieve a single word. The I-cache fetch is much wider and typically involves fetching an entire line. However, because of the high frequency of branches in applications, in particular taken branches, not all instructions in an I-cache line may actually be used.

The goal of this research is to identify such unused instructions and based on that to propose an energy-efficient fetch unit design for future wide issue processors. When a N-wide issue processor accesses the I-cache to retrieve N instructions from a line, not all N instructions may be used. This happens in two cases, which are depicted in Fig. 1:

1. One of the N instructions is a conditional or an unconditional branch that is taken – a *branch out* case. All instructions in the cache line after the taken branch will not be used.
2. An I-cache line contains a branch target, which is not at the beginning of the N-word line – a *branch into* case. The instructions before the target will not be used.

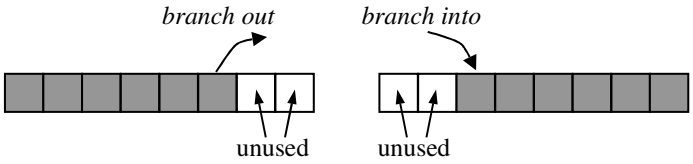

Fig. 1. *Branch out* and *branch into* cases

In this work, *Fetch Mask Determination* (*FMD*) is proposed as a technique to identify which instructions in the next I-cache line to be fetched are going to be used. Based on this information, only the useful part of the cache line is read out. Determining the unused words requires identifying the two branch cases described above. For the *branch into* case, a standard *Branch Target Buffer* (*BTB*) can be used to obtain a word address of the first useful instruction in a *next* line. For the *branch out* case, a different table is used to track if the next line to be fetched contains a conditional branch that *will* be taken. Finally, both *branch into* and *branch out* cases may occur in the same line. Therefore, both cases are combined to identify all instructions to be fetched in a given access. Once the useful instructions have been identified, the I-cache needs the ability to perform a partial access to an I-cache line. This may be achieved by using either a subbanked [8][23], wordline segmentation [22] or a divided wordline (DWL) [26] I-cache organization. Therefore, this research assumes one of these I-cache organizations to be used. The mechanism proposed in this paper will supply a bit vector to control the corresponding subbanks, pass transistors and drivers, of the underlying I-cache type.

The rest of the paper is organized as follows. Section 2 presents the related work. Section 3 motivates and describes the proposed *Fetch Mask Determination* unit. Section 4 presents the energy efficiency of the proposed mechanism. Finally, Section 5 summarizes the main conclusions of this research.

2 Background and Related Work

There have been many hardware and architectural proposals for reducing the energy consumption of caches in general (some cited above) and in particular the energy consumption of the I-cache. In a typical SRAM-based cache organization, an access to the I-cache goes through the following steps. A decoder first decodes an address and selects the appropriate RAM row by driving one wordline in the data array and one wordline in the tag array. Along the selected row, each memory cell is associated with a pair of bitlines. Initially, the bitlines are precharged high and one of them is pulled down depending on the value stored in the memory cell. Finally, a set of sense amplifiers monitors the pairs of bitlines detecting when one changes and determining the content in the memory cell. In this organization, an entire cache line is *always* read out even if only some of the instructions are used. Several approaches, that allow a partial access of a cache line, have already been applied to D-caches, supporting the idea of selectively fetching only the desired instruction words from the I-cache.

A subbanked I-cache organization divides the cache into subbanks [8][23] and activates only the required subbanks. A subbank consists of a number of consecutive bit columns of the data array. In the I-cache case, the subbank will be equal to the width of an individual instruction, typically 32 bits wide. Such a cache has been implemented in IBM's RS/6000 [1]. The instruction cache was organized as 4 separate arrays, each of which could use a different row address.

Divided wordline (DWL) [26] and wordline segmentation [22] are used to reduce the length of a wordline and thus its capacitance. This design has been implemented in actual RAMs. It typically refers to a hierarchical address decoding and wordline driving. In a way, it is or can be made similar to subbanking.

A related approach is bitline segmentation [8], which divides a bitline using pass transistors and allows sensing of only one of the segments. It isolates the sense amplifier from all other segments allowing for a more energy efficient sensing.

A number of other techniques have also been proposed to reduce the I-cache energy consumption. Way-prediction predicts a cache way and accesses it as a direct-mapped organization [11][20][24]. A phased cache [10] separates tag and data array access into two phases. First, all the tags in a set are examined in parallel but no data access occurs. Next, if there is a hit, the data access is performed for the hit way. This reduces energy consumption but doubles a cache hit time. Way-memoization [15] is an alternative to way-prediction that stores precomputed in-cache *links* to next fetch locations aimed to bypass the I-cache tag lookup and thus, reducing tag array lookup energy. Other proposals place small energy-efficient buffers in front of caches to filter traffic to the cache. Examples include block buffers [4][23], multiple line buffers [8], the filter cache [13] and the victim cache [16]. Again, these proposals trade performance for power since they usually increase the cache hit time.

A Trace Cache [21] *could* produce a similar behavior to the *Fetch Mask Determination* unit proposed in this work. A Trace Cache line identifies a dynamic stream of instructions in execution order that are going to be executed, eliminating branches between basic blocks. Therefore, the unused instructions due to taken branches are eliminated from the trace dynamically. However, a Trace Cache introduces some other inefficiencies, such as basic block replication and a higher power dissipation, trading power for performance. An energy-efficiency evaluation of the Trace Cache is out of the scope of this paper and is part of future work.

3 Energy-Effective Fetch Unit Design

3.1 Quantitative Analysis of Unused Fetched Instructions

In this section, the number of instructions that need not be fetched per I-cache line is studied to understand how these extra accesses may impact the energy consumption of a wide-issue processor. The SPEC2000 benchmark suite was studied in a processor with issue widths of 4, 8 and 16 instructions using a 32 KB, 4-way I-cache with a line size equal to fetch width.

The baseline configuration uses a 32 KB *2-level* branch predictor (in particular a PAs branch predictor, using the nomenclature from [25], whose first level is indexed by branch PC) and assumes a fetch unit that uses a standard *prefetch buffer* organized as a queue. The purpose of the prefetch buffer is to decouple the I-cache from the decode unit and the rest of the pipeline, as well as to provide a smooth flow of instructions to the decoders even in the presence of I-cache misses. Instructions are retrieved from the I-cache one line at a time and then placed in the fetch buffer, as long as there is enough space in the buffer to accommodate the entire line.

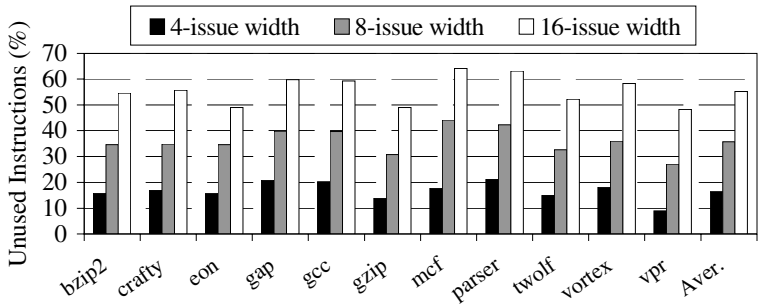

Fig. 2. Unused instructions per I-cache line for SPECint2000

Fig. 2 and Fig. 3 show the results for integer and floating point applications, respectively (see Section 4.1 for details about simulation methodology and processor configuration). For integer applications, an average of 16% of all fetched instructions are not used for the issue width of 4. This amount increases to 36% and 55% when the

issue width is increased to 8 and 16 respectively. These results show that for wide-issue processors (8-wide and up) the presence of taken branches interrupting the sequential flow of instructions is very significant and, consequently, there is a significant impact on the energy consumption of the I-cache.

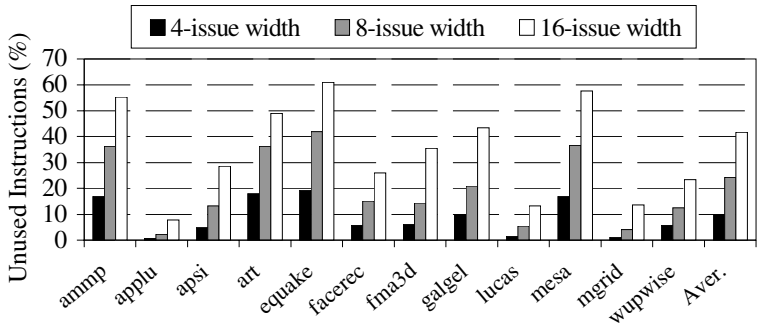

Fig. 3. Unused instructions per I-cache line for SPECfp2000

For the floating point applications (Fig. 3), the impact is not as significant. For the 8-wide issue processor the average number of unused instructions is 24%. In some benchmarks, such as *applu*, *lucas* or *mgrid*, it is less than 5%. The reason is that these applications have a high average number of instructions between branches: 200, 61, and 86, respectively. Therefore, fewer opportunities exist to locate unused instructions due to taken branches than in integer applications. For the 4-wide issue processor the average number of unused instructions is only 10%, whereas in the 16-wide issue the average is 41%, which is very significant.

These results show the potential of the proposed *Fetch Mask Determination* unit to reduce the energy consumption of the I-cache due to unused instructions in a cache line as the issue width is increased.

3.2 Fetch Mask Determination (*FMD*)

The *Fetch Mask Determination* (*FMD*) unit generates a control bit vector used to decide which instructions within an I-cache line should be fetched in the *next* cycle. 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 controls either the subbanks to be activated or the drivers in the segmented wordline, depending on the underlying I-cache organization, in order to access only the useful instructions for the next fetch cycle and, therefore, save energy.

To determine the bit mask for the next fetch cycle, let us consider each of the two cases described above: *branching into* and *branching out* of a cache line.

For *branching into* the next line, it is only necessary to determine whether the current fetch line contains a branch instruction that is going to be taken. This information is provided by both the *Branch Target Buffer* (*BTB*) and the conditional

branch predictor. Once a branch is predicted taken and the target address is known, its target position in the next cache line is easily determined. For this case, only instructions from the target position until the end of the cache line should be fetched. This mask is called a *target mask*.

For *branching out* of the next line, it is necessary to determine if the next I-cache line contains a branch instruction that is going to be taken. In that case, instructions from the branch position to the end of the line do not need to be fetched in the next cycle. To accomplish this, a *Mask Table* (*MT*) is used which identifies those I-cache lines that contain a branch that will be predicted as taken for its next execution. The number of entries in the *MT* equals the number of cache lines in the I-cache. Each entry of the *Mask Table* stores a binary-encoded mask, so each entry has $log_2(issue\_width)$ bits. Every cycle the *Mask Table* is accessed to determine whether the next I-cache line contains a taken branch. This mask is called a *mask of predictions*. When a branch is committed and the prediction tables are updated, we can check what the *next* prediction for this branch will be by looking at the saturating counter. This information is used to also update the *MT* in order to reflect if the branch will be predicted as taken the next time. Therefore, there are no extra accesses to the branch prediction tables, and thus, no additional power dissipation.

It is also important to note that there is no performance degradation associated with the proposed *Fetch Mask Determination* unit since it just anticipates the behavior of the underlying branch predictor to detect future taken branches, either correctly predicted or mispredicted. When that branch is executed again, the corresponding entry in *MT* will provide the correct mask for a *branch out* case, always in agreement with the branch prediction. In this way, the proposed *Fetch Mask Determination* unit is not performing any additional predictions and, therefore, it cannot miss. The *FMD* unit just uses the next prediction for a particular branch (*n* cycles before the next dynamic instance of the branch) to identify a *branch out* case or the information from the *BTB* to identify a *branch into* case. In addition, neither the I-cache hit rate nor the accuracy of the branch predictor affects the energy savings provided by our proposal, only the amount of branches predicted as taken. In case of branch misprediction, all necessary recovery actions will be done as usual and the corresponding *MT* entry will be reset to a mask of all 1's as explained below.

Finally, it is possible for both *branching into* and *branching out* cases to occur in the same cache line. In this case, the *target mask* and the *mask of predictions* need to be combined to determine which instructions to fetch in the next cycle. In order to simplify the *FMD* unit, the number of taken braches per cycle is limited to one. To better understand the proposed design, let us consider the following example for two different I-cache lines:

I-cache line1: I_1, $branch_1\_to\_target_A$, I_2, I_3
I-cache line2: I_4, $target_A$, $branch_2$, I_5

where I_j (j=1..5) and $target_A$ are non-branching instructions. Assume that *line1* is the line currently being fetched and that the branch in *line1* is predicted to be taken and its target is $target_A$ in *line2*. For this *branch into* case, *target_mask* = 0111 for the

second cache line. If $branch_2$ from *line2* is also going to be predicted as taken[1], then only the first three instructions from $line_2$ must be fetched. For this *branch out* case, the corresponding *MT* entry will provide a *mask_of_predictions* = 1110. When both masks are combined by a logical AND operation, the final mask is *next_fetch_mask* = 0110. This mask will be used for fetching just the required instructions from *line2*.

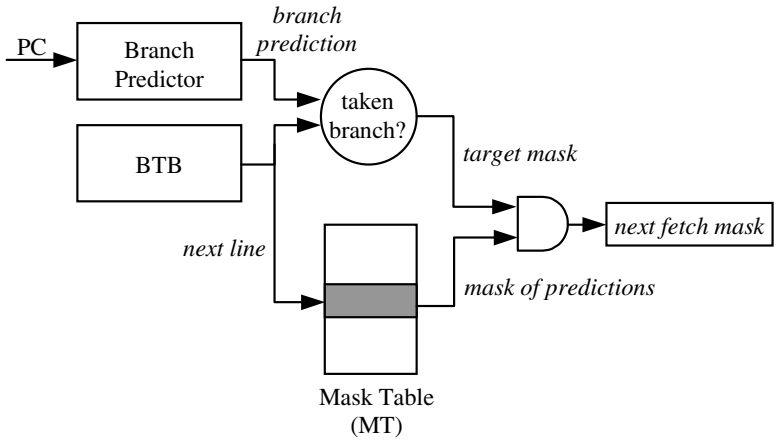

Fig. 4. The *Fetch Mask Determination* unit

The *Fetch Mask Determination* unit, depicted in Fig. 4, operates as follows:

1. Each entry in the *MT* is initialized to a mask of all 1's, which means that all instructions in the line are going to be fetched.
2. When an I-cache miss occurs and a line is replaced, the associated mask in the *MT* is reset to all 1's.
3. In the fetch stage:

```
1) if taken branch in current line
2) then use branch predictor/BTB to compute target_mask
3) else target_mask = all 1's;
4) mask_of_predictions = MT[next_line];
5) next_fetch_mask = target_mask AND mask_of_predictions;
6) if next_fetch_mask == 0
7) then next_fetch_mask = target_mask;
```

The last test above (line 6) is necessary for the following case:

I-cache line1: I_1, $branch_1$_ to_ $target_A$, I_2, I_3
I-cache line2: $branch_2$, $target_A$, I_4, I_5

If the first line is fetched and $branch_1$ is predicted as not taken, the program will continue with *line2*. If $branch_2$ is taken, the *MT* will contain for *line2* a *mask_of_predictions* = 1000. The second time that *line1* is fetched, if $branch_1$ is

[1] We are assuming that its last execution changed the 2-bit saturated counter to the *predict-as-taken* state. That information was used to update the *MT* entry with mask = 1110.

taken then *target_mask* = 0111 for *line2*. The combination of both masks will result in zero, which is incorrect. According to steps 6 and 7 above, the *next fetch mask* used for fetching *line2* must be equal to *target mask*, which is the correct mask to use.

4. When updating the branch predictor at commit, also update the *MT* entry that contains the branch. If the branch being updated will be predicted as taken for the next execution, then disable all the bits from the position of the branch to the end of the line. Otherwise, set all bits to 1. Note that the update of the *MT* is performed only if the line containing the branch is still present in the I-cache.
5. In case of a branch misprediction, reset the corresponding entry in the *MT* to all 1's. There is no other effect for our proposal in case of misprediction.

As for the effect on cycle time, note that determining the *next fetch mask* for cycle *i+1* is a two-step process. In cycle *i* the *BTB* is used to create a *target mask* for the next line. Then, the next line PC is used to access the *MT* to determine the *mask of predictions* and finally, both masks are ANDed. Since the *BTB* and *MT* accesses are sequential, the *next fetch mask* may not be ready before the end of cycle *i*. If this is the case, despite a very small *MT* size (3 Kbits – see details at the end of Section 4.2), time can be borrowed from cycle *i+1* while the address decode is in progress, before accessing the data array. Note that the decode time for the data array takes about 50% of the total access time for a 32 KB, 4-way cache per CACTI v3.2 [22]. In any case, for the chosen configurations and sizes of the I-cache, *BTB* and *MT* (shown in Table 2 in Section 4.1), the sequential access time for both the *BTB* plus *MT* has been measured to be lower than the total access time of the 32 KB, 4-way I-cache (0.95 ns) as provided by CACTI.

4 Experimental Results

4.1 Simulation Methodology

To evaluate the energy efficiency of the proposed *FMD* unit, the entire SPEC2000 suite was used[2]. All benchmarks were compiled with highest optimization level (`-O4 -fast`) by the Compaq Alpha compiler and were run using a modified version of the Wattch v1.02 power-performance simulator [5]. Due to the large number of dynamic instructions in some benchmarks, we used the *test* input data set and executed benchmarks to completion. Table 1 shows, for each integer benchmark, the input set, the total number of dynamic instructions , the total number of instructions simulated, the number of skipped instructions (when necessary) and finally, the number of conditional branches.

Table 2 shows the configuration for the simulated 8-wide issue processor. The pipeline has been lengthened to 14 stages (from fetch to commit), following the pipeline of the IBM Power4 processor [14]. For the 4- and 16-wide issue processors, the L1-cache line width, reorder buffer, load-store queue, and other functional units were resized accordingly.

[2] A preliminary evaluation of an energy-efficient fetch unit design applied to embedded processors with highly associative CAM-based I-caches can be found in [3].

Table 1. SPECint2000 benchmark characteristics

Benchmark	Input set	Total # dyn. instr. input set (Mill.)	Total # simulated instr. (Mill.)	# skipped instr (Mill.)	# dyn.cond. branch (Mill.)
bzip2	input source 1	2069	500	500	43
crafty	test (modified)	437	437	-	38
eon	kajiya image	454	454	-	29
gap	test (modified)	565	500	50	56
gcc	test (modified)	567	500	50	62
gzip	input.log 1	593	500	50	52
mcf	test	259	259	-	31
parser	test (modified)	784	500	200	64
twolf	test	258	258	-	21
vortex	test (modified)	605	500	50	51
vpr	test	692	500	100	45

Table 2. Configuration of the 8-wide issue processor. For simplicity, only *one* taken branch is allowed per cycle.

Fetch engine	Up to 8 instr/cycle, 1 taken branch, 2 cycles of misprediction penalty.
BTB	1024 entries, 2-way
Branch Predictor	32 KB PAs branch predictor (2-level)
Execution engine	Issues up to 8 instr/cycle, 128-entry ROB, 64-entry LSQ.
Functional Units	8 integer alu, 2 integer mult, 2 memports, 8 FP alu, 1 FP mult.
L1 Instr-cache	32 KB, 4-way, 32 bytes/line, 1 cycle hit lat.
L1 Data-cache	64 KB, 4-way, 32 bytes/line, 3 cycle hit lat.
L2 unified cache	512 KB, 4-way, 64 b/line, 12 cycles hit lat.
Memory	8 bytes/line, 120 cycles latency.
TLB	128 entries, fully associative.
Technology	0.10µm, Vdd = 1.1 V, 3000 MHz.

4.2 Cache Energy Consumption Model

To measure the energy savings of our proposal, the Wattch simulator was augmented with a power model for the *FMD* unit. Since the original Wattch power model was based on CACTI version 1, the dynamic power model has been changed to the one from CACTI version 3.2 [22] in order to increase its accuracy. It assumed an aggressive clock gating technique: unused structures still dissipate 10% of their peak dynamic power. The power model was extended to support partial accesses to a cache line assuming a sub-banked I-cache organization.

According to [12][22], the main sources of cache energy consumption are E_{decode}, $E_{wordline}$, $E_{bitline}$, $E_{senseamp}$ and $E_{tagarray}$. The total I-cache energy is computed as:

$$E_{cache} = E_{decode} + E_{wordline} + E_{bitline} + E_{senseamp} + E_{tagarray} \quad (1)$$

Our proposal reduces the $E_{wordline}$, $E_{bitline}$ and $E_{senseamp}$ terms since they are proportional to the number of bits fetched from the I-cache line. In general, the

$E_{wordline}$ term is very small (< 1%), whereas both $E_{bitline}$ and $E_{senseamp}$ terms account for approximately 65% of the 32 KB, 4-way I-cache energy as determined by CACTI v3.2 (which is comparable with results in [8]).

With respect to the extra power dissipated by the hardware added by the *FMD* unit, note that the size of the *MT* table is very small compared to the size of the I-cache. As cited in Section 3.2, the *MT* has the same number of entries as I-cache lines and each *MT* entry has $log_2(issue\_width)$ bits. For example, for an 8-wide issue processor with a 32 KB I-cache, the size of the *MT* is just 3 Kbits[3], which is 85 times smaller than the I-cache. In this case, the power dissipated by the *MT* has been measured to be about 1.5% of the power dissipated by the whole I-cache, which is not significant in the total processor power consumption.

4.3 Energy Efficiency of Fetch Mask Determination

This section presents an evaluation of the proposed *FMD* mechanism in a 4-, 8- and 16-wide issue processor. Figures 5 and 6 show the I-cache Energy-Delay product (EDP)[4] improvement for the SPECint2000 and SPECfp2000 respectively. In addition, the improvement achieved by an *Oracle* mechanism is also evaluated. The *Oracle* mechanism identifies precisely all the instructions used within a line in each cycle providing an upper bound on the benefits of the design proposed here.

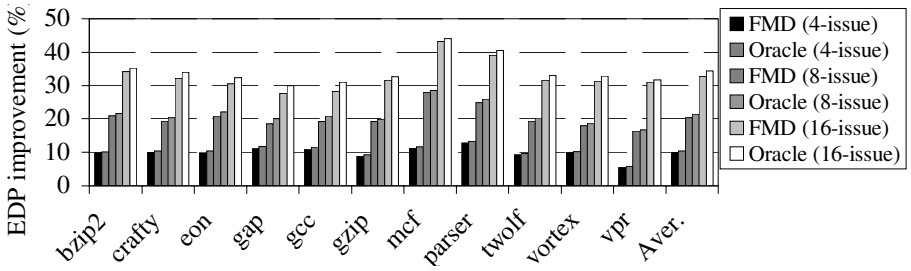

Fig. 5. I-cache Energy-Delay product improvement for SPECint2000

According to the analysis in Section 3.1, integer applications should provide more energy savings than floating point applications. As expected, Fig. 5 shows an average EDP improvement of just 10% for the 4-wide issue processor in integer codes. However, for wider issue processors the improvement increases to 20% for the 8-wide issue and 33% for the 16-wide issue. Similar trends are observed in all integer applications. Some benchmarks, such as *mcf* and *parser*, show an EDP improvement of up to 28% and 43% for the 8- and 16-issue width respectively. This high improvement is achieved because they have the lowest number of instructions per branches (*IPB*). Therefore, there is an inverse correlation between the *IPB* and the benefits of the design proposed here.

[3] This I-cache has 1024 lines, each containing eight 32-bit instructions. So, the *MT* size is 1024*3 bits.

[4] Energy savings are identical to the EDP product improvement since there is no performance degradation.

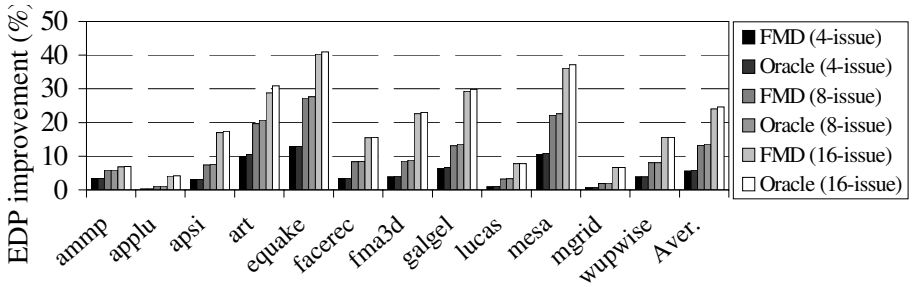

Fig. 6. I-cache EDP improvement for SPECfp2000

It is also interesting to note that *FMD* obtains an EDP improvement very close to that of the *Oracle* experiment in all benchmarks (less than 2%). This shows the effectiveness of *FMD* in determining unused instructions within an I-cache line.

For floating point applications, Fig. 6 shows an average EDP improvement of 13% for the 8-wide issue processor and 24% for the 16-wide issue processor. As in Section 3.1, some benchmarks such as *applu*, *lucas* and *mgrid* show significantly reduced EDP improvement for the 8-wide issue processor (less than 3%) due to the large number of instructions between branches. However, other floating-point applications, such as *equake* and *mesa*, have similar behavior to integer applications, and therefore, a similar EDP improvement: 40% and 36% respectively for the 16-wide issue processor.

In summary, the proposed *FMD* unit is able to provide a significant I-cache energy-delay product improvement, by not reading out of the data array in the I-cache instructions that will not be used due to taken branches. Its performance is very close to the optimal case for all benchmarks.

5 Conclusions

A modern superscalar processor fetches, but may not use, a large fraction of instructions in an I-cache line due to the high frequency of taken branches. An energy-efficient fetch unit design for wide issue processors has been proposed by means of *Fetch Mask Determination* (*FMD*), a technique able to detect such unused instructions with no performance degradation. The proposed *FMD* unit provides a bit vector to control the access to the required subbanks or to control the pass transistors in case of a segmented wordline I-cache organization. It has no impact on execution time.

The proposed design was evaluated for 4-, 8- and 16-wide issue processors. Results show an average improvement in I-cache Energy-Delay product of 10% for a 4-wide issue processor, 20% for an 8-wide issue processor and 33% for a 16-wide issue processor in integer codes. Some floating point applications show a lower EDP improvement because of the large number of instructions between branches. In addition, the proposed design was proven to be very effective in determining unused instructions in an I-cache line, providing an EDP improvement very close (< 2%) to the optimal case for all benchmarks.

Finally, the *FMD* unit is a mechanism orthogonal to other energy-effective techniques, such as fetch gating/throttling mechanisms [2] and/or way-prediction, and it can be used in conjunction with such techniques providing further energy savings.

Acknowledgements

This work has been partially supported by the Center for Embedded Computer Systems at the University of California, Irvine and by the Ministry of Education and Science of Spain under grant TIC2003-08154-C06-03. Dr. Aragón was also supported by a post-doctoral fellowship from Fundación Séneca, Región de Murcia (Spain).

References

1. IBM RISC System/6000 Processor Architecture Manual.
2. J.L. Aragón, J. González and A. González. "Power-Aware Control Speculation through Selective Throttling". *Proc. Int. Symp. on High Performance Computer Architecture (HPCA'03)*, Feb. 2003.
3. J.L. Aragón, D. Nicolaescu, A. Veidenbaum and A.M. Badulescu. "Energy–Efficient Design for Highly Associative Instruction Caches in Next–Generation Embedded Processors". *Proc. of the Int. Conference on Design, Automation and Test in Europe (DATE'04)*, Feb. 2004.
4. I. Bahar, G. Albera and S. Manne. "Power and Performance Trade-Offs Using Various Caching Strategies". *Proc. of the Int. Symp. on Low-Power Electronics and Design*, 1998
5. D. Brooks, V. Tiwari, and M. Martonosi. "Wattch: A Frame-Work for Architectural-Level Power Analysis and Optimizations". *Proc. of the Int. Symp. on Computer Architecture*, 2000.
6. L.T. Clark *et al*. "An embedded 32b microprocessor core for low-power and high-performance applications". *IEEE Journal of Solid State Circuits*, 36(11), Nov. 2001.
7. L.T. Clark, B. Choi, and M. Wilkerson. "Reducing Translation Lookaside Buffer Active Power". *Proc. of the Int. Symp. on Low Power Electronics and Design*, 2003.
8. K. Ghose and M.B. Kamble. "Reducing Power in Superscalar Processor Caches using Subbanking, Multiple Line Buffers and Bit-line Segmentation". *Proc. Int. Symp. on Low Power Electronics and Design*, pp 70-75, 1999.
9. M.K. Gowan, L.L. Biro and D.B. Jackson. "Power Considerations in the Design of the Alpha 21264 Microprocessor". *Proc. of the Design Automation Conference*, June 1998.
10. A. Hasegawa et al. "SH3: High Code Density, Low Power". *IEEE Micro*, 15(6):11-19, December 1995.
11. K. Inoue, T. Ishihara, and K. Murakami. "Way-Predicting Set-Associative Cache for High Performance and Low Energy Consumption". *Proc. Int. Symp. on Low Power Electronics and Design*, pp 273-275, Aug. 1999.
12. M. B. Kamble and K. Ghose. "Analytical Energy Dissipation Models for Low Power Caches". *Proc. Int. Symp. on Low-Power Electronics and Design*, August 1997.
13. J. Kin, M. Gupta and W.H. Mangione-Smith. "The Filter Cache: An Energy Efficient Memory Structure". *Proc. Int. Symp. on Microarchitecture*, December 1997.
14. K. Krewell. "IBM's Power4 Unveiling Continues". *Microprocessor Report*, Nov. 2000.
15. A. Ma, M. Zhang and K. Asanovic. "Way Memoization to Reduce Fetch Energy in Instruction Caches", ISCA Workshop on Complexity-Effective Design, July 2001.

16. G. Memik, G. Reinman and W.H. Mangione-Smith. "Reducing Energy and Delay using Efficient Victim Caches". *Proc. Int. Symp. on Low Power Electronics and Design*, 2003.
17. J. Montanaro *et al.* "A 160Mhz, 32b, 0.5W CMOS RISC Microprocessor". *IEEE Journal of Solid State Circuits*, 31(11), pp 1703-1712, November 1996.
18. D. Nicolaescu, A.V. Veidenbaum and A. Nicolau. "Reducing Power Consumption for High-Associativity Data Caches in Embedded Processors". *Proc. Int. Conf. on Design, Automation and Test in Europe (DATE'03)*, pp. 11064-11069, March 2003.
19. D. Nicolaescu, A.V. Veidenbaum and A. Nicolau. "Reducing Data Cache Energy Consumption via Cached Load/Store Queue". *Proc. Int. Symp. on Low Power Electronics and Design (ISLPED'03)*, pp. 252-257, August 2003.
20. M. D. Powell, A. Agarwal, T. Vijaykumar, B. Falsafi, and K. Roy. "Reducing Set-Associative Cache Energy via Way-Prediction and Selective Direct-Mapping". *Proc. Int. Symp. on Microarchitecture*, December 2001.
21. E. Rotenberg, S. Bennett, and J.E. Smith. "Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching". *Proc. of the 29th Int. Symp. on Microarchitecture*, Nov. 1996.
22. P. Shivakumar and N.P. Jouppi. "Cacti 3.0: An Integrated Cache Timing, Power and Area Model". Tech. Report 2001/2, Digital Western Research Lab., 2001.
23. C. Su and A. Despain. "Cache Design Tradeoffs for Power and Performance Optimization: A Case Study". *Proc Int. Symp. on Low Power Design*, 1995.
24. W. Tang, A.V. Veidenbaum, A. Nicolau and R. Gupta. "Integrated I-cache Way Predictor and Branch Target Buffer to Reduce Energy Consumption". *Proc. of the Int. Symp. on High Performance Computing, Springer LNCS 2327*, pp.120-132, May 2002.
25. T.Y. Yeh and Y.N. Patt. "A Comparison of Dynamic Branch Predictors that Use Two Levels of Branch History". *Proc. of the Int. Symp. on Computer Architecture*, pp. 257-266, 1993.
26. M. Yoshimito, K. Anami, H. Shinohara, T. Yoshihara, H. Takagi, *et al*. "A Divided Word-Line Structure in the Static RAM and its Application to a 64k Full CMOS RAM". *IEEE J. Solid-State Circuits*, vol. SC-18, pp. 479-485, Oct. 1983.
27. M. Zhang and K. Asanovic. "Highly-Associative Caches for Low-power processors". *Proc. Kool Chips Workshop, 33rd Int. Symp. on Microarchitecture*, 2000.