Hot-and-Cold: Using Criticality in the Design of Energy-Efficient Caches

Rajeev Balasubramonian[γ], Viji Srinivasan[†], Sandhya Dwarkadas[‡], Alper Buyuktosunoglu[†]
[γ] School of Computing, University of Utah
[†] IBM T. J. Watson Research Center
[‡] Department of Computer Science, University of Rochester
Email: rajeev@cs.utah.edu

Abstract

As technology scales and processor speeds improve, power has become a first-order design constraint in all aspects of processor design. In this paper, we explore the use of criticality metrics to reduce dynamic and leakage energy within data caches. We leverage the ability to predict whether an access is in the application's critical path to partition the accesses into multiple streams. Accesses in the critical path are serviced by a high-performance (hot) cache bank. Accesses not in the critical path are serviced by a lower energy (and lower performance (cold)) cache bank. The resulting organization is a physically banked cache with different levels of energy consumption and performance in each bank. Our results demonstrate that such a classification of instructions and data across two streams can be achieved with high accuracy. Each additional cycle in the cold cache access time slows performance down by only 0.8%. However, such a partition can increase contention for cache banks and entail non-negligible hardware overhead. While prior research has effectively employed criticality metrics to reduce power in arithmetic units, our analysis shows that the success of these techniques are limited when applied to data caches.

Keywords: *Low-power microprocessors, data caches, instruction criticality.*

1 Introduction

Technology improvements resulting in increased chip density have forced power and energy consumption to be first-order design constraints in all aspects of processor design. Furthermore, in current processors a large fraction of chip area is dedicated to cache/memory structures and with each technology generation this fraction continues to grow. As a result, caches account for a significant fraction of overall chip energy. For example, in the Alpha 21264 [11], caches account for 16% of energy consumed.

In this work we focus on reducing both dynamic *and* leakage energy of L1 data caches by exploiting information on instruction criticality. Instructions of a program have data, control, and resource dependences among them. Chains of dependent instructions that determine a program's execution time are referred to as the critical paths. In other words, instructions that can be delayed for one or more cycles without affecting program completion time are considered to not be on the critical path. Such instructions, referred to as non-critical instructions, afford some degree of latency tolerance. By identifying these instructions consistently and correctly, they are directed to access a statically designed low energy and lower performance (*cold*) cache bank. Critical instructions are directed toward a (*hot*) cache bank designed for high performance. The resulting organization is a physically banked cache with different levels of energy consumption and performance in each bank.

The challenges in such an implementation are two-fold: (i) determining the bank in which to place a given piece of data, and (ii) partitioning the instruction stream into a critical and a non-critical stream. Our analysis of the data and instruction stream of a range of applications shows that the criticality of both instructions and data shows high consistency. Based on this analysis, we steer instructions to cache banks based on the instruction's program counter, and place data in these banks based on the percentage of critical instructions accessing the data line.

If the *cold* cache is designed to be highly energy-efficient (consuming 20% of the dynamic and leakage energy of the *hot* cache), we observe L1 data cache energy savings of 37%. Our results indicate that critical instruction (and data) prediction is reliable enough that performance degrades by only 0.8% for each additional cycle in the *cold* cache access time. This allows us to employ power-saving techniques within the *cold* cache, that might have dramatically degraded performance if employed in a conventional cache. However, the re-organization of data across the *hot* and *cold* banks increases contention and this degrades performance by 2.7% compared to a conventional word-interleaved cache. Hence, for the *hot-and-cold* organization to be effective, the latency cost of employing the power-saving techniques in the conventional cache has to be prohibitive. While prior work has effectively employed criticality metrics to design low-power arithmetic units [23], our results show that criticality-directed low power designs are not highly effective in L1 caches of high-performance processors.

In Section 2, we elaborate on techniques that have been proposed to address energy consumption in caches, and motivate the use of statically designed caches. In Section 3, we analyze the consistency of a program's instructions and data in terms of criticality to determine if their behavior lends itself to criticality-based classification. Section 4 describes our cache implementation. We present its performance and energy characteristics in Section 5. Finally, we conclude in Section 6.

2 Energy-Delay Trade-offs in SRAM Design

A number of circuit-level and architectural techniques can be employed to reduce dynamic and leakage energy in caches. At the circuit level, as process technology improves, careful transistor sizing can be used to reduce overall capacitance, and hence dynamic energy. Since dynamic energy is roughly proportional to the square of V_{dd}, lowering V_{dd} can help reduce dynamic energy. Simultaneous to the above circuit-level techniques, architectural techniques such as banking, serial tag and data access, and way prediction [21], help lower dynamic energy by reducing the number of transistors switched on each access. This comes at the cost of increased latency and/or complexity. For example, delay is roughly inversely proportional to V_{dd}.

Several techniques have been proposed to reduce leakage energy while minimizing performance loss [3, 4, 10, 12, 13, 16, 20, 31]. For example, higher V_t devices help reduce leakage energy [20]. However, when applied statically to the entire cache, especially the L1 cache, these techniques increase access latency. Since an L1 cache access time must typically match processor speed, an increase in access latency by even a cycle can have a significant impact on performance [12]. Circuit-level techniques that use dynamic deactivation to switch to a low leakage mode increase manufacturing cost and/or affect latency and energy consumption of the fast mode (either in steady-state or due to the transition). Our goal in this paper is to examine ways by which static low-power designs of caches might be exploited using architectural techniques to reduce *both* leakage and dynamic energy consumption while minimally affecting performance.

3 Instruction and Data Classification

From the discussion in the previous section, it is clear that the cost of decreasing energy consumption of L1 caches is an increase in average access time of the cache. One way to mitigate this performance loss is to overlap the extra access time penalty incurred due to the energy saving techniques with the execution of other instructions in the program. Such overlapping is only possible if the corresponding load instruction is not on the application critical path, where the critical path is defined by the longest chain of dependent instructions whose execution determines application completion time. We propose a technique to identify such non-critical instructions and steer data accessed by these instructions to a low energy and lower performance *cold* bank, while critical loads are still served from a fast, *hot* bank.

3.1 Criticality Metrics

Recent work [8, 9, 26, 27, 28] has examined the detection of instruction (and/or data) criticality. Srinivasan and Lebeck [27] used a simulator with roll-back capabilities to accurately classify each instruction as critical or not. More recent studies have proposed heuristics that approximate the above detailed classification method to allow feasible implementations. Srinivasan *et al.* [26] and Fisk and Bahar [9] determine that load instructions are critical if they incur a cache miss, or lead to a mispredicted branch, or slow down the issue rate while waiting for data to arrive. Fields *et al.* [8] classify an instruction as critical if it is part of a chain of successive wake-up events. Tune *et al.* [28] propose a number of heuristics that predict whether instructions are critical or not. For example, their analysis shows that treating the oldest instructions in the issue queue as critical performs as well as other more complicated metrics that use data dependence chain information to determine criticality.

Our analysis also confirms that using the position of the instruction in the issue queue to determine its criticality performs comparably to techniques that use more complicated metrics. Using this *Oldest-N* technique, an instruction is deemed critical if it is among the oldest N instructions in the issue queue at issue time, where N is a pre-defined parameter. Since ready instructions that are further downstream (not among the oldest N) have a greater degree of latency tolerance, it is fair to mark them as non-critical. Note that such a heuristic tends to identify instructions along mispredicted paths as being non-critical, because mispredicted instructions are usually not the oldest instructions in the issue queue. The following advantages motivate the use of *Oldest-N* in our design: (i) No hardware table is required to predict instructions as critical because the position in the issue queue at the time of issue is sufficient to determine criticality. (ii) The ratio of critical to non-critical instructions can be tuned by varying N.

3.2 Classifying Load Instructions

This subsection attempts to quantify the consistency of criticality behavior of load instructions with the *Oldest-N* metric. Similar results were seen when employing other complex criticality metrics. In our analysis, $N = 5$ allows the most accurate classification of instructions as critical or not – in other words, this set of instructions yielded minimal performance impact when slowed by a cycle. Figure 1 shows a histogram of the percentage of loads that show the same criticality behavior as their last dynamic invocation.

In Figure 1, we observe that for most of the applications, over 85% of dynamic loads have the same criticality as their previous invocation. Only *gap* (80%) and *twolf* (84%) have a slightly lower degree of consistency. On average, over 88% of loads show consistent criticality behavior. The high consistency exhibited by loads motivates the use of hardware predictors to partition accesses into critical and non-critical streams.

3.3 Classifying Data Blocks

While the results in the previous subsection reveal that instructions can be statically categorized as critical or non-critical, the same behavior need not hold true for accessed data blocks. A single cache line is accessed by a number of loads and stores, not all of which may have the same criticality behavior. In order to be able to place data in either

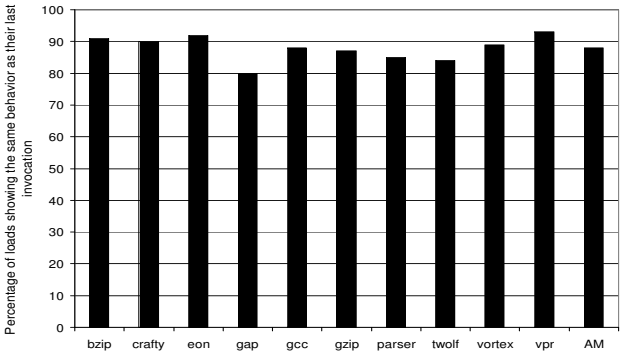

Figure 1. Consistency of Load Criticality

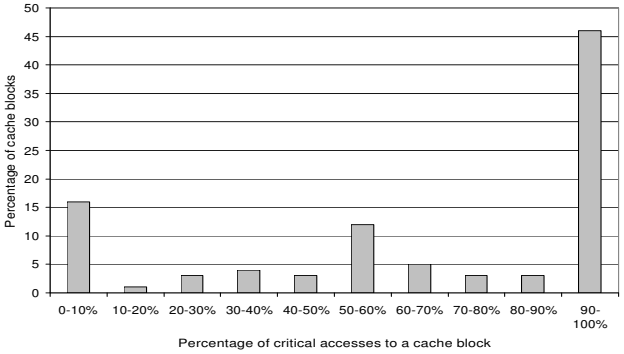

Figure 2. Percentage of critical accesses to a data cache block. The figure is a histogram showing the percentage of data blocks that had a particular fraction of accesses from critical loads.

the hot or cold cache bank, we have to determine if the accesses to cache blocks are dominated by either critical or non-critical loads/stores. Figure 2 shows a histogram of the distribution of data cache blocks based on the percentage of accesses to each block that were attributable to critical loads. This is computed by averaging the histograms for each individual program. From Figure 2, we observe that nearly 46% of data cache blocks have 90 to 100% of their accesses from critical loads. Similarly, 16% of data blocks have only 0 to 10% of their accesses from critical loads (i.e., over 90% of their accesses are due to non-critical memory operations). These results indicate that data blocks are also often exclusively critical or non-critical. However, there is a non-trivial percentage of blocks that are accessed equally by critical and non-critical loads, and steering such data blocks to one of the cache banks will impact performance and/or energy. When using smaller cache line sizes, blocks are more strongly polarized as being critical or non-critical. Although smaller lines reduce interference from access behaviors of other words in a line, there still remain a number of words that are accessed equally by critical and non-critical loads/stores.

4 The Hot-and-Cold Banked Cache

Proposed Organization. Motivated by the results in Section 3, we propose a new organization for the L1 data cache that is composed of multiple cache banks, with some

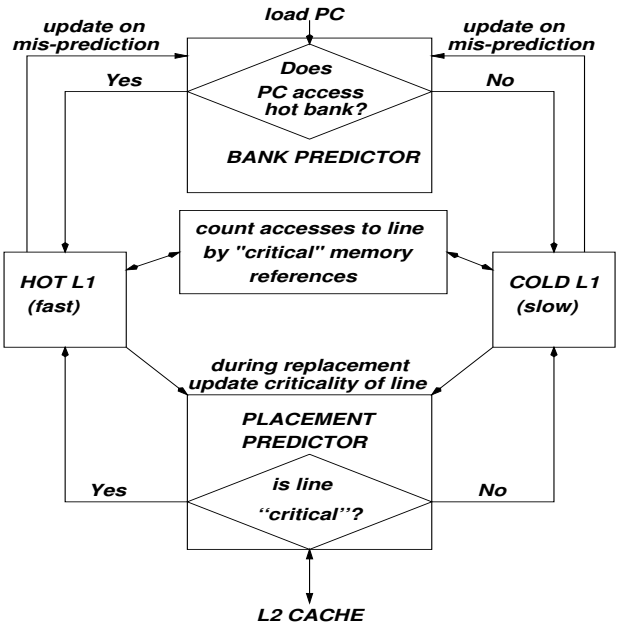

Figure 3. The Energy-Aware Hot-and-Cold Banked Cache

banks being energy-efficient and the rest being designed for the fastest possible access time. Figure 3 shows a high-level block diagram of the proposed *hot-and-cold* L1 data cache. The L1 data cache is split into two banks – a "hot" bank and a "cold" bank. The hot bank is slated to contain data blocks that are marked critical, while the cold bank is slated to contain data blocks that are marked non-critical.

The *hot* cache provides the fastest possible access time, and the *cold* cache services requests in an energy-efficient manner while incurring a longer latency. Since the *cold* bank is similar to other conventional cache banks, it could use any proposed architectural energy-saving techniques like serial tag-data lookup, or way prediction [21] to reduce energy per access. In addition, the cold cache can be designed with more energy-efficient circuits using transistor sizing (as discussed in section 2) to reduce overall capacitance, high V_t for reduced leakage, or gated-ground SRAM cells [3].

Data Placement using Placement Predictor. Every time a cache block is fetched from the L2 cache, it is placed into one of the hot or cold banks based on the history of accesses to that block when it last resided in the L1 cache. For this purpose, we track the fraction of accesses to each cache block due to critical loads/stores. As explained in the last section, we use the *Oldest-N* metric to determine whether a load instruction is critical or not. For each data block in the L1 cache we maintain an n-bit up/down counter, initialized to 2^{n-1} to track the number of accesses to that block due to critical memory access instructions. The counter is incremented for every critical access to the block and decremented for every non-critical access. A 4-bit counter was chosen to avoid miscategorization of a cache line as either critical or non-critical.

When a data block is being replaced from the L1 cache, we mark the corresponding line as "critical" if the counter is greater than or equal to 2^{n-1}. Else, it is marked "non-critical". To save this information, we could use 1 extra bit (the "criticality" bit) per line in the L2 cache directory. Subsequently, when the data block is brought back into the L1 cache from L2, it is placed in the hot cache if the criticality bit of that line in the L2 cache is set to 1; otherwise, it is placed in the cold cache. The criticality bit of all lines in L2 are initialized to 1. As the criticality bit in an L2 cache line stores the most recent classification of the cache block, the placement of blocks in the hot and cold cache adapts dynamically depending on the change in access patterns to a cache block. Moreover, updates to the criticality bit are not in the critical path because they are performed only when a block is replaced from the L1 cache.

In spite of the low overhead for storing this 1 bit in the L2 cache, it is desirable to eliminate it for the following reasons: (i) The bit has to be communicated to L2 even if the block being replaced is not dirty. (ii) Despite having a bit per L2 cache line, the bit is lost if the cache block is evicted from L2. Hence, we explored the use of a separate structure (the *placement* predictor) that stores these bits for a subset of cache blocks. Every time a block is evicted from L1, the block address is used to index into this structure and update the classification of that block using the criticality information for the line. The bit is set to 1 if the line is critical, and to 0 if the line is non-critical. Every time a block is brought in from L2, the structure is indexed to determine the classification and place data in either the *hot* or the *cold* bank. We found that a structure of size as small as 4K bits was enough to accurately classify blocks as being critical and non-critical. Using a separate structure avoids having the size of the table grow in proportion to the L2.

Load/Store Steering Using Bank Predictor. The above mechanism partitions data blocks across two streams. Next, we have to partition memory operations and steer them to the appropriate cache bank. For each load/store instruction, we use a predictor indexed by the instruction's PC to steer the access to the bank that contains the data being accessed. Note that this steering does not take into account the criticality nature of the instruction itself – because a critical load can access a block that is classified as non-critical and a non-critical instruction can access a block that is classified as critical.

We maintain a hardware-based dynamically updated bank predictor that keeps track of the bank that was last accessed by a particular instruction. We experimentally observed that a bank predictor of size 4Kx1 bit was sufficient to steer memory accesses with an average accuracy of greater than 90%. Moreover, a PC-based predictor to steer accesses to hot and cold banks allows the steering to be done as soon as a load/store instruction is decoded. Hence, selecting between banks does not incur any additional cycle time penalty. The predictor contains a bit for each entry. If the value of the bit is one, the access is steered to the hot cache, and if the value is zero, the access is steered to the cold cache. The counter value is set to one if the data is found in the hot cache and reset to 0 if it is found in the cold cache.

During every access, tags for both banks are accessed simultaneously. This allows us to detect a steering misprediction. For each such misprediction, we incur additional performance and energy penalty for probing both the hot and the cold cache array. This need not introduce additional complexities in the issue logic as the operation is similar to the load replay operation on a cache miss. However, such bank mispredicts and the resulting replays can significantly impact performance and energy inefficiency [17]. The variable latency across different loads can also be handled – at the time of issue, the bank being accessed (and hence, its latency) is known, and wake-up operations can accordingly be scheduled.

Related Work. The *hot-and-cold* cache has a read bandwidth of two with a single read port per bank because in any given cycle one hot and one cold word can be accessed. Thus, it behaves like a two-banked cache in which the bank steering is done based on the criticality nature of cache blocks. We therefore compare the performance of the *hot-and-cold* cache with a word-interleaved two-banked cache. Rivers *et al.* [15] show that partitioning the cache in a word-interleaved manner minimizes bank conflicts because most applications have an even distribution of accesses to odd and even words.

Recent work by Abella and Gonzalez [2] examines a split cache organization with a fast and slow cache. Contrary to our proposal, their policies for data placement and instruction steering are both based on the criticality nature of the accessing instruction. There is also considerable performance-centric related work that has looked at cache partitioning. For example, the MRU [25], Victim [14], NTS [22], and Assist [18] caches, different forms of buffering [30], and the Stack Value File [19] study various ways to partition the cache to improve average cache access latency. The focus of our work is quite different; the main motivation for the design choices we explored in this paper is to provide faster access to a subset of cache lines while saving energy when accessing the rest of the lines.

5 Results

5.1 Methodology

To evaluate our design, we use a simulator based on Simplescalar-3.0 [5] for the Alpha AXP instruction set. The register update unit (RUU) is decomposed into integer and floating point issue queues, physical register files, and reorder buffer (ROB). The memory hierarchy is modeled in detail, including word-interleaved access, bus and port contention, and writeback buffers. The important simulation parameters are summarized in Table 1.

We estimated power for different cache organizations using CACTI-3.0 [24] at 0.1μm technology. CACTI uses an

Fetch queue size	16
Branch predictor	comb. of bimodal and 2-level
Bimodal predictor size	2048
Level 1 predictor	1024 entries, history 10
Level 2 predictor	4096 entries
BTB size	2048 sets, 2-way
Branch mpred penalty	at least 12 cycles
Fetch width	4 (across up to two basic blocks)
Dispatch, commit width	4
Issue queue size	20 (int and fp, each)
Register file size	80 (int and fp, each)
Re-order Buffer	80
Integer ALUs/mult-div	4/2
FP ALUs/mult-div	4/2
L1 I and D cache	16KB 2-way, 2 cycles, 32 byte line
L2 unified cache	2MB 8-way, 16 cycles, 64 byte line
TLB	128 entries, 8KB page size
Memory latency	90 cycles for the first chunk

Table 1. Simplescalar Simulation Parameters.

Benchmark	Base IPC	L1 miss rate	L2 miss rate
bzip	1.42	2.96	5.22
crafty	1.32	4.37	0.16
eon	1.60	0.90	0.04
gap	1.68	0.62	17.12
gcc	1.18	9.98	0.24
gzip	1.37	2.84	1.37
parser	1.20	4.77	4.83
twolf	1.09	9.51	0.11
vortex	1.14	2.71	0.92
vpr	1.02	4.38	10.65

Table 2. Base Statistics for Benchmarks Used.

analytical model to estimate delay and power for tag and data paths. We obtained an energy per read access of 0.67 nJ for a two-banked, 16KB, 2-way associative L1 cache with a 32 byte line size. Of this, 0.56 nJ was due to bit-lines and sense amplifiers. For write accesses the cache essentially behaves like a 1-way cache, and 50% of bit-line and sense amplifier energy can be eliminated. Hence, energy per write access is 0.39 nJ ($0.67 - 0.5 * 0.56$). We also take into account the energy cost of reading and writing entire cache lines during writeback and fetch. Note that our evaluations only show energy consumed within the L1 data cache and not overall processor energy. The contribution of the data cache to total processor power can be as little as 5% in a high-performance processor and as much as 50% in an embedded processor. In future technologies, increased leakage energy is likely to increase the contribution of the L1D to total chip power. Given the wide range of possible values, we restrict ourselves to presenting the savings in data cache energy only.

We analyzed additional energy expended due to the hardware counters and predictors used in the *hot-and-cold* cache. With CACTI, we derived energy per access of the tag array for the L1 and L2 cache to be 0.055 nJ and 0.162 nJ, respectively. Based on this, we derived the energy per access due to the 4-bit counter used in the L1 tag array (for tracking critical/non-critical accesses of a cache block) to be 0.004 nJ (which is only an additional 1% energy per access for the L1 cache). Similarly, we estimated the additional energy per access to the *placement* predictor to be 0.5 pJ, which is a negligible fraction per L2 access (since the predictor is updated and accessed only on an L1 miss or eviction). The energy for the *bank* predictor used for steering memory accesses to the hot or cold bank is 0.5 pJ per access (same size as the *placement* predictor), which is again a negligible fraction of L1 data cache energy per access.

We simulated 10 programs from SPEC2k-Int[1] with reference input datasets. The simulation was fast-forwarded 2 billion instructions, another 1 million instructions were used to warm up processor state, and the next 500M instructions were simulated in detail. Table 2 presents the benchmarks with their base IPC and L1 and L2 miss rates.

5.2 Comparison of Performance

Figure 4 compares performance of the hot-and-cold cache to the baseline L1 data cache, which is dual banked and word interleaved. The first bar presents IPC of the baseline L1 data cache. The second bar shows IPC for the hot-and-cold cache with data allocation to banks based on criticality, assuming perfect steering (no bank prediction) of loads/stores to banks. The criticality metric used is *Oldest-N*, where *N* is fixed at 7.

By using the *hot-and-cold* organization, allocation of data across the two banks is different. As a result, overall IPC is reduced by about 1.8%, which is a result of two factors: (i) There is an imbalance in the number of accesses to each bank while using the hot-and-cold cache. Some applications (*bzip, gzip, twolf*) have many more critical accesses, while others (*gcc, vortex*) have many more non-critical accesses. This results in excess contention for the limited cache ports as compared to the word-interleaved banked cache, in which, accesses to each bank are roughly equal. (ii) Accesses to a critical or non-critical cache are more bursty. When an instruction completes and wakes up its dependents, there is a high probability that the dependents would either all be critical or all be non-critical. Since an entire cache line resides in one bank, spatial and temporal locality also dictate that the same bank would be repeatedly accessed. We verified this by examining the number of accesses to each bank in a 10-cycle window. Figure 5 shows a histogram indicating the percentage of such windows that encountered a particular ratio of accesses to the two banks (using an *Oldest-7* threshold for the *hot-and-cold* cache). For the word-interleaved cache, an average of 36% of all time windows had roughly the same number of accesses to each bank. For the *hot-and-cold* cache, this number was only 19%, while the number of windows that had exclusively either critical or non-critical accesses was as high as 26%. This shows that reorganizing data in the banks based on criticality results in an increase in data cache port contention.

While we cannot completely eliminate the bursty nature of critical and non-critical accesses since this is in-

[1] *Perlbmk* did not run with our simulator and *mcf* is too memory bound for its performance to be affected by changes to the CPU and cache.

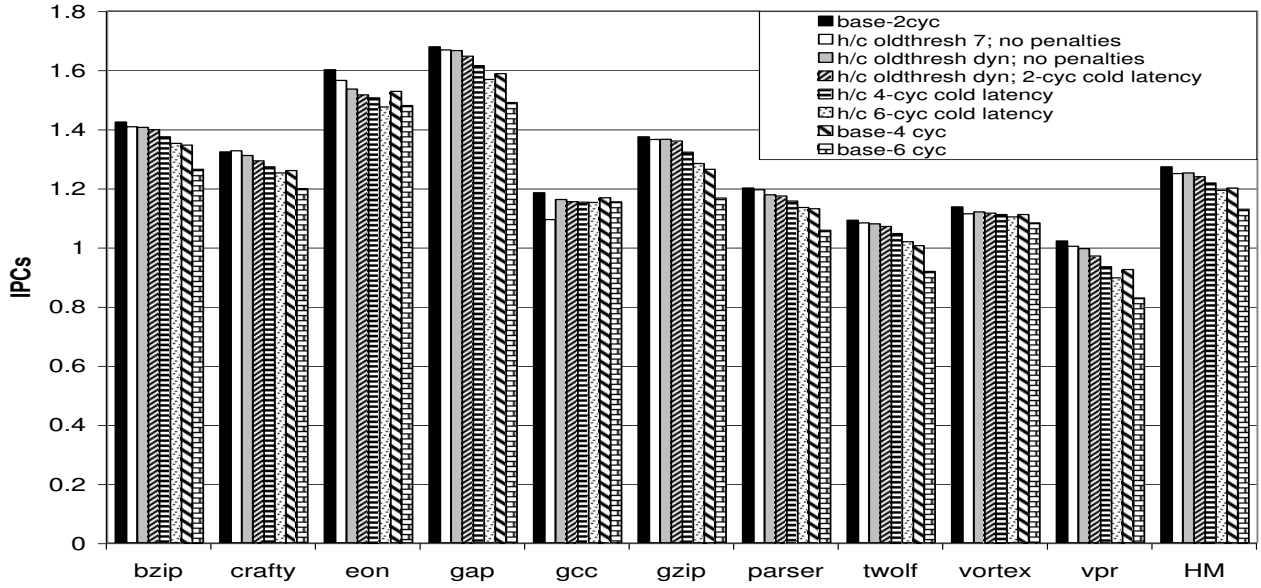

Figure 4. Performance of Hot-and-Cold Cache Relative to Baseline Word-Interleaved

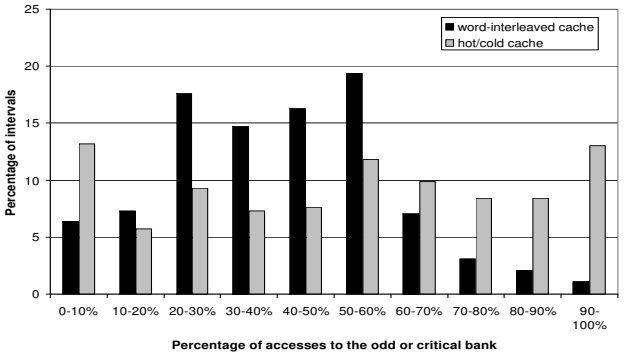

Figure 5. Histogram showing the percentage of 10-cycle intervals that experienced a particular ratio of accesses to the two banks (using *Oldest-7* for the *hot-and-cold* cache).

herent to program behavior, we used the following method to improve the distribution of accesses to the banks. By varying the parameter N, we changed the number of critical accesses and thus, the allocation of blocks to the two banks. We found that keeping the number of accesses to each bank roughly equal resulted in the least amount of port contention. This is achieved at run-time with a simple mechanism that uses statistics over a past interval to determine the value of N for the next interval. We used 10M instruction intervals and discarded statistics from the first half of an interval. Over the latter half of the interval, we counted the number of accesses to each bank. In terms of hardware, this requires two counters to keep track of the number of accesses to the two banks as well as a comparison triggered every 10M instructions. If the percentage of accesses to the hot bank was less than 45%, we increased the value of N so as to classify more loads as critical. Likewise, if the percentage of accesses to the hot bank was more than 55%, we decreased N. The statistics from the first half of each 10M instruction interval are discarded to allow enough cache block evictions and fetches that the parameter change is reflected in statistics collected for the latter half. Striving for a 50% share of critical accesses minimizes port contention, improving IPC by 1%. However, we found that keeping the share of critical accesses to 60% was better at minimizing IPC loss from a slower cold cache. The third bar in Figure 4 represents such an organization and is only marginally better than a fixed value of $N = 7$. However, since it classifies more instructions as critical, it sees a much lower performance degradation when cold cache latency is increased. Note that this represents a model where loads and stores are perfectly steered to the bank that caches the data. *Gcc* is an example of a program that is highly constrained by cache port contention. Using a value of $N = 7$ causes a high imbalance in accesses to each bank and degrades performance. As a result, the dynamic tuning of N is very important in this case to minimize additional port contention stalls causes by the *hot-and-cold* cache organization.

The fourth bar in the figure shows a model that uses the *bank* predictor. An incorrect prediction results in a probe of both the banks, resulting in a higher latency and greater port contention. Due to this, overall IPC goes down by an additional 1%. We found that the mispredict rate was 9.5% on average, thus confirming our earlier hypothesis about the easy predictability of the nature of blocks accessed by loads and stores. Table 3 shows various statistics that help us explain the changes in performance. The table shows that the hot-and-cold cache organization has to handle more accesses and this increase is caused by mispredictions while steering loads and stores. We also note that the distribution of accesses to odd and even banks in the base case is fairly even (except in *gcc*). By tuning the value of N, the distribution of accesses to hot and cold banks is adjusted to approximately be in the ratio 60:40. The notable exception is *vortex*, where most instructions issue from the last issue queue entry and are classified as non-critical. Because of

Benchmark	Base case Total accesses	bank 1:bank 2 Accesses	Stalls due to port contention	Hot-and-cold Total accesses	Hot:Cold Accesses	Stalls due to port contention	Steering mispredictions
bzip	191M	60:40	41.6M	198M	62:38	103.6M	7.1M
crafty	194M	58:42	59.1M	216M	59:41	127.1M	21.1M
eon	232M	56:44	76.8M	263M	59:41	231.3M	33.0M
gap	183M	57:43	51.6M	213M	60:40	106.0M	30.2M
gcc	345M	76:24	1754M	354M	53:47	1894M	8.7M
gzip	177M	62:38	80.0M	185M	58:42	117.8M	7.9M
parser	175M	62:38	62.6M	187M	59:41	106.1M	12.3M
twolf	173M	58:42	74.1M	194M	62:38	94.4M	24.6M
vortex	218M	63:37	176.4M	241M	36:64	267.7M	24.3M
vpr	210M	51:49	59.1M	238M	58:42	134.5M	29.8M

Table 3. Data access statistics for the *hot-and-cold* using a dynamic *Oldest-N* threshold and for the base word-interleaved cache

the increased number of accesses to the *hot-and-cold* cache and the inherent bursty nature of these accesses (Fig 5), we see that the number of stall cycles due to port contention is much higher. On average, the hot-and-cold cache has twice as many stall cycles as the word-interleaved base case (In *gcc*, the program is already highly constrained by port contention, so the increase caused by the *hot-and-cold* reorganization does not result in a doubling of the number of stall cycles.).

Finally, the fifth and sixth bars show the effect of increasing the cold cache latency to four and six cycles, respectively. The seventh and eighth bars show IPCs for a word-interleaved cache where all accesses take four and six cycles, respectively. In spite of slowing down as many as 45% of all memory operations, increasing cache latency from 2 to 4 cycles only results in a 1.6% IPC loss. Uniformly increasing the latency of every access to 4 cycles in the base case results in an IPC penalty of 5.7%. Thus, the use of the criticality metric helps restrict the IPC penalty of a slower cache access time. However, owing to the penalty from increased port contention and mis-steers, the hot-and-cold cache with the 4-cycle cold cache latency does only marginally better than the 4-cycle word-interleaved cache. The value of the proposed organization is seen when the energy-saving techniques threaten to slow the cache to a latency of six cycles. The hot-and-cold organization with the 6-cycle cold cache latency outperforms the 6-cycle base case by an overall 5.7%, demonstrating its ability to tolerate the additional latency.

As we demonstrated in Figure 2, there are a number of blocks that are not easily classifiable as critical or non-critical. This causes some amount of inefficiency in the system – when critical loads access non-critical blocks, performance is lost, and when non-critical operations access critical blocks, they unnecessarily consume additional energy. We noticed that 26% of all memory operations were of this kind. This inefficiency cannot be microarchitecturally eliminated as it is an artifact of the program that the same data is accessed by different kinds of loads and stores.

Finally, it could be argued that a word-interleaved cache like the base case with half the total capacity could match the energy consumption of the *hot-and-cold* cache if the base capacity is not fully utilized. We make the assumption (which we verified for our base case design parameters) that the capacity of the base case is chosen to work well across most programs and that halving its capacity would severely impact a number of programs, rendering such an organization unattractive.

5.3 Comparison of Energy

5.3.1 Energy-Delay Trade-Offs

Our proposal does not make any assumption on the particular energy-saving techniques that can be employed for the cold bank. Since the cold bank is like any other conventional cache bank, it could use any of the already proposed architecture or circuit level power-saving techniques. Hence, our results are parameterized across multiple access time and energy consumption characteristics.

In order to provide a more detailed understanding of energy-delay trade-offs possible through circuit-level tuning, we performed circuit simulations using a typical SRAM cross-section from the predecoder to the wordline driver in 0.13μ CMOS technology. The predecoder consists of 3-to-8 NORs and the decoder is an n-input NAND, where n is the number of 3-to-8 predecode blocks. Finally, the wordline drivers consist of inverters that drive the load, which includes all associated wires and SRAM cells. We used a formal static tuning tool, EinsTuner [6], to vary total device width. EinsTuner [29, 6] is built on top of a static transistor-level timing tool (EinsTLT) that combines a fast event-driven simulator (SPECS) with a timing tool (Einstimer). The SPECS simulator provides timing information such as delay and slew along with first derivatives with respect to transistor width. EinsTuner uses this information to formulate the optimization problem as a linear combination of slack and area. This formulation is then solved by a non-linear optimization package LANCELOT [7], which treats all device widths as free parameters and solves for minimum delay. Finally, energy values are obtained using AS/X circuit simulations [1] for a given switching activity.

Figure 6 shows normalized energy-delay trade-offs for an SRAM cross-section. The primary y-axis shows delay (*norm.delay*) corresponding to a given energy consumption. The secondary y-axis shows the normalized sum of transistor widths (*norm.sumw*). For this experiment, we used parameters such as beta constraint (i.e. PMOS/NMOS width ratio), internal slew rate, primary output slew rate, and input

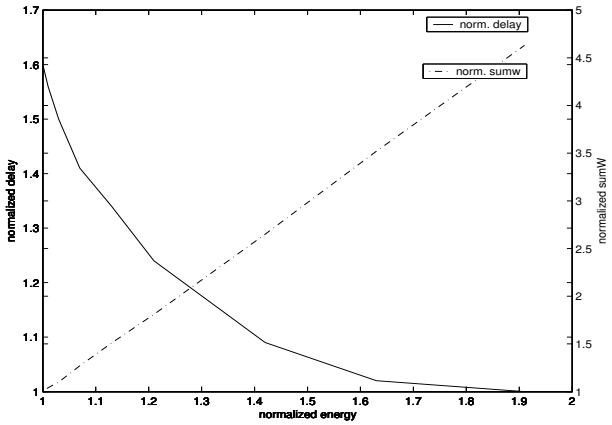

Figure 6. Energy-Delay Curve for SRAM Cross-section. The dotted line plots the normalized sum of transistor widths against the normalized energy consumption. The solid line plots the normalized delay against the normalized energy consumption.

V_t	Normalized Leakage Energy	Normalized Delay
low	8.5	0.88
nominal	1	1
high	0.23	1.34

Table 4. Leakage Energy-Delay Trade-offs for Different V_t

capacitance/stage constraints from real designs. The initial data point for this experiment is obtained by minimizing delay without area constraints, and the other points show minimum delay with specific area constraints. From Figure 6, we observe that by increasing minimum delay by 60%, we can achieve up to 48% reduction in energy.

Table 4 shows normalized leakage energy-delay trade-offs for the same SRAM cross-section using transistor widths that correspond to minimum delay. We observe that decreasing V_t increases leakage energy dramatically (8 times), while increasing V_t decreases leakage energy by 77% at the expense of 34% increase in delay.

5.3.2 Dynamic and Leakage Energy Savings

As discussed in Section 2, many techniques can be employed to reduce cache energy, including transistor sizing, lowered V_{dd}, banking, serial tag and data access, higher V_t, etc. Since any of the above techniques can be applied to the *cold* bank, we present results for energy savings assuming the cold bank consumes either 0.2 or 0.6 times the dynamic and leakage energy consumed by the hot bank. Results for using just one of the circuit-level techniques — transistor sizing from Figure 6 — would lie in between (corresponding to roughly 0.5 time the dynamic energy consumed by the hot bank, with roughly twice the access latency in cycles). When designing with a higher V_t for the cold bank, the 77% reduction in leakage energy shown in Table 4 corresponds roughly to 0.2 times the leakage consumed within the hot bank with roughly a 1 cycle increase in delay for our cache organization. The use of multiple techniques could potentially bring more aggressive energy savings at a potentially higher access penalty, justifying our choice of range

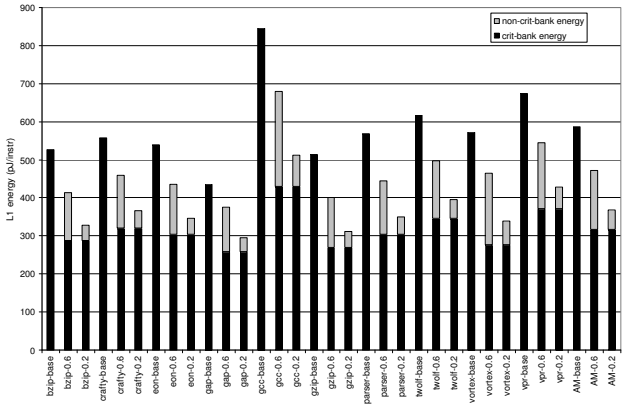

Figure 7. L1 data cache energy (both leakage and dynamic). The first bar represents a word-interleaved base case. The second and third bars represent a hot-and-cold cache organization, where the dynamic and leakage energy within the cold bank are 0.6 and 0.2 times the energy within the hot bank, respectively. For the hot-and-cold cache, the black and grey portions of the bars represent the energy consumed within the hot and cold banks, respectively.

in terms of energy savings (40% to 80% of the base case) and access penalty (1.5 to 3 times the base case).

Figure 7 shows potential energy savings from the proposed organization. For each program, we show L1 data cache energy for three organizations - (i) word-interleaved base case, (ii) hot-and-cold organization, where the hot bank has characteristics identical to a bank of the base case, and the cold bank consumes 0.6 times the dynamic and leakage energy consumed by the hot bank, (iii) hot-and-cold organization, where the hot bank is the same and the cold bank consumes 0.2 times the energy consumed by the hot bank. For the hot-and-cold cache, the figure also shows the contribution to total energy from the two banks. Since 45% of all accesses are steered to the cold bank, that number serves as an approximate upper bound to the potential energy savings.

By having a highly energy-efficient cold bank (like that represented by the third bar in the figure), the energy consumption in the data cache reduces by an average of 37%. There is little effect on L2 energy consumption since the miss rates of the two caches are comparable. Leakage energy consumed is a function of total execution time (which is slightly longer for the *hot-and-cold* cache). We observed that the contribution to total energy savings came equally from dynamic and leakage components.

Note that energy savings can be further increased by improving steering prediction accuracy. Our results take into account the additional energy overhead of a steering misprediction – about 10% of all loads and stores access both banks. Also note that the distribution of accesses across different banks is almost the same in all programs, resulting in very little variation in energy trends across the benchmark set.

6 Conclusion

We have presented and evaluated the design of a banked cache, where each bank can be fixed at design time to be either *hot* or *cold*, i.e., high energy and low latency, or low energy and high latency, respectively. The performance impact of accessing the *cold* cache can be minimized effectively by separating load and store instruction streams into a critical and non-critical stream. Our results demonstrate that performance impact is reasonably insensitive to the latency of the *cold* bank, allowing aggressive power reduction techniques. This is made possible by the consistent classification of instructions and data as critical and non-critical streams. Each additional cycle in the *cold* cache latency impacts performance by about 0.8%. Energy savings are proportional to the fraction of accesses to the *cold* cache, with L1 energy reduction being an average of 37% (compared to a word-interleaved base case) for an energy-efficient *cold* bank.

However, allocation of data blocks in the *hot* and *cold* banks increases contention and introduces bank steering mispredicts. This results in an IPC degradation of 2.7%, compared to a word-interleaved conventional cache, that severely limits the effectiveness of this approach. To alleviate these problems, bank prediction would have to be improved or base cases with low contention would have to be considered. Such problems were not encountered in the design of criticality-based arithmetic units with different power/performance characteristics [23]. The *hot-and-cold* cache becomes more effective when the cost of employing any power-saving technique becomes prohibitive. For example, a *hot-and-cold* organization with a 2-cycle *hot* latency and a 6-cycle *cold* latency outperforms a 6-cycle word-interleaved base case by 5.7%.

We are working with circuit designers in order to help define the energy consumption ratio between hot and cold cache banks. Our initial analysis reveals that simple techniques like transistor sizing and high V_t can dramatically reduce dynamic and leakage energy consumption, validating the choice of parameters in our evaluation. We also plan to evaluate the use of asymmetric sizes (and organizations) for hot and cold banks.

References

[1] *AS/X User's Guide, IBM Corporation, New York.* 1996.

[2] J. Abella and A. Gonzalez. Power Efficient Data Cache Designs. In *Proceedings of ICCD-21*, Oct 2003.

[3] A. Agarwal, H. Li, and K. Roy. DRG-cache: A data retention gated-ground cache for low power. In *Proceedings of the 39th Conference on Design Automation*, June 2002.

[4] N. Azizi, F. Najm, and A. Moshovos. Low Leakage Asymmetric-Cell SRAM. In *Proceedings of ISLPED*, Aug 2002.

[5] D. Burger and T. Austin. The Simplescalar Toolset, Version 2.0. Technical Report TR-97-1342, University of Wisconsin-Madison, June 1997.

[6] A. R. Conn, I. M. Elfadel, W. W. Molzen, P. R. O'Brien, P. N. Strenski, C. Visweswariah, and C. B. Whan. Gradient-based optimization of custom circuits using a static-timing formulation. In *Proceedings of Design Automation Conference*, pages 452–459, June 1999.

[7] A. R. Conn, N. I. M. Gould, and P. L. Toint. *LANCELOT: A Fortran Package for Large-Scale Non-Linear Optimization (Release A)*. Springer Verlag, 1992.

[8] B. Fields, S. Rubin, and R. Bodik. Focusing Processor Policies via Critical-Path Prediction. In *Proceedings of ISCA-28*, July 2001.

[9] B. Fisk and I. Bahar. The Non-Critical Buffer: Using Load Latency Tolerance to Improve Data Cache Efficiency. *IEEE International Conference on Computer Design*, October 1999.

[10] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy Caches: Simple Techniques for Reducing Leakage Power. *29th Annual International Symposium on Computer Architecture*, May 2002.

[11] M. Gowan, L. Biro, and D.Jackson. Power Considerations in the Design of the alpha 21264 Microprocessor. In *35th Design Authomation Conference*, pages 726–731, June 1998.

[12] H. Hanson, M. S. Hrishikesh, V. Agarwal, S. W. Keckler, and D. Burger. Static Energy Reduction Techniques for Microprocessor Caches. *2001 International Conference on Computer Design*, September 2001.

[13] S. Heo, K. Barr, M. Hampton, and K. Asanovic. Dynamic Fine-Grain Leakage Reduction Using Leakage-Biased Bitlines. *29th Annual International Symposium of Computer Architecture*, May 2002.

[14] N. P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the 17th International Symposium on Computer Architecture (ISCA-17)*, pages 364–373, May 1990.

[15] J.Rivers, G. S. Tyson, E. Davidson, and T. Austin. On High-Bandwidth Data Cache Design for Multi-Issue Processors. In *Proceedings of the 30th International Symposium on Microarchitecture*, Dec. 1997.

[16] S. Kaxiras, Z. Hu, and M. Martonosi. Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power. *28th Annual International Symposium on Computer Architecture*, June 2001.

[17] S. Kim, N. Vijaykrishnan, M. Irwin, and L. John. On Load Latency in Low-Power Caches. In *Proceedings of ISLPED*, Aug 2003.

[18] G. Kurpanek, K. Chan, J. Zheng, E. DeLano, and W. Bryg. Pa7200: A pa-risc processor with integrated high performance mp bus interface. *COMPCON Digest of Papers*, pages 375–382, 1994.

[19] H. Lee, M. Smelyanskiy, C. Newburn, and G. S. Tyson. Stack Value File: Custom Microarchitecture for the Stack. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, pages 5–14, Jan. 2001.

[20] K. Nii, H. Makino, Y. Tujihashi, C. Morishima, Y. Hayakawa, H. Nunogami, T. Arakawa, and H. Hamano. A low power sram using auto-backgate-controlled MT-CMOS. In *International Symposium on Low-Power Electronics and Design*, 1998.

[21] M. Powell, A. Agrawal, T. N. Vijaykumar, B. Falsafi, and K. Roy. Reducing set-associative cache energy via selective direct-mapping and way prediction . *34th Annual International Symposium on Microarchitecture*, December 2001.

[22] J. A. Rivers and E. S. Davidson. Reducing Conflicts in Direct-Mapped Caches with a Temporality-Based Design. In *Proceedings of the 1996 International Conference on Parallel Processing*, pages 151–162, Aug. 1996.

[23] J. Seng, E. Tune, D. Tullsen, and G. Cai. Reducing Processor Power with Critical Path Prediction. In *Proceedings of MICRO-34*, Dec 2001.

[24] P. Shivakumar and N. P. Jouppi. CACTI 3.0: An Integrated Cache Timing, Power, and Area Model. Technical Report TN-2001/2, Compaq Western Research Laboratory, August 2001.

[25] K. So and R. Rechtschaffen. Cache operations by mru change. *IBM Technical Report, RC-11613*, 1985.

[26] S. T. Srinivasan, R. D. Ju, A. R. Lebeck, and C. Wilkerson. Locality vs. criticality. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, June 2001.

[27] S. T. Srinivasan and A. R. Lebeck. Load Latency Tolerance in Dynamically Scheduled Processors. *Journal of Instruction-Level Parallelism*, 1, Oct 1999.

[28] E. Tune, D. Liang, D. Tullsen, and B. Calder. Dynamic Prediction of Critical Path Instructions. In *Proceedings of HPCA-7*, Jan 2001.

[29] C. Visweswariah and A. R. Conn. Formulation of static circuit optimization with reduced size, degeneracy and redundancy by timing graph manipulation. In *IEEE International Conference on Computer-Aided Design*, pages 244–251, November 1999.

[30] K. Wilson, K. Olukotun, and M. Rosenblum. Increasing Cache Port Efficiency for Dynamic Superscalar Microprocessors. In *Proceedings of the 23rd ISCA*, May 1996.

[31] S.-H. Yang, M. D. Powell, B. Falsafi, K. Roy, and T. N. Vijaykumar. An Integrated Circuit/Architecture Approach to Reducing Leakage in Deep-Submicron High Performance I-Caches. *Seventh International Symposium on High-Performance Computer Architecture*, January 2001.