Reducing the frequency of tag compares for low power I-cache design

Ramesh Panwar and David Rennels
4731G Boelter Hall
Computer Science Department
University of California
Los Angeles, California 90024
panwar,rennels@cs.ucla.edu

Abstract

In current processors, the cache controller, which contains the cache directory and other logic such as tag comparators, is active for each instruction fetch and is responsible for 20-25% of the power consumed in the I-cache. Reducing the power consumed by the cache controller is important for low power I-cache design. We present three architectural modifications, which in concert, allow us to reduce the cache controller activity to less than 2% for most applications. The first modification involves comparing cache tags for only those instructions that result in fetches from a new cache block. The second modification involves the tagging of those branches that cause instructions to be fetched from a new cache block. The third modification involves augmenting the I-cache with a small on-chip memory called the S-cache. The most frequently executed basic blocks of code are statically allocated to the S-cache before program execution. We present empirical data to show the effect that these modifications have on the cache controller activity.

1 Introduction

Caches are an integral part of processors because of the increasing disparity between processor cycle time and memory access time. While caches are important for high performance, they are also important for low power since they reduce the amount of off-chip traffic. In a typical processor with a split cache architecture, the I-cache consumes more power than the D-cache because the I-cache is accessed for each instruction while the D-cache is accessed only for loads and stores. Since loads and stores constitute less than 25% of the executed instructions, the activity of the D-cache is less than 25% of the activity of the I-cache. The cache controller, which contains the cache directory and other logic such as tag comparators, can account for 20-25% of the power consumed in the I-cache. This work focuses on reducing the power consumption of the I-cache by reducing the controller activity through architectural modifications.

The organization of this paper is as follows. Section 2 presents a discussion on prefetch buffers and shows why an I-cache with a prefetch buffer is not good for low power. The discussion in this section sets the stage for the rest of the paper which assumes that low power processors will not have prefetch buffers. Section 3 presents the conditional tag compare scheme whereby a cache directory lookup and tag compare is done only for those instructions that result in fetches from a new cache block. This scheme requires minor hardware modifications involving a few gates; hence its cost is negligible. This scheme also relies on an architectural modification called branch tagging whereby branches that result in an instruction fetch from a new cache block are tagged. This entails a change to the instruction set architecture. However, since most instruction sets have unused space in the fields of the branch instruction, this modification doesn't incur any extra cost. Section 4 presents the S-cache which is a small on-chip memory that augments the I-cache. The most frequently executed basic blocks are statically allocated to the S-cache before program execution. The S-cache obviates the need for tag compares for the instructions in these basic blocks. The impact of all the three modifications has been evaluated empirically using several benchmarks. The benchmarks and the empirical protocol are presented in Section 6. Finally, we conclude the paper with a summary of the results.

2 Prefetch buffers

Most recent processors such as the PowerPC [6] and the Pentium [4] have prefetch buffers which allow an entire block of data to be fetched from the I-cache on a

*Partial support for this work was provided by the Office of Naval Research under grant N00014-91-J-1009.

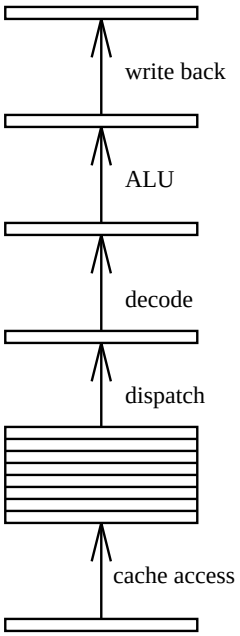

Figure 1: Pipeline with prefetch buffers

single cache access. A representative pipeline for such a processor is shown in figure 2. The prefetch buffer allows the fetch unit to build a stockpile of instructions into which it is able to look ahead to detect branch instructions early. When a branch is encountered, the prefetch buffer can be used to continue feeding the pipeline while the branch target is fetched. Prefetch buffers can thus reduce branch penalties in processors. While prefetch buffers are good for performance, they are bad for power as can be shown using the metrics[1] that follow.

To show the impact of prefetch buffers on power consumption, we ignore the cache controller since its activity can be reduced to less than 2% using the techniques that follow. The dominant contribution to the power consumption in the I-cache is then due to the bit line capacitance and the sense amplifiers [3]. If we consider the energy required to fetch a block of instructions, then it makes no difference as to whether the instructions in the block are read out one at a time or whether the entire block of instructions is read out in a single cycle. Consider a prefetch buffer that can hold s_B instructions. Let the hit rate of the prefetch buffer be h_B. If an application executes n instructions, then $n(1-h_B)$ I-cache fetches will be required. Since s_B instructions are fetched on each access, the total number of instructions fetched from the I-cache is $n(1-h_B)s_B$. The impact of the prefetch buffer on the power consumption can thus be characterized by a metric called the buffer traffic ratio t_B which is defined as the ratio of the number of instructions fetched from the I-cache in the absence of a prefetch buffer to the number of instructions fetched in the presence of a prefetch buffer. Clearly, the buffer traffic ratio is given by:

$$t_B = \frac{1}{(1-h_B)s_B}$$

Figure 2 shows the buffer traffic ratios for some applications as a function of the buffer size. The applications used are the same applications that we will use later in Section 6.

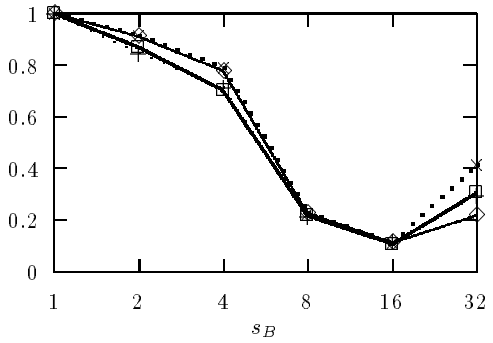

Figure 2: Buffer traffic ratios

The figure shows that even a small prefetch buffer that holds only four instructions has a very detrimental effect on the power consumption of the I-cache since the number of instructions fetched from the I-cache increases by at least 60%. The detrimental effect of the prefetch buffer on the I-cache power increases with increasing buffer size since the likelihood of prefetching useless instructions into the buffer increases. A prefetch buffer that can hold 16 instructions will increase the I-cache power consumption by 1000%. Clearly, prefetching the entire cache block is not a good idea for low power design. In low power processors, instructions should be fetched on a cycle-by-cycle basis from the I-cache. When instructions are fetched on a cycle-by-cycle basis, the cache controller is active on each cycle and can account for 20-25% of the power consumed in the I-cache. The techniques presented in the rest of this paper show how the cache controller activity can be reduced to less than 2% even when instructions are fetched on a cycle-by-cycle basis.

3 Conditional tag compares

Cache directory lookups and tag compares do not need to be done for all instruction fetches. Let us con-

[1] The best metric for evaluating the impact of any architectural feature would be the power or energy consumed by representative applications. However, direct power or energy metrics require many implementation details. To avoid making implementation assumptions, we shall use indirect metrics in our paper. These metrics can be easily translated into direct power or energy metrics when the implementation details are known.

sider two instructions i and j where the execution of j immediately follows the execution of i. We need to consider four cases. The first case, called intrablock non-sequential flow, occurs when i is a taken branch instruction with j as its target and i and j reside in the same cache block. The second case, called intrablock sequential flow, occurs when i is a non-branch or untaken branch instruction and i and j reside in the same cache block. The third case, called interblock non-sequential flow, occurs when i is a taken branch instruction with j as its target and i and j reside in different cache blocks. The fourth case, called interblock sequential flow, occurs when i is a non-branch or untaken branch instruction and i and j reside in different cache blocks. If i and j map to the same cache block, then a tag compare for j is not necessary since the block containing j must be present in the cache. Thus, cache directory lookup and tag compare is required only for interblock non-sequential flow and interblock sequential flow.

3.1 Interblock non-sequential flow

A non-sequential instruction fetch is directly indicated by the control signal that loads the program counter (PC). Interblock non-sequential flow can be indicated by branch tagging whereby unused space in the opcode is used as a compiler hint that the branch will transfer control to an instruction outside the current cache block. Table 1 shows the branch instruction frequency f_1 in the four applications that we used in our study. The probability p_f of a branch being a forward branch is also shown in the table. Table 2 shows the probability p_i of a forward branch causing interblock flow as a function of the cache block size in bytes. We have found that all backward branches get captured by the S-cache which we present in the next section. Hence we are concerned only with forward branches for interblock non-sequential flow. The frequency of tag compares necessitated by this type of flow is given by $p_i p_f f_1$.

Application	f_1	p_f
espresso	0.1058	0.7411
gzip	0.0964	0.7180
latex	0.1350	0.8062
wacc	0.0997	0.7588

Table 1: Branch statistics

The frequency of branch instructions is heavily dependent on the architecture, the compiler and the application. For example, architectures which have the capability of guarded or conditional execution can have fewer non-sequential instruction fetches because the effective size of the block blocks is increased [5]. A guarded instruction is a normal instruction augmented with a guard condition specifier. The instruction is executed if the guard condition evaluates to true else the instruction is treated as a NOP. Some of the recent RISC architectures such as the SPARC V9 [7] provide simple guard instructions like conditional moves.

Block size	espresso	gzip	latex	wacc
8	1.0000	1.0000	1.0000	1.0000
16	0.9834	0.9750	0.9918	0.9489
32	0.8381	0.9245	0.8709	0.8878
64	0.6690	0.6507	0.7244	0.6929
128	0.5381	0.4920	0.5900	0.5205
256	0.4245	0.2819	0.4224	0.3892

Table 2: Forward branch interblock flow probability

3.2 Interblock sequential flow

Interblock sequential flow can be detected quite simply by looking at a single bit of the program counter (PC) and EXORing it with the value of the bit from the previous cycle. Let the cache block size be 2^k. If the bits of the PC are labeled 31,30,...,1,0 with 0 being the least significant bit, then interblock sequential flow can be detected by EXORing bit k of the PC with the previous value of the bit. The frequency f_2 of tag compares necessitated by interblock sequential flow depends on the size of the cache block. It can be shown that this frequency is given by the following expression:

$$f_2 = \frac{1 - f_1}{s_B}$$

where f_1 is the frequency of non-sequential instruction fetches and s_B is the number of instructions contained in a single cache block.

It can be seen that the hardware overhead for doing conditional tag compares is quite minimal. A signal can be generated on a cycle-by-cycle basis as to whether a tag compare is needed and the I-cache controller enabled accordingly. Otherwise the cache controller can be disabled and the tag array can be kept in a state of constant precharge. We now present the S-cache which will allow us to reduce the cache controller activity even further.

4 S-cache

The Pareto principle applies to program execution since programs spend most of their execution time

within a few basic blocks of code. This principle can be exploited to reduce the cache controller activity even further by augmenting the I-cache with a small on-chip memory in which frequently executed basic blocks of code are stored before program execution. This small memory is called the S-cache since basic blocks of code are statically cached in this memory. On-chip program memories are found in many digital signal processors. Our S-cache was inspired by the write control store (WCS) of some earlier machines. The difference between the S-cache and the WCS of earlier machines is that the WCS was used to store microinstructions while the S-cache is used to store macroinstructions. The S-cache may even store decoded instructions to bypass the instruction decoding stage to save power.

4.1 Size

The issue of the size of the S-cache can be addressed by observing that only processes that run for significant amounts of time and thus consume significant amounts of energy should request space allocation in the S-cache. Such processes which can request space in the S-cache are called PSC (Possibly Statically Cached) processes. The linker/loader needs to make the appropriate patches to a PSC executable when it is loaded for running. Some processes do not need to have basic blocks allocated to the S-cache. such processes which use only the I-cache for execution are called PDC (Purely Dynamically Cached) processes. As an example, the process associated with the Unix `hostname` command executes less than 2000 instructions and should be compiled as a PDC executable. In Section 5, we will show that an space allocation of 4 KB in the S-cache is sufficient for most processes. Thus, if the S-cache is required to support the simultaneous execution of n_P PSC processes, then a size of $4n_P$ KB is sufficient. If more than n_P PSC processes need to run simultaneously, then they can still do so except that only n_P of them will be able to get their basic blocks allocated to the S-cache. The other PSC processes can either wait until space is available in the S-cache or they can go ahead and execute without using the S-cache. The kernel may implement other policies such as swapping out the S-cache state of a PSC process to allow another PSC process to use the S-cache. The parameter n_P is crucial in determining the size of the S-cache. Various tradeoffs will need to be made by the system architect in arriving at a reasonable value for this parameter. Our guess is that $n_P = 4$ would suffice for most portable applications.

4.2 Allocation

S-cache code allocation for PSC executables is done at the the granularity level of basic blocks. The basic blocks of a PSC executable that should be allocated to the S-cache are identified by simulating the program using several different inputs and noting the frequency of execution of the various basic blocks. With each basic block b, we can associate two integers $s(b)$ and $v(b)$ representing the size and the value of the basic block. The size of the basic block is the number of instructions it contains. The value of the basic block is the product of its execution count and size. The value $v(b)$ is thus the number of instruction executions that are contributed by the basic block. The problem of determining the best subset of the set of basic blocks $\mathcal{B}$ of a program for S-cache allocation can be formulated as a 0-1 knapsack problem as follows:

Problem 1 *Given a finite set $\mathcal{B}$ and for each $b \in \mathcal{B}$ a size $s(b) \in Z^+$ and a value $v(b) \in Z^+$ and a positive integer S, find the subset $\mathcal{B}' \subseteq \mathcal{B}$ such that $\sum_{b \in \mathcal{B}'} v(b)$ is maximized under the constraint $\sum_{b \in \mathcal{B}'} s(b) \leq S$.*

The integer S represents the size of the per-process space in the S-cache. We have stated above, that a good choice of the per-process space would be 4 KB which corresponds to 1024 instructions. The 0-1 knapsack can be solved optimally by using dynamic programming [2].

4.3 Block patching

Since basic blocks of code are moved to the S-cache, some blocks may need to be patched to ensure correct execution. Only blocks that are terminated by conditional branches need patches. A block b_i that is terminated by a conditional branch needs to be patched if it is marked for S-cache allocation and its succeeding block b_j isn't. The patch to such a basic block b_i involves the addition of a jump instruction as the last instruction of the block. This jump instruction transfers control from block b_i to block b_j so that the correct execution is ensured when the conditional branch, which is now the penultimate instruction in block b_i, is not taken and execution falls through to patching jump instruction. Likewise, a block b_i that is not marked for S-cache allocation needs to be patched if its succeeding block b_j is marked for S-cache allocation.

5 Controller activity

The combined effect on the cache controller activity can be modeled analytically. Let us designate the S-cache hit rate as h_S. The cache controller is inactive when instructions are fetched from the S-cache. When instructions are fetched from the I-cache, the cache controller is active only when we have an interblock flow in the program execution. The frequency f_T of interblock flow is given by:

$$f_T = p_i p_f f_1 + f_2$$

The controller activity a_C is merely $(1 - h_S)f_T$ and is given by:

$$a_C = (1 - h_S)\frac{1 + (p_i p_f s_B - 1)f_1}{s_B}$$

If we choose some typical numbers for the parameters in the above equation, then we can obtain a rough estimate of the controller activity. If we choose $s_B = 8$, $f_1 = 0.1$, $p_i = 0.6$, $p_f = 0.8$, and $h_S = 0.8$, then the controller activity is around 3%. This implies that the combination of the techniques presented in this paper can almost completely eliminate the cache controller activity in I-caches. We present the empirical validation of this result.

Run	Instructions executed			
	espresso	wacc	gzip	latex
1	302813600	57488356	107409918	16762184
2	107272678	215085286	45891599	30107819
3	64649453	52917720	61732450	37300037
4	74371383	57908955	176942537	53222302
5	78960096	83162293	130333009	48597884
6	27755207	80954023	211816605	80954023
7	156488163	81306286	143738664	34959011

Table 3: Instructions executed in each benchmark run

6 Empirical evaluation

The efficiency of the techniques was evaluated using four benchmarks. The benchmarks used in this study were *espresso*, a PLA minimization program, *wacc*, a lossy image compression program using wavelets and arithmetic coding, *gzip*, the Gnu lossless compression program using a Lempel-Ziv compressor, and *latex*, a program for document preparation. Each benchmark was run with seven different inputs since our hypothesis was that a single run would not provide sufficient confidence for basic block allocation in the static cache. It appeared at the onset of the experiment that a program like *espresso*, which implements several different algorithms for PLA minimization, can use different algorithms depending on the nature of the PLAs and hence emphasize different basic blocks. Likewise, for a program like *latex*, two inputs that differ significantly, such as one requiring a lot of mathematical typesetting but having no figures or tables and the other requiring very little mathematical typesetting but having a lot of figures and tables, can emphasize different basic blocks. After running an extensive set of simulations we have concluded that even a single run provides a very good estimate of the relative execution frequency of the basic blocks as would be experienced as an average over all possible runs. Our experiments resulted in roughly 2.6 billion instructions being executed which is an average of 93.5 million instructions per run. Table 3 shows the number of instructions executed in the various benchmark runs.

Shade, an instruction-level tracing tool [1], was used for instruction tracing on SPARC workstations running SunOS 5.3. Shade allowed us to do custom analysis of the instruction traces for the SPARC V8 architecture.

7 Results

We first investigated the applicability of the Pareto principle for program execution. The Pareto graph for *wacc* benchmark is shown in Figure 3. The Pareto graph gives a first cut estimate of the expected S-cache hit ratio. From the Pareto graph for *wacc*, it appears that even an S-cache allocation of 1 KB will result in a hit rate of larger than 90%.

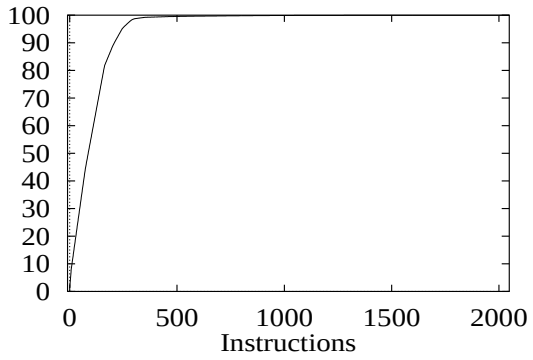

Figure 3: Pareto graph for instruction execution in wacc

Figure 4 shows the number of basic blocks that can be allocated in the S-cache for varying amounts of per-process allocation space for *espresso*. The figure also shows the number of blocks that require patches. We find that about 150 basic blocks can be allocated to the S-cache for a per-process allocation of 4 KB. The identity of these blocks was determined by dynamic programming as indicated above.

Tables 4 and 5 show the cache controller activity a_C for *espresso* and *latex*. Due to constraints of space, we have not shown the results for *wacc* and *gzip*. The results for *wacc* and *gzip* are even better than those of *espresso* and *latex* that are shown here. The activity is shown as a function of the per-process allocation in the S-cache and the block size of the I-cache. The experimental data shows that the S-cache in conjunction

with conditional tag compares can almost completely eliminate the power overhead of the cache controller by rendering it inactive 98% of the time for an S-cache per-process allocation of 4 KB and 64-byte blocks in the I-cache.

8 Conclusion

We presented a case against prefetch buffers in low power processors. Since the lack of prefetch buffers will necessitate instruction fetching on a cycle-by-cycle basis from the I-cache, the I-cache controller can be active 100% of the time unless we adopt architectural techniques to reduce the controller activity. The I-cache controller can consume 20-25% of the I-cache power if it is active 100% of the time. We have presented three architectural modifications that when used in concert, reduce the I-cache controller activity to less than 2% for most applications. We have presented empirical data using benchmarks with over 2.6 billion instructions in the address traces to justify our claims of the efficiency of the proposed techniques.

References

[1] Cmelik, R., "Shade: a fast instruction set simulator for execution profiling," *Technical Report TR-93-12*, Sun Microsystems, Mountain View, California, 1993.

[2] Horowitz, E. and Sahni, S., "Computing partitions with applications to the knapsack problem," *Journal of the Association for Computing Machinery*, **21**, 277–292, 1974.

[3] Liu, D. and Svensson, C., "Power consumption estimation in CMOS VLSI chips," *IEEE Journal of Solid-State Circuits*, **29**, 663–670, 1994.

[4] *Pentium Processor User's Manual.* Mt. Prospect, IL: Intel, 1993.

[5] Pnevmatikatos, D. and Sohi, G., "Guarded execution and branch prediction in dynamic ILP processors," *Proceedings of the 21st Annual International Symposium on Somputer Architecture*, 120–129, 1994.

[6] Weiss, S. and Smith, J., *Power and PowerPC.* San Francisco, CA: Morgan Kauffman, 1994.

[7] *The SPARC Architecture Manual.* Englewood Cliffs, NJ: Prentice Hall, 1993.

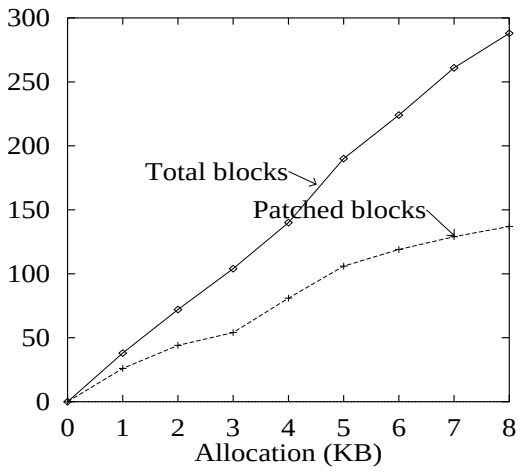

Figure 4: Number of basic blocks in S-cache

S-cache (KB)	I-cache block size 32	64	128	256
1	4.4107	3.2023	2.5557	2.2486
2	3.4587	2.5087	2.0015	1.7593
3	2.7588	2.0073	1.5966	1.4032
4	2.2368	1.6279	1.2948	1.1371
5	1.7821	1.2931	1.0467	0.9064
6	1.4312	1.0381	0.8287	0.7263
7	1.1761	0.8529	0.6807	0.5981
8	0.9901	0.7172	0.5731	0.5045

Table 4: Controller activity for espresso

S-cache (KB)	I-cache block size 32	64	128	256
1	4.7974	3.7513	3.1652	2.8713
2	3.9543	3.0956	2.6078	2.3674
3	3.3708	2.6371	2.2236	2.0193
4	2.9106	2.2801	1.9293	1.7490
5	2.5563	1.9982	1.6802	1.5294
6	2.2543	1.7723	1.4973	1.3187
7	2.0156	1.5752	1.3298	1.2020
8	1.8029	1.4073	1.1883	1.0594

Table 5: Controller activity for latex