# Power and Performance Tradeoffs using Various Caching Strategies

R. Iris Bahar [†] Gianluca Albera [§†] Srilatha Manne [‡]

[†] Brown University
Division of Engineering
Providence, RI 02912

[§] Politecnico di Torino
Dip. di Automatica e Informatica
Torino, ITALY 10129

[‡] University of Colorado
Dept. of ECE
Boulder, CO 80309

## Abstract

*In this paper, we propose several different data and instruction cache configurations and analyze their power as well as performance implications on the processor. Unlike most existing work in low power microprocessor design, we explore a high performance processor with the latest innovations for performance. Using a detailed, architectural-level simulator, we evaluate full system performance using several different power/performance sensitive cache configurations such as increasing cache size or associativity and including buffers along side L1 caches. We then use the information obtained from the simulator to calculate the energy consumption of the memory hierarchy of the system. As an alternative to simply increasing cache associativity or size to reduce lower-level memory energy consumption (which may have a detrimental effect on on-chip energy consumption), we show that, by using buffers, energy consumption of the memory subsystem may be reduced by as much as 13% for certain data cache configurations and by as much as 23% for certain instruction cache configurations* without *adversely effecting processor performance or on-chip energy consumption.*

## 1 Introduction

The performance of high-end microprocessors continues to grow. This growth is due in part to the use of speculative, out-of-order execution coupled with highly accurate branch prediction. Branch prediction has increased instruction-level parallelism (ILP) by allowing programs to speculatively execute beyond control boundaries, while out-of-order execution has increased ILP by allowing more flexibility in instruction execution. In this paper we concentrate on reducing the energy demands of the high-performance processors which make use of these aggressive hardware-based techniques. In particular, we investigate architectural-level solutions that achieve a power reduction in the memory subsystem of the processor *without* compromising performance.

Prior research has been aimed at measuring and recommending optimal cache configuration for power. For instance, in [10], the authors determined that high performance caches were also the lowest power consuming caches since they reduce the traffic to the lower level of the memory system. The work by Kin [7] proposed accessing a small *filter cache* before accessing the first level cache to reduce the accesses (and energy consumption) from DL1. The idea lead to a large reduction in memory hierarchy energy consumption, but also resulted in a substantial reduction in processor performance. While this reduction in performance may be tolerable for some applications, the high-end market will not make such a sacrifice. Furthermore, energy is a function of time; if a task takes longer to execute, overall energy consumption might increase although the power dissipation is reduced. This paper proposes memory hierarchy configurations that reduce power while retaining performance.

Reducing cache misses due to line conflicts has been shown to be effective in improving overall system performance in high-performance processors. Techniques to reduce conflicts include increasing cache associativity, use of victim caches [5], or cache bypassing with and without the aid of a buffer [4, 9, 11]. Figure 1 shows the design of the memory hierarchy when using buffers alongside the first level caches.

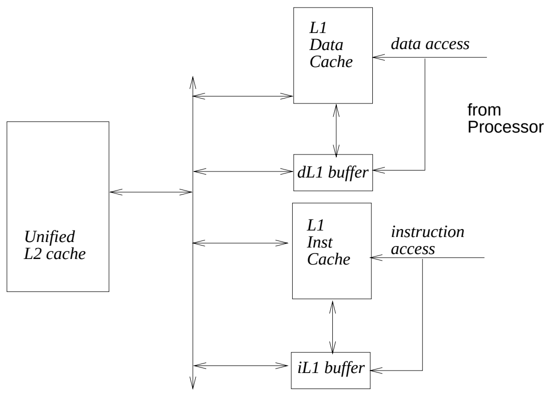

Figure 1: Memory hierarchy design using buffers alongside the L1 caches. These buffers may be used as victim caches, non-temporal buffers, or speculative buffers.

The buffer is a small cache, between 8–16 entries, located between the first level and second level caches. The buffer may be used to hold specific data (e.g. non-temporal or speculative data), or may be used for general data (e.g. "victim" data). In this paper we analyze various uses of this buffer in terms of both power and performance. In addition, we compare the power and performance impact of using this buffer to more traditional techniques for reducing cache conflicts such as increasing cache size and/or associativity.

## 2 Experimental Setup

This section presents our experimental environment. First, the CPU simulator will be briefly introduced and then we will describe how we obtained data about the energy consumption of the caches. Finally we will describe each architectural design we considered in our analysis.

### 2.1 Full Model Simulator

We use an extension of the *SimpleScalar* [1] tool suite. SimpleScalar is an execution-driven simulator that uses binaries compiled to a MIPS-like target. SimpleScalar can accurately model a high-performance, dynamically-scheduled, multi-issue processor. We use an extended version of the simulator that more accurately models all the memory hierarchy, implementing non-blocking caches and complete bus bandwidth and contention modeling [2]. Other modifications were added to handle precise modeling of cache fills.

Tables 1, 2, and 3 show the configuration of the processor modeled. Note that first level caches are on-chip, while the unified second level cache is off-chip. In addition we have a 16-entry buffer associated with each first level cache; this buffer was implemented as a fully associative cache with LRU replacement. Note that we chose a 8K first level cache configuration in order to obtain a reasonable hit/miss rate from our benchmarks since they were originally developed for smaller memory subsystem configurations than those currently available [13]. In Tables 2 and 3 note that some types of resource units (e.g., the FP Mult/Div/Sqrt unit) may have different latency and occupancy values depending on the type of operation being performed by the unit.

Our simulations are executed on SPECint95 benchmarks; they were compiled using a re-targeted version of the GNU *gcc* compiler, with full optimization. This compiler generates 64 bit-wide instructions, but only 32 bits are used, leaving the others for future implementations; in order to model a typical actual machine, we convert these instructions to 32 bits before executing the code. Since one of our architectures was intended by the authors for floating point applications [9], we also ran a subset of SPECfp95 in this case. Since we are executing a full model on a very detailed simulator, the benchmarks take several hours to complete; due to time constraints we feed the simulator with a small set of inputs. However we execute all programs entirely (from 80M instructions in *compress* to 550M instructions in *go*).

Table 1: Machine configuration parameters.

| Parameter | Configuration |
|---|---|
| L1 Icache | 8KB direct; 32B line; 1 cycle latency |
| L1 Dcache | 8KB direct; 32B line; 1 cycle latency |
| L2 Unified Cache | 256KB 4-way; 64B line; 12 cycle latency |
| Memory | 64 bit-wide; 20 cycles latency on page hit, 40 cycles on page miss |
| Branch Pred. | (MCFarling) 2k gshare + 2k bimodal + 2k meta |
| BTB | 1024 entry 4-way set assoc. |
| Return Addr. Stack | 32 entry queue |
| ITLB | 32 entry fully assoc. |
| DTLB | 64 entry fully assoc. |

Table 2: Processor resources.

| Parameter | Units |
|---|---|
| Fetch/Issue/Commit Width | 4 |
| Integer ALU | 3 |
| Integer Mult/Div | 1 |
| FP ALU | 2 |
| FP Mult/Div/Sqrt | 1 |
| DL1 Read Ports | 2 |
| DL1 Write Ports | 1 |
| Instruction Window Entries | 64 |
| Load/Store Queue Entries | 16 |
| Fetch Queue | 16 |
| Minimum Misprediction Latency | 6 |

Table 3: Latency and occupancy of each resource.

| Resource | Latency | Occupancy |
|---|---|---|
| Integer ALU | 1 | 1 |
| Integer Mult | 3 | 1 |
| Integer Div | 20 | 19 |
| FP ALU | 2 | 1 |
| FP Mult | 4 | 1 |
| FP Div | 12 | 12 |
| FP Sqrt | 24 | 24 |
| Memory Ports | 1 | 1 |

### 2.2 Power Model

Energy dissipation in CMOS technology circuits is mainly due to charging and discharging gate capacitances; on every transition we dissipate $E_t = \frac{1}{2} \cdot C_{eq} \cdot V_{dd}^2$ Watts. To obtain the values for the equivalent capacitances, $C_{eq}$, for the components in the memory subsystem, we follow the model given by Wilton and Jouppi [12]. Their model assumes a 0.8 $\mu$m process; if a different process is used, only the transistor capacitances need to be recomputed. To obtain the number of transitions that occur on each transistor, we refer to Kamble and Ghose [6], adapting their work to our overall architecture.

An $m$-way set associative cache consists of three main parts: a data array, a tag array and the necessary control logic. The data array consists of $S$ rows containing $m$ lines. Each line contains $L$ bytes of data and a tag $T$ which is used to uniquely identify the data. Upon receiving a data request, the address is divided into three parts. The first part indexes one row in the cache, the second selects the bytes or words desired, and the last is compared to the entry in the tag to detect a hit or a miss. On a hit, the processor accesses the data from the first level cache. On a miss, we use a write-back, write-allocate policy. The latency of the access is directly proportional to the capacitance driven and to the length of bit and word lines. In order to maximize speed we kept the arrays as square as possible by splitting the data and tag array vertically and/or horizontally. Sub-arrays can also be folded. We used the tool CACTI [12] to compute sub-arraying parameters for all our caches.

According to [6] we consider the main sources of power to be the following three components: $E_{bit}$, $E_{word}$, $E_{output}$. We are not considering the energy dissipated in the address decoders, since we found this value to be negligible compared to the other components. Similar to Kin [7], we found that the energy consumption of the

Table 4: **Baseline results:** Number of cycles, accesses and energy consumption in the base case. Energy is given in Joules
.

| | | DL1 Cache | | IL1 Cache | | UL2 Cache | |
|---|---|---|---|---|---|---|---|
| *Test* | *Cycles* | *Accesses* | *Eng. (Joules)* | *Accesses* | *Eng. (Joules)* | *Accesses* | *Eng. (Joules)* |
| compress | 70 538 278 | 27 736 186 | 0.092 | 66 095 704 | 0.207 | 3 134 116 | 0.402 |
| go | 826 111 517 | 169 860 620 | 0.612 | 430 587 086 | 1.499 | 60 055 380 | 6.819 |
| vortex | 250 942 324 | 93 470 624 | 0.284 | 108 663 648 | 0.375 | 15 336 216 | 1.796 |
| gcc | 392 296 667 | 103 457 950 | 0.317 | 176 609 209 | 0.618 | 22 959 880 | 2.652 |
| li | 134 701 535 | 74 445 274 | 0.232 | 140 347 169 | 0.446 | 5 856 655 | 0.652 |
| ijpeg | 139 376 153 | 73 358 576 | 0.214 | 135 804 661 | 0.437 | 3 906 431 | 0.451 |
| m88ksim | 659 451 084 | 130 319 219 | 0.369 | 241 836 872 | 0.882 | 34 953 990 | 3.980 |
| perl | 372 034 266 | 90 388 059 | 0.293 | 148 817 944 | 0.531 | 24 532 245 | 2.794 |

decoders is about three orders of magnitude smaller than that of the other components. The energy consumption is computed as:

$$E_{cache} = E_{bit} + E_{word} + E_{output} \qquad (1)$$

A brief description of each of these components follows.

**Energy Dissipated in Bit-Lines:** $E_{bit}$ is the energy consumption in the bit-lines; it is due to precharging lines (including driving the precharge logic) and reading or writing data. We assume the bit-lines are precharged to an intermediate voltage value, $\frac{1}{2}V_{dd}$. Since we also assume a sub-arrayed cache, we need to precharge and discharge only the portion directly related with the address we need to read/write.

Note that in order to minimize the power overhead introduced by buffers, in the fully associative configuration, we perform first a tag look-up and access the data array only on a hit. If timing constraints make this approach not feasible, direct mapped buffers should be considered.

**Energy Dissipated in Word-Lines:** $E_{word}$ is the energy consumption due to assertion of word-lines; once the bit-lines are all precharged we select one row, performing the read/write to the desired data.

**Energy Dissipated Driving External Buses:** $E_{output}$ is the energy used to drive external buses; this component includes both the data sent/returned and the address sent to the lower level memory on a miss request.

## 3 Experimental Results

The following section describes in detail the various caching strategies we considered and evaluates their effectiveness in terms of performance and energy consumption for several SPEC95 benchmark experiments. All experiments were compared against the base case which will be described first.

**Base Case:** As stated previously, our base case uses 8K direct mapped on-chip first level caches (i.e. DL1 for data and IL1 for instruction), with a unified 256K 4-way off-chip second level cache (UL2). Table 4 shows the execution time measured in cycles and, for each cache, the number of accesses and the energy consumption (measured in Joules). The next sections' results will show percentage decreases over the base case; thus positive numbers will mean an improvement in power or performances and negative numbers will mean a worsening in power/performance. Note that we are measuring energy for the caches only and that energy consumption refers to the *memory subsystem* only and not the energy consumption of the entire microprocessor or board.

First level caches are on-chip, so their energy consumption refers to the CPU level, while the off-chip second level cache energy refers to the board level. In the following sections we show what happens to the energy in the overall cache architecture (L1 plus L2), and also clarify whether variations in power belong to the CPU or to the board. As shown in Table 4 the dominant portion of energy consumption is due to the UL2, because of its bigger size and high capacitance board buses. Furthermore we see that the instruction cache demands more power than the data cache, due to a higher number of accesses.

**Traditional Techniques:** We first considered applying traditional approaches to reducing cache misses in order to improve energy consumption. Traditional approaches for reducing cache misses utilize bigger cache sizes and/or increased associativity.

Table 5 presents results obtained by increasing the instruction cache size and/or associativity. Consider first columns 2–7 where we present results for 8K 2-way and 16K direct mapped configurations, both with a 1 cycle latency. Reduction in cycles, energy in IL1 and total energy are shown. The overall energy consumption (i.e. *Total Energy*) is generally reduced, due to a reduced activity in the second level cache (since we decrease the IL1 miss rate). However we can also see cases (e.g. *compress* and *ijpeg*) in which we have an increase in energy. This increase occurs since these benchmarks already present a low miss rate in the base case; increasing associativity or size will increase L1 energy consumption more than it will decrease energy consumption in the total memory subsystem. Note also that even if the total energy improves, we show that the on-chip power increases significantly (up to 23% in the case of *vortex*) as the cache becomes bigger.

Increasing the size or associativity of the cache is often not possible without also increasing the cycle latency. Unfortunately, this can cause an overall decrease in processor performance even if the cache miss rate goes down This was shown in the Alpha 21264 microprocessor where processor performance decreased by about 4%

Table 5: **Increasing instruction cache size/associativity:** Percent improvement in performance and power compared to the base case. Note that columns 2–7 assume a 1 cycle lookup while columns associated with **8K-2way-2cycles** assumes a 2 cycle lookup.

| *Test* | **8K-2way-1cycle** | | | **16K-direct-1cycle** | | |
|---|---|---|---|---|---|---|
| | *%Cycles* | *%IL1 Eng.* | *%Tot. Eng.* | *%Cycles* | *%IL1 Eng.* | *%Tot. Eng.* |
| compress | 0.241 | -18.393 | -5.260 | 0.342 | -20.311 | -5.665 |
| go | 13.145 | -20.593 | 8.978 | 21.172 | -22.612 | 15.900 |
| vortex | 8.296 | -22.117 | 6.136 | 21.141 | -23.216 | 18.074 |
| gcc | 5.077 | -18.946 | 2.079 | 19.287 | -20.044 | 15.438 |
| li | 7.200 | -18.776 | 1.582 | 2.466 | -20.108 | -5.062 |
| ijpeg | 4.759 | -18.024 | -2.283 | 4.409 | -19.882 | -3.286 |
| m88ksim | 16.096 | -17.237 | 12.893 | 27.325 | -15.530 | 24.727 |
| perl | 11.131 | -18.834 | 7.854 | 19.099 | -20.270 | 14.983 |

Table 6: **Victim cache fully associative:** Percent improvement in performance and power compared to the base case.

| *Test* | **Swapping** | | | | | | **Non Swapping** | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *Data Only* | | *Inst. Only* | | *Data & Inst.* | | *Data Only* | | *Inst. Only* | | *Data & Inst.* | |
| | *%Cyc.* | *%Eng.* | *%Cyc.* | *%Eng.* | *%Cyc.* | *%Eng.* | *%Cyc.* | *%Eng.* | *%Cyc.* | *%Eng.* | *%Cyc.* | *%Eng.* |
| compress | 2.302 | 6.786 | -0.440 | -1.511 | 2.829 | 5.974 | 2.913 | 6.981 | -0.439 | -1.496 | 2.774 | 5.654 |
| go | 3.330 | 10.980 | 4.398 | 3.476 | 7.836 | 14.494 | 4.216 | 12.889 | 6.500 | 4.597 | 10.917 | 17.999 |
| vortex | 3.037 | 12.008 | 5.201 | 4.804 | 7.429 | 12.989 | 3.426 | 12.336 | 5.630 | 4.855 | 9.749 | 17.737 |
| gcc | 1.722 | 6.983 | 3.008 | 2.536 | 4.609 | 8.964 | 2.108 | 7.414 | 4.300 | 3.370 | 6.691 | 11.266 |
| li | 2.880 | 6.616 | 3.058 | 2.040 | 6.258 | 9.108 | 3.818 | 8.961 | 6.105 | 4.900 | 10.689 | 15.004 |
| ijpeg | 4.411 | 11.039 | 1.605 | -0.715 | 5.924 | 11.440 | 4.859 | 10.582 | 2.661 | -0.242 | 7.543 | 12.158 |
| m88ksim | 0.355 | 3.062 | 9.799 | 9.392 | 10.219 | 12.512 | 0.607 | 4.412 | 21.071 | 18.136 | 21.854 | 23.116 |
| perl | 0.899 | 4.485 | 5.591 | 5.063 | 6.750 | 9.781 | 1.692 | 5.420 | 7.581 | 5.637 | 8.496 | 11.364 |

when going to a 2-cycle pipelined cache configuration [3].

Since simply increasing cache size or associativity may have a detrimental effect on power and/or energy consumption, we next considered the idea of adding small buffers to hold additional L1 data (see Figure 1). These buffers effectively prevent the L1 cache from replacing blocks of data that will be referenced later (and thereby reduces the cache miss rate.)

**Victim Cache:** The first use of buffers we considered was modeled after the idea of a *victim cache* originally presented by Jouppi in [5], but with minor changes. The author presented the following algorithm. On a main cache miss, the victim cache is accessed; if the address hits the victim cache, the data is returned to the CPU and at the same time it is promoted to the main cache; the replaced line in the main cache is moved to the victim cache, therefore performing a "swap". If the victim cache also misses, an L2 access is performed; the incoming data will fill the main cache, and the replaced line will be moved to the victim cache. The replaced entry in the victim cache is discarded and, if dirty, written back to the second level cache.

We first made a change in the algorithm, performing a parallel look-up in the main and victim cache, as we saw that this helps performance without significant drawbacks on power. We refer to this algorithm as *victim cache swapping*, since swapping is performed on a victim cache hit.

We found that the time required by the processor to perform the swapping, due to a victim hit, was detrimental to performance, so we also tried a variation on the algorithm that does not require swapping (i.e. on a victim cache hit, the line is not promoted to the main cache). We refer to it as *victim cache non-swapping*.

Table 6 shows effects using a fully associative *victim cache*. They refer to the *swapping* and *non swapping* mechanisms. We present cycles and overall energy reduction for three different schemes; they use a buffer associated with the *Data* cache, the *Instruction* cache, or both of them. We observed that the combined use of buffers for both caches offers a roughly additive improvement over the single cache case. This result generally applies to all uses of buffers we tried. As stated before, we show that the *non swapping* mechanism generally outperforms the original algorithm presented by the author. In [5], the data cache of a single issue processor was considered, where a memory access occurs approximately one out of four cycles; thus the victim cache had ample time to perform the necessary swapping. The same cannot be said of a 4-way issue processor that has, on average, one data memory access per cycle. In this case the advantages obtained by swapping are often outweighed by the extra latency introduced.

Buffers need not be used only for holding "evicted" data. Instead, specific data or instructions may be sorted out and placed in the buffer before it has the opportunity of being written to the L1 cache. We next present three different sorting schemes that can be used with these buffers and evaluate their effective on performance and energy consumption.

**Non-Temporal Buffer:** We next evaluated the energy and performance benefits of using a *non-temporal* buffer as introduced by Rivers and Davidson [9]. In their work on data caches, the au-

Table 7: **Speculative buffer:** Percent improvement in performance and power compared to the base case.

| *Test* | *Data Only* | | *Inst. Only* | | *Data & Inst.* | |
|---|---|---|---|---|---|---|
| | *%Cyc.* | *%Eng.* | *%Cyc.* | *%Eng.* | *%Cyc.* | *%Eng.* |
| compress | 1.902 | 5.366 | -0.197 | -1.271 | 2.401 | 4.786 |
| go | 3.410 | 10.758 | 6.277 | 4.489 | 9.839 | 15.408 |
| vortex | 2.389 | 7.753 | 5.781 | 5.088 | 7.995 | 12.838 |
| gcc | 1.511 | 5.271 | 4.521 | 3.591 | 6.154 | 9.128 |
| li | 2.438 | 4.992 | 5.943 | 5.662 | 8.748 | 10.839 |
| ijpeg | 3.328 | 8.009 | 1.399 | -0.724 | 4.582 | 7.380 |
| m88ksim | 0.292 | 2.203 | 22.883 | 21.650 | 23.127 | 24.469 |
| perl | 0.656 | 3.245 | 6.599 | 5.646 | 7.379 | 9.147 |

thors observed that in numerical applications data accesses may be divided in two categories: *scalar* accesses that present *temporal* behavior, i.e. are accessed more than once during their lifetime in the cache, and *vector* accesses that present *non-temporal* behavior, i.e. once accessed are no longer referenced [9]. This is true, since vectors are generally accessed sequentially and often their working-set is bigger than the cache itself, so that when they need to be referenced again, they no longer reside in the cache. The idea is to use a special buffer to contain the non-temporal data and reduce conflicts in the main cache. They presented a history based algorithm that tags each line with a temporal/non-temporal bit; this information is also saved in the second level cache, requiring additionally write backs for replaced clean blocks.

Since this algorithm was intended for numerical applications we added a subset of SPECfp95 to this set of runs. Results show this technique not to be effective for integer programs; performance generally is worse by 3%–20%, while power increases by 7%–58%. We also observed that only specific numeric applications benefit from this algorithm; for example *swim* improves 6.8% in performance and 46% in overall energy consumption, but others such as *hydro2d* were 7% worse in energy consumption and 1% worse in performance.

The main reason of these negative results is due to an increased write back activity to the second level cache required by the algorithm that saves in the L2 the temporal/non-temporal information. Given a shorter latency L2 cache (e.g. on-chip L2 design), this algorithm might present better overall results.

**Speculative Buffer:** Using branch prediction, microprocessors may allow a program to execute "speculatively" down a particular branch direction before the outcome of the branch is known. Even with branch prediction accuracies over 90%, many instructions are executed unnecessarily from the wrong path. One side effect of this is that unnecessary data may be placed in the L1 cache, evicting data that may be needed again (and thus causing a miss). As an alternative use for the buffers, we considered sorting data according to its "speculative confidence". We use the confidence predictors presented in Manne *et al.* [8] to mark every cache access with a confidence level obtained by examining the processor speculation state and the current branch prediction estimate. We use the main cache to accommodate misses that are most likely on the correct path (high confidence) and the *speculative buffer* for misses that have a high probability to be from a mis-speculated path (low confidence). This idea originates from a study in which the authors found that line fills coming from mis-speculated path misses have a lower probability of being further accessed.

Table 7 presents results using a fully associative *speculative buffer*. Compared to the *victim cache* scheme, the *speculative buffer* gave mixed results. When used in conjunction with the instruction cache, the *speculative buffer* provided a clear advantage in terms of both energy and performance compared with a *victim cache* implemented with swapping. When used with a data cache, the advantage is not so apparent. Furthermore, using a *victim cache* without swapping gives overall better results compared to the speculative buffer for both instruction and data.

We also tried a variation on this algorithm deciding randomly whether to put an incoming fill either in the main cache or in the buffer. We tried, among other cases, to put randomly 10% or 20% of the fills in the buffer. It is interesting to note that the *random* case presents, on average, good results. This demonstrates that simply adding "associativity" to the main cache, suffices in eliminating most of the contention misses. This may also suggest that our benchmarks are not ideal candidates for *speculative sorting* though there may exist applications that would benefit in particular from a *speculative buffer*.

Whether using the buffers as a *victim* or *speculative buffer*, it is important to note that using these buffers not only reduces energy consumption in the overall cache memory system, but it does so without significantly increasing on-chip energy consumption. For instance, for the results listed in Table 7 we found that the on-chip energy consumption in the data portion increases on average only by 0.7%, and in the instruction portion by 2.8% (these number are not shown in the table). Even more, for some programs like *go* we reduce the on-chip data cache portion up to 8%. This is in contrast to results shown in Table 5 where on-chip power increased by as much as 23%.

**Penalty Buffer:** We observed that, as opposed to data cache behavior, there is not a fixed correlation between variations in instruction cache miss-rate and performance gain. This is due to the fact that some misses are more critical than others; fetch latency often may be hidden by the Fetch/Dispatch queue as well as the Instruction Window (Resources Reservation Unit — RUU in SimpleScalar terminology) making some instruction misses non-critical. That is, a full queue implies the processor may have a choice of instructions to execute next whereas an empty queue would imply few (if any) choices. Since misses sometimes present a burst behavior, these hardware structures can remain empty and all latency is detrimen-

Table 8: **Penalty buffer:** Percent improvement in performance and power compared to the base case. Note that the *penalty buffer* was used with the instruction cache only.

| *Test* | *Dispatch, Th=10* | | *RUU, Th=28* | |
|---|---|---|---|---|
| | *%Cycles* | *%Energy* | *%Cycles* | *%Energy* |
| compress | 0.016 | -1.037 | 0.076 | -1.224 |
| go | 6.018 | 4.259 | 6.371 | 4.599 |
| vortex | 6.518 | 5.403 | 6.292 | 4.282 |
| gcc | 4.764 | 3.944 | 4.692 | 3.878 |
| li | 6.249 | 5.696 | 6.352 | 5.881 |
| ijpeg | 2.952 | 1.067 | 2.214 | -0.940 |
| m88ksim | 18.400 | 17.174 | 24.551 | 23.140 |
| perl | 7.048 | 5.946 | 6.558 | 5.291 |

tal for performance.

The final caching strategy we considered is sorting misses on a "penalty" basis; we monitor the state (i.e. number of valid entries) of the Dispatch Queue and RUU upon a cache miss and we mark them as *critical* (Dispatch Queue or RUU with few valid entries) or *non-critical* (Dispatch Queue or RUU with many valid entries). We place *non-critical* instruction misses in the *penalty buffer* and *critical* ones in the main cache. In this way we preserve the instructions contained in the main cache that are presumed to be more critical.

We tried two different schemes in deciding when to bypass the main cache and place fills instead in the penalty buffer. In the first scheme we compared the number of Dispatch Queue or RUU valid entries to a fixed threshold. In the second scheme, we used a history-based method that saves in the main cache the state of the hardware resources at the moment of the fill (e.g. how many valid instructions we have in the RUU). This number will be compared to the current one at the moment of a future replacement. If the actual miss is more critical than the past one, it will fill the main cache, otherwise it will fill the *penalty* buffer.

Table 8 presents results using a 16-entry fully associative *penalty buffer*. Note that this scheme was applied only to the instruction cache. Columns 2,3 show results using the Dispatch Queue with a fixed threshold of 10 and columns 4-5 show results using the RUU with a threshold of 28 (these threshold values provided the best overall results). We observed some good results (in several cases we improve over the *victim* or *speculative buffer*), but when we tried different threshold values or history schemes, we found great variability in the behavior. This variability is due to the fact that in some cases looking separately at the Dispatch Queue or the RUU doesn't give a precise model of the state of the pipeline, thus creating interference in the *penalty buffer* mechanism. Nevertheless, this scheme produces better results than the *speculative buffer* scheme. We are currently investigating a combined use of these two schemes that may prove even better.

## 4 Conclusions and Future Work

In this paper we presented tradeoffs between power and performance in cache architectures, using conventional as well as innovative techniques; we showed that adding small buffers can offer an overall energy reduction along with a performance improvement in most cases. Using a buffer along with a first-level data cache can improve performance by up to 4% and energy consumption by 13%. Using a buffer along with the first-level instruction cache has been shown to improve performance by up to 25% and energy consumption by 23%. Including buffers for both data and instruction caches tend to have an additive effect. It is also important to note that using small buffers suffice in reducing the overall cache energy consumption, without a notable increase in on-chip power that traditional techniques present. Furthermore use of a buffer is not mutually exclusive to increasing the cache size/associativity, offering the possibility of a combined effect. We are currently investigating further improvements in combined techniques as well as replacement policies. In the results we showed LRU policy was adopted; we are now studying policies that may be more favorable for reducing energy consumption.

## Acknowledgments

We wish to thank Doug Burger for providing us with the framework for the SimpleScalar memory model.

## References

[1] D. Burger, and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," Technical Report TR#1342, University of Wisconsin, June 1997.

[2] D. Burger, and T. M. Austin, "SimpleScalar Tutorial ," presented at *30th International Symposium on Microarchitecture*, Research Triangle Park, NC, December, 1997.

[3] L. Gwennap, "Digital 21264 Sets New Standard," *Microprocessor Report*, October, 1996. http://www.digital.com/semiconductor/microrep/digital2.htm

[4] T. L. Johnson, and W. W. Hwu, "Run-time Adaptive Cache Hierarchy Management via Reference Analysis," *ISCA-97: ACM/IEEE International Symposium on Computer Architecture*, pp. 315–326, Denver, CO, June 1997.

[5] N. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," *ISCA-17: ACM/IEEE International Symposium on Computer Architecture*, pp. 364–373, May 1990.

[6] M. B. Kamble and K. Ghose, "Analytical Energy Dissipation Models for Low Power Caches," *ACM/IEEE International Symposium on Low-Power Electronics and Design*, August, 1997.

[7] J. Kin, M. Gupta, and W. H. Mangione-Smith, "The Filter Cache: An Energy Efficient Memory Structure," *MICRO-97: ACM/IEEE International Symposium on Microarchitecture*, pp. 184–193, Research Triangle Park, NC, December 1997.

[8] S. Manne, D. Grunwald, A. Klauser, "Pipeline Gating: Speculation Control for Energy Reduction," to appear in *ISCA-25: ACM/IEEE International Symposium on Computer Architecture*, June 1998.

[9] J. A. Rivers, and E. S. Davidson, "Reducing Conflicts in Direct-Mapped Caches with a Temporality-Based Design," *International Conference on Parallel Processing*, pp. 154-163, August 1996.

[10] C. Su, and A. Despain, "Cache Design Tradeoffs for Power and Performance Optimization: A Case Study," *ACM/IEEE International Symposium on Low-Power Design*, pp. 63–68, Dana Point, CA, 1995.

[11] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun, "Managing Data Caches using Selective Cache Line Replacement," *Journal of Parallel Programming*, Vol. 25, No. 3, pp. 213–242, June 1997.

[12] S. J. E. Wilton, and N. Jouppi, "An Enhanced Access and Cycle Time Model for On-Chip Caches," Digital WRL Research Report 93/5, July 1994.

[13] Jeffrey Gee, Mark Hill, Dinoisions Pnevmatikatos, Alan J. Smith, "Cache Performance of the SPEC Benchmark Suite," IEEE Micro, Vol. 13, Number 4, pp. 17-27 (August 1993)