University of Maryland Technical Report UMIACS-TR-2012-11

Multi-Level Cache Resizing

Inseok Choi and Donald Yeung
Department of Electrical and Computer Engineering
University of Maryland at College Park
{inseok, yeung}@umd.edu

Abstract

Hardware designers are constantly looking for ways to squeeze waste out of architectures to achieve better power efficiency. Cache resizing is a technique that can remove wasteful power consumption in caches. The idea is to determine the minimum cache a program needs to run at near-peak performance, and then reconfigure the cache to implement this efficient capacity. While there has been significant previous work on cache resizing, existing techniques have focused on controlling resizing for a single level of cache only. This sacrifices significant opportunities for power savings in modern CPU hierarchies which routinely employ 3 levels of cache.

This paper investigates multi-level cache resizing *(MCR). MCR independently resizes all caches in a modern cache hierarchy to minimize dynamic and static power consumption at all caching levels simultaneously. Specifically, we study a static-optimal version of MCR, and find resizing a 3-level hierarchy can reduce total energy dissipation by 58.9% with only 4.4% degradation in performance. Our study shows a non-trivial portion of this gain–$\frac{1}{3}$rd for programs exhibiting good temporal locality–comes from optimizing the interactions between resizing decisions at different caching levels. We also propose several dynamic resizing algorithms that can automatically find good size configurations at runtime. Our results show dynamic MCR can achieve between 40–62% energy savings with slightly higher performance degradation than static-optimal MCR.*

1. Introduction

The power wall is currently the main limiter to achieving high performance in modern CPUs, and has been one of the most critical problems facing computer architects over the past several years [17]. Unfortunately, this problem will only get worse in the future as process technologies continue to scale to smaller feature sizes. As such, power efficiency will remain an extremely important design goal, and will require hardware designers to continue efforts to squeeze wasteful power consumption out of architectures.

A key place to look for power savings is in the on-chip cache hierarchy. Caches occupy a large portion of the CPU's available die area–upwards of 50% in today's CPUs–so they contribute significantly to a processor's overall power dissipation. In addition, caches are sized for the worst case. This means an average computation cannot effectively utilize all of the cache capacity. Such cache over-provisioning can result in significant waste that, if eliminated, can yield large power savings without sacrificing much performance.

Several researchers have investigated *cache resizing techniques* [1, 3, 4, 18, 19, 23, 32, 33] to target this form of waste. Cache resizing is an architecture-level power management technique that determines the minimum cache a program needs to run at near-peak performance, and then reconfigures the cache by enabling/disabling cache ways or sets to implement this efficient capacity. Resizing can reduce the amount of cache activated per access, and also enables circuit-level techniques (*i.e.*, gated-V_{dd} [23]) to shut down unused portions of the cache. This can translate into significant dynamic and static power savings. At the same time, though, resizing can also increase a cache's miss rate, resulting in greater power dissipation for the next level of cache (and degraded overall performance). Thus, techniques must finesse a balance between these conflicting factors in order to achieve a net power efficiency gain.

Although there has been significant work on cache resizing, existing techniques are limited in their optimization scope. In particular, most studies consider resizing a single level of cache only [1, 18, 19, 23, 32, 33], typically the L1 cache. In Balasubramonian's work [3, 4], two levels of cache are resized, but not independently (the sum of the two cache sizes is fixed). So, there's still only one cache whose size is explicitly controlled.

The trend for modern CPUs is towards deeper cache hierarchies, however, which distributes the power consumption across many caching levels. Today, three levels of cache is commonplace. For dynamic power consumption, the L1 is the greatest culprit, but the L2 and L3 can also consume non-negligible dynamic power, especially for memory-intensive workloads. For static power consumption, the L3 is by far the greatest concern due to its large area. But non-trivial static power can also be dissipated in the L2 as well. By only controlling the size of a single level of cache, existing techniques potentially miss significant opportunities for power savings.

The current lack of comprehensive cache resizing is partly due to the availability of other power management options, especially for caches below the L1. Because these caches are only referenced on an L1 miss, CPU performance is somewhat insensitive to their actual delay. Hence, it is feasible to trade off delay for power in the post-L1 caches. This has been exploited extensively by circuit-level techniques to mitigate static power consumption. In particular, multiple V_t devices [2, 14], adaptive body bias (ABB) [13, 22], and dynamic voltage scaling (DVS) [8, 15] all convert modest increases in cache access latency into significant static power reductions.

While extremely effective, circuit-level techniques for mitigating static power do not obviate the need for architectural approaches like cache resizing. Circuit mitigation only *reduces* leakage current. In contrast, cache resizing (plus power gating) can suppress leakage practically to zero for the gated portions of cache. Moreover, circuit- and architecture-level approaches are orthogonal. So, applying them in concert may ultimately yield the greatest static power savings.

In addition to flexibility for reducing static power, the low latency sensitivity of post-L1 caches also offer alternatives for reducing dynamic power. For example, serializing tag and data access ensures only a single data way is energized regardless of the number of total active ways, thus reducing dynamic power at the expense of some increased delay. But again, this does not preclude cache resizing. A serial cache still incurs wasteful tag energy as well as significant interconnect energy that resizing can address. And in some cases, serial caches may be too slow–for example, at the L2 given an L1 with a high miss rate–limiting their application.

This paper investigates *multi-level cache resizing* (MCR). MCR independently resizes all caches in a multi-level cache hierarchy–using selective cache ways [1] as the resizing mechanism–to minimize power consumption at all caching levels simultaneously. Our work quantifies the potential power benefits of MCR, providing insights into where savings come from as well as the challenges that must be overcome in order to attain the full benefits. We also investigate controlling MCR. Cache hierarchies with multiple reconfigurable caches exhibit a large number of sizing configurations. Our work develops techniques to navigate this complex search space to quickly find the best configurations. Currently, our focus is on the cache hierarchy beneath a single core–*i.e.*, one "vertical slice" of a multicore cache system. While we show how MCR can be integrated into multicores, evaluating MCR in a multicore CPU is beyond the scope of this paper. More specifically, we make the following contributions.

First, we study static resizing algorithms to quantify the potential benefits of our approach. Our study presents a static-optimal version of MCR that uses exhaustive off-line search to find the best sizing configuration. We apply static-optimal MCR to a 3-level reconfigurable cache hierarchy that can support 512 unique sizing configurations. (Our cache hierarchy also employs serial access and ABB to provide an efficient baseline). We find static-optimal MCR can reduce total cache energy dissipation by 58.9% while degrading performance by only 4.4% across 22 SPEC CPU2006 benchmarks. Moreover, our results show every caching level contributes significantly to the overall savings, underscoring the importance of resizing all the caches in a multi-level hierarchy.

Second, we show that finding the optimal configuration requires considering the interactions between resizing decisions across different caching levels. As mentioned earlier, cache resizing balances the power consumed by a cache against the power consumption it inflicts on the next level of cache through its cache misses. Notice, a cache's balance point depends on both the upstream and downstream caches (if any), which in MCR are themselves resizable. Thus, the optimal MCR configuration is the one that achieves balance *globally* across all the caches at the same time.

To quantify the impact of optimizing such inter-cache interactions, we compare static-optimal MCR against an algorithm that only achieves "local balance," which we call *sequential MCR*. We find static-optimal MCR reduces energy by 10.4% more than sequential MCR across all the SPEC benchmarks. But for the SPECint benchmarks which exhibit good temporal locality, static-optimal MCR saves $\frac{1}{3}rd$ more energy as compared to sequential MCR. So, while MCR provides significant overall power benefits, a non-trivial portion comes from optimizing inter-cache interactions, which is complex because the interactions grow as the *product* of the number of per-cache configurations.

Third, we study dynamic resizing algorithms to enable MCR at runtime. Our dynamic MCR work addresses the runtime overheads associated with finding the best cache sizing configurations for multiple levels of cache using intelligent search and prediction-based techniques. In particular, we propose using proportional sizing of non-searched levels to provide an optimized context for per-level searches. We also propose using hill-climbing to guide search, enabling search of any configuration. Lastly, we propose predicting the best configuration from reuse distance profiles acquired via way counters [27]. We call these techniques *proportional*, *hill climbing*, and *reuse distance-based prediction*, respectively.

Our results show dynamic MCR techniques are quite effective, providing between 40% and 62% energy savings on average. Reuse distance-based prediction provides the most energy savings, with hill climbing also providing good energy savings for the SPECint benchmarks, due to the ability to effectively optimize inter-cache interactions. We also show hill climbing and reuse distance-based prediction incur more performance degradation than static-optimal MCR, 7% and 9%, respectively, due to runtime overheads. Unfortunately, the proportional search strategy incurs a 12% performance loss because it often tries very poor configurations.

The remainder of this paper is organized as follows. Section 2 studies static-optimal MCR, and how inter-cache resizing decisions interact. Then, Section 3 presents several dynamic MCR techniques, and Section 4 evaluates their power benefits and performance. Finally, Section 5 discusses related work, Section 6 addresses multicore issues, and Section 7 concludes the paper.

2. Static-Optimal MCR

In this section, we study a static-optimal version of MCR to illustrate the potential gains of multi-level resizing and to provide insights into the interactions between different caching levels. The latter will motivate the potential complexity of controlling MCR, which is relevant for our dynamic MCR techniques presented in Section 3. Our discussion proceeds in three parts. We begin by describing the experimental methodology. Then, we present our static-optimal results. Lastly, we address multi-level interactions.

2.1. Experimental Methodology

We use the Simplescalar tools for the Alpha ISA [5] to conduct our study. In particular, we model a single out-of-order core attached to a 3-level cache hierarchy consisting of a split 8-way 32KB L1 cache, a unified 8-way 256KB L2 cache, and a unified 16-way 2MB L3 cache. (These cache sizes were chosen to match typical capacity-per-core found in today's CPUs). The cache block size is 64 bytes for all three caches, and the L2 and L3 caches maintain inclusion. Table 1 lists the detailed parameters for the core and cache hierarchy used in our experiments.

To enable cache reconfiguration, we modified Simplescalar's cache module to model way selection [1]. We assume all caches in the hierarchy, except for the L1 I-cache, are reconfigurable and can change their capacity in increments of a cache way from 1 to the associativity number of ways in the cache. (Our work does not consider I-cache resizing, and assumes the I-cache is always fixed). Hence, for our hierarchy, there are 8, 8, and 16 different configurations for the L1, L2, and L3 caches, respectively. *In the static-optimal version of MCR, we try all possible permutations of the per-cache configurations to identify the one that is most power-efficient*. We use energy $\times$ delay-squared to quantify power efficiency. (By emphasizing delay, ED^2 ensures that optimizing for power does not sacrifice performance too much). To limit the number of simulations, we only try configurations with an even number of L3 ways.[1] In total, the static-optimal version of MCR explores 512 (8^3) unique configurations.

While each cache's access delay also changes across different configurations, we assume a constant number of CPU cycles to access each cache chosen to handle that cache's worst-case access delay (*i.e.*, with all ways enabled).

Before resizing caches, we apply existing techniques to ensure the baseline cache hierarchy is reasonably efficient. In particular, we assume the L3 cache serializes tag and data accesses such that only a single data way is ever accessed regardless of the number of configured cache ways. Due to greater latency sensitivity, we perform parallel tags and data access in the L2 cache, though we serialize broadcasting the accessed data block in data array h-tree. Moreover, we assume the ability to dynamically change the threhsold voltage for the L2 and L3 caches through body biasing. Specifically, we assume super high-V_t devices throughout [12], but apply

[1]This also avoids configuring the L3 cache with a single way (128KB) which makes several of the L2 configurations infeasible due to the need to maintain inclusion.

Core	
	4-way out-of-order issue
	4-entry IFQ, 16-entry ROB, 8-entry LSQ
	4-int, 4-FP, 2-ldst
	1K-entry comb (2K bimod and 1K Gag) predictor
Cache Hierarchy	
L1 Cache	split, 32 KB, 8-way, 64-byte blocks
	Latency: 2 cycles
	Parallel tag, data, and data array h-tree
	Data Read Energy: 0.0729 nJ
	Leakage: 22.11 mW
L2 Cache	unified, 256 KB, 8-way, 64-byte blocks
	Latency: 3 cycles
	Paralle tag and data, serial data array h-tree
	Data Read Energy: 0.7361 nJ
	Leakage (standby/active): 44.09 mW / 2380 mW
L3 Cache	unified, 2 MB, 16-way, 64-byte blocks
	Latency: 7 cycles
	Serial tag, data, and data array h-tree
	Data Read Energy: 1.888 nJ
	Leakage: 88.31 mW / 6711 mW
DRAM	Data Read Energy: 10 nJ

Table 1: Simulation parameters used for the experiments.

reverse body bias (RBB) in standby mode to further reduce standby leakage [30]. When an access occurs, we apply a forward body bias (FBB) to restore the threshold voltage for low access delay. We assume that applying FBB does not impact the access delay for the cache [30]. We utilize stack effect in conjunction with ABB to model way selection [25, 30]. We use CACTI 6.5 [21] for power modeling and use the Model for Assessment of cmoS Technologies And Roadmaps (MASTAR 2011) from ITRS [20] to derive parameters required for CACTI according to the assumption.

To drive our simulations, we use 22 SPEC CPU2006 benchmarks (11 integer and 11 floating point), as shown in Table 2. We compiled the benchmarks on Alpha CPU emulator. We run Debian Etch on it and used native Alpha compiler, gcc-4.1.1. We compiled the benchmarks with optimization of -O2 option and linked glibc-2.5 statically. One integer benchmark (403.gcc) and six floating point benchmarks (416.gamess, 433.milc, 447.dealll, 450.soplex, 465.tonto, and 481.wrf) could not be compiled, so they have been omitted from our study.

Using the reference inputs, all of the benchmarks were run to completion on SimPoint [10]. The second column in Table 2, labled "SPt," reports the number of simulation points that SimPoint identified (each simulation point contains 100M instructions). Then, we ran each benchmark's simulation points on our modified Simplescalar simulator 512 times, once for each of the 512 possible cache sizing configurations, and measured the resulting ED^2. The three columns labeled "Static Optimal" in Table 2 report the number of L1, L2, and L3 cache ways corresponding to the configuration with the best ED^2. This is the static-optimal MCR configura-

Benchmark		Static Optimal			%	%	Sequential		
	SPt	L1	L2	L3	CPI	DRAM	L1	L2	L3
Integer									
400.perlbench	14	2	2	4	8%	4%	4	8	6
401.bzip2	21	2	3	16	2%	0%	2	4	16
429.mcf	27	2	2	16	0%	0%	2	2	16
445.gobmk	22	2	2	4	12%	6%	4	8	6
456.hmmer	19	4	3	12	2%	3%	4	3	12
458.sjeng	16	2	1	4	8%	1%	4	8	4
462.libquantum	17	1	1	4	0%	0%	1	2	4
464.h264ref	15	2	2	4	17%	12%	8	3	12
471.omnetpp	3	3	2	16	2%	0%	5	2	16
473.astar	21	2	2	12	4%	2%	2	3	12
483.xalancbmk	20	2	2	6	9%	7%	2	8	8
Floating Point									
410.bwaves	24	8	2	4	0%	0%	8	2	4
434.zeusmp	29	2	1	4	6%	7%	3	3	4
435.gromacs	20	1	1	4	3%	2%	2	2	4
436.cactusADM	6	2	2	4	3%	1%	3	2	4
437.leslie	25	5	2	4	4%	4%	8	2	6
444.namd	23	1	1	4	7%	0%	6	2	4
453.povray	16	2	2	4	6%	0%	3	3	4
454.calculix	16	1	1	4	-1%	1%	2	2	4
459.GemsFDTD	22	3	3	4	3%	4%	4	4	6
470.lbm	14	2	1	4	0%	0%	2	4	6
482.sphinx3	19	2	1	4	3%	6%	3	2	4
Average		2.4	1.8	6.5	4.4%	2.7%	3.7	3.6	7.4

Table 2: SPEC CPU2006 benchmarks used in the experiments. Columns report number of simulation points, number of L1/L2/L3 ways employed by static-optimal MCR, percent increase in CPI, percent increase in DRAM dynamic energy, and number of L1/L2/L3 ways employed by sequential MCR.

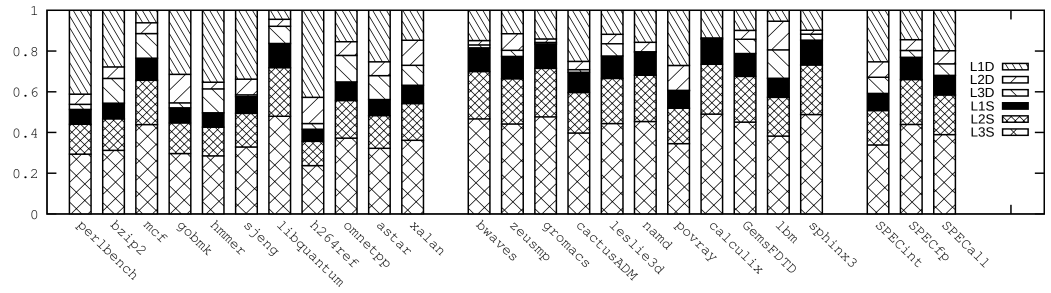

Figure 1: Breakdown of L1, L2, and L3 dynamic and static energy for the SPEC CPU2006 benchmarks. The last 3 bars show the average over the SPECint (first 11 bars), SPECfp (second 11 bars), and all benchmarks.

tion.

2.2. Evaluation

Before presenting the MCR results, we first characterize our baseline cache hierarchy by showing the total energy dissipation across all of the caches. (Because our results later on will involve techniques that affect execution time, we always report energy rather than power consumption). In particular, Figure 1 breaks down the energy consumed in the L1, L2, and L3 caches including both dynamic and static energy, labeled "L1 dynamic," "L2 dynamic," "L3 dynamic," "L1 static," "L2 static," and "L3 static," respectively. The first 11 bars show breakdowns for the SPECint benchmarks, the second 11 bars show breakdowns for the SPECfp benchmarks, while the last 3 bars show breakdowns averaged across the integer and floating point benchmarks, labeled "SPECint" and "SPECfp," and then across all the benchmarks, labeled "All."

Not surprisingly, the L1 cache's high access frequency leads to significant dynamic energy dissipation. In Figure 1, we see L1 dynamic energy accounts for 22.1% of the total on-chip cache energy averaged across all the benchmarks. When including static energy consumption, the L1 cache accounts for almost $\frac{1}{3}rd$ of the total cache energy (31.7%). But this leaves a significant portion of the energy unaccounted for. In fact, the most dominant component is the L3's static energy, accounting for 38.0% of the total on-chip cache energy on average. Notice, the L3 is still a major consumer even after applying aggressive ABB techniques to mitigate its leakage. Moreover, other sources of energy dissipation across the cache hierarchy are significant as well. In particular, Figure 1 shows the L2's static energy contributes 19.2% on average. Even the L2 and L3's dynamic energy (6.3% and 5%, respectively) are non-trivial. These results show there is significant energy/power consumption across all of the caches. So, applying cache resizing to *every* cache in the multi-level hierarchy has the best chance to achieve large power savings.

Figure 2 presents our MCR results. The bars labeled "SO" report the energy consumption of the static-optimal version of MCR, which uses the cache-way configurations listed in Table 2 (columns labeled "Static Optimal"). Each bar in Figure 2 is normalized to the corresponding application's energy consumption for the baseline cache hierarchy reported in Figure 1 and is broken down into the same six components as before. Results for the SPECint and SPECfp benchmarks appear in the top and bottom graphs of Figure 2, respectively.

The static-optimal version of MCR achieves significant energy savings. As the "All" bars in Figure 2 show, static-optimal MCR reduces energy dissipation by as much as 81.8% (libquantum), and by 58.9% on average across all of the benchmarks. Six benchmarks experience an energy reduction exceeding 70%.

At the same time, static-optimal MCR does not degrade performance significantly. The sixth column of Table 2, labeled "% CPI," reports the percentage increase in CPI for static-optimal MCR compared to the baseline cache hierarchy. As Table 2 shows, 14 of the 22 benchmarks incur less than 5% performance degradation, with all but 2 benchmarks slowing down by less than 10%. Averaged across all the benchmarks, static-optimal MCR degrades performance by only 4.4%.

These performance degradations are due in part to an increased L3 cache miss rate. Having more L3 misses not only impacts performance, it also increases dynamic power consumption in main memory. The seventh column of Table 2, labeled "% DRAM," reports the increase in dynamic energy incurred within DRAMs as a percentage of the total on-chip cache energy. As Table 2 shows, the energy increase in main memory is only 2.7% of the total on-chip cache energy when averaged across all the benchmarks. Even after accounting for main memory effects, static-optimal MCR is still able to provide a 56.2% reduction in energy consumption. Overall, our results show static-optimal MCR reduces energy significantly at a minimal cost to performance.

The large energy/power savings of static-optimal MCR are due to its aggressive down-sizing of caches. As the "Static Optimal" columns in Table 2 show, our technique employs only 2.4, 1.8, and 6.5 cache ways on average (last row in Table 2) for the L1, L2, and L3 caches, respectively. This represents a 70%, 77.5%, and 59.4% reduction in cache capacity. More importantly, notice this cache down-sizing is comprehensive, occurring significantly across *all three levels of cache*. In fact, Table 2 shows static-optimal MCR chooses a smaller number of ways than the baseline in every case except three (the L3 cache for bzip2, mcf, and omnetpp).

Because the entire cache hierarchy is consistently downsized, static-optimal MCR targets every power source pointed out in Figure 1. Comparing the "All" bars in Figures 1 and 2, we see the two major sources–L1 dynamic and L3 static–are reduced by 68% and 60% on average, respectively. L2 static energy is reduced even more, by 77%. And the L1 static and L2 dynamic components are reduced by 68% and 49%, respectively. (Interestingly, L3 dynamic energy increases by 48% due to multi-level interactions which we will discuss in the next section.) These results emphasize the effectiveness of multi-level resizing in exploiting all opportunities for energy/power savings across a multi-level cache hierarchy.

Finally, we point out static-optimal MCR gains are asymmetric across datatype. As the "SPECfp" and "SPECint" bars from Figure 2 show, energy is reduced by 68.8% for the floating point benchmarks compared to 50.7% for the integer benchmarks. The SPECfp benchmarks exhibit less temporal locality compared to the SPECint benchmarks, so configuring caches with large capacity tends to be wasteful. This encourages more aggressive cache down-sizing. The most dramatic example of this is for the L3 cache. On the baseline cache hierarchy, L3 static energy is 38% higher for the SPECfp benchmarks compared to the SPECint benchmarks (see Figure 1). But after applying static-optimal MCR, it becomes 37% lower (see Figure 2). The greater opportunity to reduce cache capacity without impacting performance results in superior energy savings for the floating point benchmarks.

2.3. Multi-Level Interactions

As discussed in Section 1, cache resizing changes the balance between the power consumed by a cache and the power consumption it inflicts on the next level of cache through its cache misses. A significant challenge for MCR is *to negotiate this balance simultaneously across all resized caches*.

More specifically, resizing decisions across caching levels are coupled, a fact that MCR can exploit. When trying to pick a cache size at a particular caching level, MCR is not "stuck" with the access energy of a downstream cache. Instead, MCR can down-size the downstream cache to reduce the cost of cache misses, thus enabling more aggressive down-sizing for the cache in question. MCR is also not a helpless bystander in terms of the upstream cache's incident reference stream. Instead, MCR can up-size the upstream cache to reduce its miss rate, thus enabling more aggressive up-sizing for the cache in question, if desirable.

This section evaluates how important optimizing such inter-cache interactions is to the gains reported in Section 2.2. In particular, we compare static-optimal MCR–which by definition achieves the best balance across all caching levels–

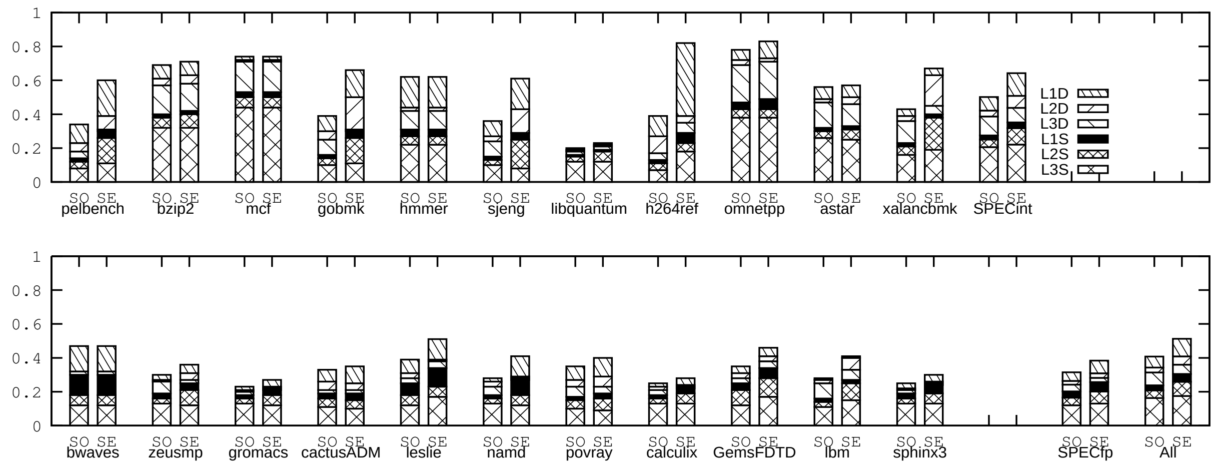

Figure 2: Comparison of static-optimal and sequential MCR. Individual bars are broken down into L1, L2, and L3 dynamic and static energy. Top graph shows SPECint results while bottom graph shows SPECfp results.

against a different technique that only achieves local balance. In the alternate technique, we sequentially search each caching level separately, allowing only that level to resize. During each level's search, the other non-searched levels maintain their baseline capacities. Then, we combine the best capacities found (*i.e.*, achieving best ED^2) from all per-level searches into a single configuration. We call this technique "sequential MCR." Sequential MCR still aggressively down-sizes individual caches, but it cannot optimize the coordination of resizing decisions between caching levels. On the other hand, sequential MCR requires searching fewer configurations to find its solution (more on this in a moment).

The last three columns in Table 2, labeled "Sequential," report the number of L1, L2, and L3 cache ways used by sequential MCR. As Table 2 shows, sequential MCR employs 3.7, 3.6, and 7.4 cache ways on average (last row in Table 2) for the L1, L2, and L3 caches, respectively. But as mentioned earlier, static-optimal MCR employs only 2.4, 1.8, and 6.5 cache ways. So, static-optimal MCR down-sizes by an additional 1.3, 1.8, and 0.9 cache ways on average compared to sequential MCR. This represents a 16.3%, 22.5%, and 5.6% further reduction in cache capacity for the L1, L2, and L3 caches, respectively.

In Figure 2, the bars labeled "SE" plot the energy dissipation of the sequential version of MCR across our SPEC benchmarks. Comparing the SE and SO bars in Figure 2, we see that sequential MCR does not achieve as much energy savings compared to static-optimal MCR for almost every benchmark. In some cases, the energy savings gap is large. For 5 benchmarks (perlbench, gobmk, sjeng, h264ref, and xalancbmk), static-optimal MCR achieves 25% or more energy savings compared to sequential MCR. For one benchmark (h264ref), the gap is 42%. Averaged across all of the benchmarks, sequential MCR reduces energy dissipation by 48.5% compared to the baseline, which is 10.4% worse than static-optimal MCR.

As in Section 2.2, static-optimal versus sequential MCR gains are also asymmetric across datatype, with a larger energy savings gap for the integer benchmarks. Figure 2 shows sequential MCR reduces energy consumption by only 34.5% for the SPECint behcmarks compared to 50.7% for static-optimal MCR–*i.e.*, static-optimal MCR saves $\frac{1}{3}rd$ more energy than sequential MCR for SPECint. As discussed in Section 2.2, the SPECfp benchmarks exhibit low temporal reuse, favoring aggressive down-sizing especially for the L3 cache. This leaves very little room for multi-level interactions to make a difference. In contrast, the SPECint benchmarks make better use of the on-chip cache, so there is a larger range of "interesting" cache sizes. In this case, there is much more room for coordinating cache resizing across levels to make a bigger difference.

Overall, Figure 2 demonstrates MCR algorithms that consider the interactions between different resizing decisions can achieve significant additional energy/power savings, especially for benchmarks with good locality characteristics. But along with this opportunity comes a challenge: inter-cache interactions grow as the product of the number of per-cache configurations. Thus, optimizing them involves complex search. For example, in our study, sequential MCR requires considering only 24 configurations while static-optimal MCR requires considering 512 configurations.

3. Dynamic MCR Techniques

Having considered static off-line approaches, we now study dynamic MCR techniques. Dynamic MCR techniques determine multi-level cache configurations at runtime. As such, they control cache resizing fully automatically–*i.e.*, without off-line profiling. In addition, they also have the ability to adapt to time-varying application behavior.

To be successful, dynamic MCR must employ efficient algorithms for determining the best cache configurations; otherwise, runtime overheads may outweigh the benefits of cache resizing. The overhead issue is especially acute for dynamic MCR since it must determine cache sizes for multiple levels of cache, as opposed to previous techniques that explicitly control only a single level of cache. Worse yet, optimizing inter-cache interactions, which is necessary to achieve the best configurations, is combinatorially complex and has the potential to drastically increase overheads (see Section 2.3).

This section presents several runtime algorithms for dynamic MCR. Section 3.1 describes techniques that employ intelligent search to find the best configurations rapidly. Then, Section 3.2 introduces techniques that use reuse distance profiling to predict the best configuration outright. We discuss each technique's runtime overhead, as well as its ability to achieve the optimal cache configuration, before conducting experiments later in Section 4.

3.1. Search

Search techniques directly measure different cache configurations' performance and power consumption to find the best one. Section 2 performed exhaustive search across multiple program runs. For dynamic MCR, however, the search process occurs during the production run itself, so only a single run of the program is available. Moreover, because most configurations are sub-optimal, search slows down the program, introducing runtime overhead. Hence, it is crucial for these techniques to minimize the amount of time spent searching.

Rather than search exhaustively, we investigate different strategies for intelligently picking the configurations to try so that the optimal configuration (or at least a very good one) is found quickly. The search process can be repeated periodically to increase the likelihood of finding high-quality solutions. This can also track time-varying application behavior.

To implement search on-line, we divide a program's execution into short time intervals, called *epochs*, and try different cache configurations across different epochs. For all of our techniques, we assume a fixed epoch size of 1M instructions. After search completes, the cache hierarchy is configured with the best configuration found, and the program is allowed to execute with this configuration. When searching repetitively, we alternate between "search phases" and "execute phases" until program termination. The number of epochs spent in execute versus search phases can be tuned to trade off overhead for adaptivity.

3.1.1. Sequential. We consider three different search strategies. The first is a dynamic version of the off-line sequential MCR technique from Section 2.2. As in Section 2.2, we sequentially search each caching level, trying all capacities at each level while holding the non-searched levels at their baseline capacities. We then combine the best capacities found from the per-level searches into a single configuration. The only difference is that we try each capacity for only a single epoch instead of the entire program run.

The advantage of the sequential strategy is that it finds a good configuration fairly quickly, requiring 24 epochs for each search phase. But as discussed in Section 2.2, the sequential strategy rarely finds the best configuration because it does not effectively optimize inter-cache interactions.

3.1.2. Proportional. The problem with the sequential strategy is that it holds the non-searched caches at their baseline capacities. For most programs, the baseline capacities are too large. Hence, the sequential per-level searches find the best capacities given the rest of the cache hierarchy is over-provisioned, but this is usually not the global optimum.

We try to improve upon the sequential strategy in two ways. First, after finding the best capacity for a particular caching level, we set that level to its best capacity during subsequent searches at the other levels. This provides a more optimized cache hierarchy for the later searches, resulting in better choices down the line. Moreover, we start searching for the L1, then for the L2, and finally for the L3, so the caches are fixed in smallest to largest cache order. From our experience, this produces the best results.

Second, when searching a particular caching level, we resize the non-fixed levels (*i.e.*, the higher-capacity caches) in proportion to the level being searched. In other words, when resizing the L1, the L2 uses the same number of ways as the L1. And when resizing the L2, the L3 uses twice the number of ways as the L2. This tends to provide more optimized cache capacities downstream in the non-fixed levels. We call the search technique with both of these improvements the "proportional" strategy.

Like sequential, the proportional strategy also completes in 24 epochs, but it finds the optimal configuration more frequently due to its better search strategy. However, proportional can still miss the best configurations. While it provides better "context" for the per-level searches, there are still many configurations that are impossible to search. Again, this limits the ability to fully optimize inter-cache interactions.

3.1.3. Hill-Climbing. Our last search strategy is hill-climbing. In this approach, search phases begin by trying the configuration with half the maximum capacity at each caching level–4 ways of L1, 4 ways of L2, and 8 ways of L3–and setting this to be the "current-best" configuration. Then, we try "nearby" configurations that differ by one cache way. In particular, we try six configurations, each adding or subtracting a single cache way to or from the current-best's L1, L2, or L3. Among these trials, we identify the one with largest ED^2. If this "best-neighbor" is better than current-best, we set current-best to best-neighbor and repeat the search across neighbors from the new current-best configuration. This process continues until no neighbor improves on current-best, at which time the search phase completes.

In contrast to sequential and proportional, the hill-climbing strategy can reach any configuration in the search space, so it can potentially find the optimum everytime. However, hill-climbing's movement towards the optimal configuration may be obstructed by local optima and/or noisy dynamic behavior across epochs.

Whereas sequential and proportional always finish after 24 epochs, hill-climbing's latency is application dependent. Hill-climbing is generally more expensive because it moves slowly, requiring six epochs to learn the direction of largest ED^2 increase. On the other hand, because sampling starts from the "average" configuration, hill-climbing begins close to all solutions. For optimal configurations that are near the average configuration, hill-climbing can exhibit even lower runtime overhead than sequential or proportional.

3.2. Prediction

In addition to search, we also consider prediction-based techniques. These techniques predict the behavior of different cache sizing configurations via reuse distance profiles, thus eliminating search overhead. Our approach relies on *way counters* [27]. In this technique, a separate counter is implemented per cache way, each representing a different stack position across the cache sets. On a cache hit, the stack depth of the hitting cache block is identified, and the corresponding way counter is incremented. Hence, each way counter tracks the number of hits attributable to cache blocks at a particular stack depth, permitting prediction of the number of additional misses that would occur as cache capacity is reduced in way increments. Way counters have been used extensively to partition shared caches for multiprogrammed workloads [6, 11, 24, 16, 27, 28, 29].

We adapt way counters for dynamic MCR. In particular, we extend all three caches in our cache hierarchy with way counters. We also replace the search phases from Section 3.1 with "profiling phases." During a profiling phase, we configure all caches with their maximum capacity, thus acquiring way counts (and cache-miss predictions) for every possible capacity at each caching level. Since all capacities are profiled simultaneously, profiling phases can be short. When profiling on-line, each profiling phase lasts for three epochs.

At the end of each profiling phase, we perform prediction. Because way counters can predict cache misses for every capacity, we can exhaustively predict per-level cache-miss counts for all combinations of capacities across the three caching levels–*i.e.*, 512 configurations. After predicting each configuration's cache-miss counts, we then predict performance and power consumption. (Section 4.1 will discuss how this is done). This yields an ED^2 prediction for each cache sizing configuration, allowing identification of the best configuration.

Although reuse distance profiling avoids search overhead, it does incur runtime overhead to compute the predictions. In addition, reuse distance profiling may incur prediction error, especially since it cannot directly measure performance. Lastly, the technique consumes slightly more power during profiling phases due to the addition of way counters. On the other hand, prediction via reuse distance profiles is the most comprehensive technique since it can exhaustively evaluate inter-cache interactions.

4. Dynamic MCR Results

This section evaluates the dynamic MCR techniques introduced in Section 3. We first discuss implementation issues. Then, we present the on-line performance and power results.

4.1. Implementation

We implemented our dynamic MCR techniques in the simulator from Section 2.1. In particular, we modified our simulator to distinguish between search/profiling phases and execute phases. During search or profiling, our simulator posts an interrupt every 1M instructions–*i.e.*, every epoch–and executes an interrupt handler. (This occurs for as many epochs as needed to complete the search or profiling phase). We also modified the simulator to allow software to reconfigure its caches. For the search-based techniques, the interrupt handlers implement one of the search strategies from Section 3.1, reconfiguring the cache across epochs to try different cache sizing configurations. After each epoch, the interrupt handler measures the performance and power consumed for the previous configuration before trying the next configuration.

Performance and power measurements are provided by hardware performance counters in the simulator. In particular, each interrupt handler reads a cycle counter to measure an epoch's execution time. We also implemented cache access and cache miss counters for each caching level. Using per-access energies from CACTI and the cache access counts, the interrupt handler can compute an epoch's dynamic energy consumption. And using per-cycle leakage currents from CACTI and cycle counts, the interrupt handler can compute an epoch's static energy consumption. Together, these measurements yield ED^2 values for each searched configuration.

To facilitate prediction, our simulator implements way counters at each caching level[2] which the interrupt handler can also read. At the end of each profiling phase, the interrupt handler predicts ED^2 for all possible cache sizing configurations, as described in Section 3.2. Energy is computed in the same way as for the search-based techniques, except the cache access and miss counts are predicted from the way counts rather than being measured. Performance is computed by using the predicted cache miss counts at each caching level to derive the average memory access time, which is then used to compute CPI assuming a single-issue in-order processor–*i.e.*, $\mathrm{CPI}_{in-order}$. To estimate the impact of ILP exploitation on actual CPI, we measure the ratio CPI_{actual} / $\mathrm{CPI}_{in-order}$ in previous profiling phases, and apply the same ratio to derive CPI_{actual} for the current profiling phase.

Our simulator accounts for the overheads associated with resizing each cache. When up-sizing, we assume 1000 cycles

[2]Although negligible, the way counters' power consumption is accounted for during the profiling phases by our simulator.

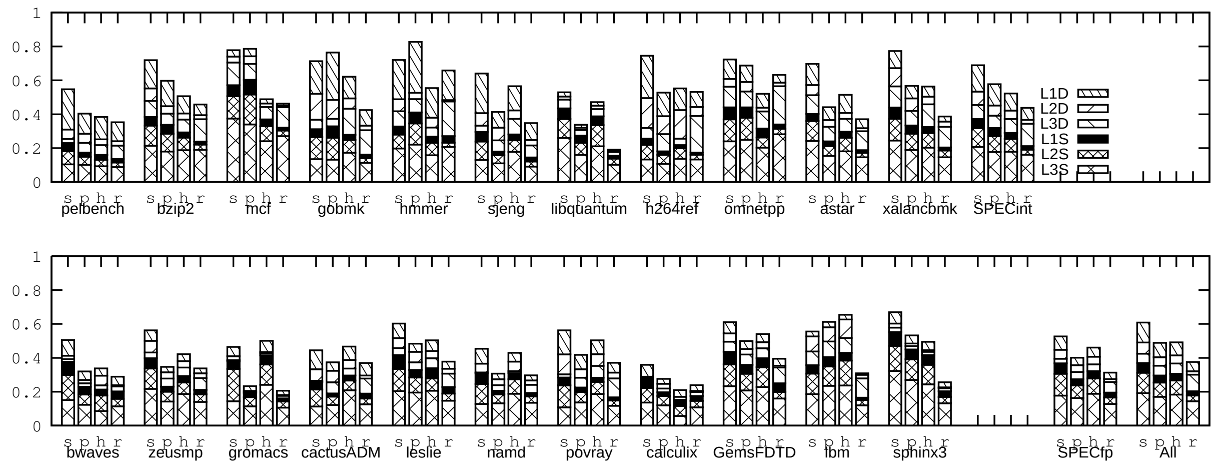

Figure 3: Comparison of sequential, proportional, hill climbing, and RD-based prediction. Individual bars are broken down into L1, L2, and L3 dynamic and static energy. Top graph shows SPECint results while bottom graph shows SPECfp results.

to power up each way, and 10 cycles per way to flash invalidate the newly powered-on cache blocks. When down-sizing, we walk the down-sized way(s) to flush the contents. Clean cache blocks are discarded after checking upstream caches to maintain inclusion. Dirty cache blocks check upstream caches and are also written back to the next-lower level. We assume these operations are piplined such that flushing takes 1 cycle per walked cache block. Down-sized ways are selected in reverse way ID order. Because we do not physically move cache blocks once they are filled, the flushed cache blocks have an equal probability of being at any position in the LRU stack. Moreover, we do not attempt to reconstruct the per-set LRU stacks after flushing. Resizing is performed on the L1, then the L2, and lastly the L3 cache.

To drive our experiments, we use the SPEC CPU2006 benchmarks from Table 2. In particular, we use the same simulation points as reported in Table 2, except we extend each one from 100M instructions to 500M instructions. Finally, we allow each dynamic MCR technique to perform four cache resizings (*i.e.*, we run four search/profiling phases each followed by an execute phase) per 500M instruction simulation point.

4.2. Results

Figure 3 presents our dynamic MCR results. This figure shows the energy consumption in the cache hierarchy for different dynamic MCR techniques in a format similar to Figure 2. The bars labeled "S," "P," and "H" report total energy consumption for the sequential, proportional, and hill climbing search techniques, respectively, while the bars labeled "R" report total energy consumption for the reuse distance-based prediction technique. Each bar in Figure 3 is normalized to the corresponding application's energy consumption for the baseline cache hierarchy, and is broken down into the same dynamic/static components for the L1, L2, and L3 as in Figure 2. Averages over different datatypes are labeled "SPECint" and "SPECfp" while averages over all benchmarks are labeled "All."

These results demonstrate our dynamic MCR techniques can provide significant energy savings compared to the baseline cache hierarchy. As the "All" bars in Figure 3 show, dynamic MCR provides between 40% energy savings when using the sequential strategy to as much as 62% energy savings when using the reuse distance-based prediction strategy averaged across all of the SPEC benchmarks. In addition, comparing these results to Figure 2, we see our dynamic MCR techniques achieve the potential energy savings that off-line techniques provide. Specifically, the on-line version of sequential MCR loses only 9% of the off-line sequential MCR savings (40% vs. 49%), and the best dynamic MCR technique, reuse distance-based prediction, actually achieves slightly more savings than the static-optimal MCR savings (62% vs. 58.9%). This is impressive considering dynamic MCR incurs runtime overhead to find the best configurations and to resize caches.

Figure 3 also illustrates the benefits of our intelligent search and prediction techniques. As the "P" bars in the "All" category show, the proportional strategy provides 51% energy savings, which is an additional 11% more than the sequential strategy. In fact, comparing the "P" and "S" bars across all the benchmarks, we see proportional is better than sequential in 18 of the 22 benchmarks. This shows proportional's approach in providing better contexts for the per-level searches compared to sequential can make a significant difference.

Looking at the "H" bars in the "All" category of Figure 3, we see the hill climbing strategy is comparable to the proportional strategy, also providing an energy savings of 51%. As mentioned earlier, hill-climbing can exhibit long search times and can get stuck on its way to the optimum. We found the latter to be especially true for the SPECfp benchmarks. In this

	% CPI				% DRAM			
	S	P	H	R	S	P	H	R
SPECint Benchmarks								
400.perlbench	17.6%	25.9%	23.2%	14.5%	14%	15.1%	14.1%	5.3%
401.bzip2	19.7%	42.2%	21.2%	25.6%	17.5%	36.8%	19.8%	19.9%
429.mcf	2.5%	3.4%	9.6%	3%	4.6%	5%	15%	3.5%
445.gobmk	8.9%	9.7%	9.1%	12.2%	6.5%	7.1%	2%	3.2%
456.hmmer	7.7%	12.3%	9.3%	17.2%	17.8%	26.7%	22.8%	16%
458.sjeng	3.9%	14.1%	5.6%	8.1%	2.1%	3.6%	-0.5%	0.6%
462.libquantum	1.3%	0%	0%	0%	1.5%	-5.7%	-1.4%	-0.3%
464.h264ref	12.8%	27.4%	14.4%	23.3%	11.1%	16.8%	9.5%	10.1%
471.omnetpp	10.7%	12.7%	16.1%	10.4%	22.8%	14%	20.3%	11.5%
473.astar	7%	9%	11.5%	1.7%	10.1%	22.5%	16.6%	-4.1%
483.xalancbmk	7.3%	18.7%	6%	16.5%	7.5%	18.6%	4.1%	13.4%
SPECfp Benchmarks								
410.bwaves	3.1%	4.6%	5.1%	5%	2%	0%	1.5%	0.3%
434.zeusmp	3.3%	8.5%	3.5%	6.1%	5.1%	7.9%	2.5%	4.8%
435.gromacs	3.3%	11.3%	1.4%	2.1%	4.4%	14.6%	1.3%	1.1%
436.cactusADM	6.1%	10.4%	3.5%	6.5%	6.4%	2%	0.6%	0.9%
437.leslie	4.5%	9.3%	4%	13.8%	4.7%	1.7%	0.5%	2.5%
444.namd	4.5%	9.3%	3.4%	6.4%	3.8%	4.3%	0.9%	0.5%
453.povray	10.1%	10.1%	4.9%	10.7%	7%	4.8%	0.8%	0.2%
454.calculix	0%	0%	0%	0%	2.8%	0.7%	0.6%	0.1%
459.GemsFDTD	2.9%	7.9%	1.5%	18.2%	3%	2.3%	0%	3.5%
470.lbm	0.2%	0.1%	0%	2.6%	1.2%	-1%	-3.6%	-0.4%
482.sphinx3	1.9%	3.5%	1.6%	3.2%	2.9%	4.4%	4%	6.2%
Avg	6%	12%	7%	9%	7%	9%	6%	4%

Table 3: Dynamic MCR performance and DRAM energy results. "% CPI" and "% DRAM" columns report % increase in CPI and DRAM dynamic energy, respectively.

case, hill climbing is actually *worse* than proportional. But for the SPECint benchmarks where there is a greater potential for optimizing inter-cache interactions (see Section 2.3), the increased flexibility of hill climbing's search strategy compared to proportional results in noticeable benefits. As the "SPECint" bars in Figure 3 show, hill-climbing provides 5.5% more energy savings than proportional.

While the benefits of optimizing inter-cache interactions is apparent in hill climbing, it is most visible for reuse distance-based prediction. Comparing the "R" and "P" bars in the "All" category of Figure 3, we see prediction provides 11.1% more energy savings than proportional search; and comparing the "R" and "H" bars in the "All" category, we see prediction provides a similar 11.4% more energy savings than hill climbing search. As discussed in Section 3.2, reuse distance-based prediction exhaustively considers all sizing configurations. The results in Figure 3 show such comprehensive evaluation can provide increased energy savings through better optimization of inter-cache interactions.

The columns in Table 3 labeled "S," "P," "H," and "R" under "% CPI" report the percentage increase in CPI for the sequential, proportional, hill climbing, and reuse distance-based prediction techniques, respectively. Averaged across all the benchmarks, the performance degradation for sequential, hill climbing, and prediction is between 6–9%. As shown in Table 2, the performance degradation for static-optimal MCR is only 4.4%. So, these dynamic MCR techniques pay an additional 1.6–4.6% performance loss on average for their runtime overheads. Unfortunately, the proportional strategy has noticeably higher performance degradation, 12%. Although proportional only runs 24 epochs to complete each search phase, many of the searched configurations have very poor performance. In particular, proportional scans the L3 cache when searching for the best L1 and L2 capacities, so it often runs with very high L3 miss rates which impact performance significantly.

Finally, the columns in Table 3 under "% DRAM" report the increase in dynamic energy incurred within DRAMs as a percentage of the total on-chip cache energy. (Again the "S," "P," "H," and "R" columns refer to our four dynamic MCR techniques). These results show off-chip memory consumes between 4–9% more energy due to increased misses from L3 cache down-sizing. Again, these are larger than the static-optimal MCR results (2.7% in Table 2) and represent the lower-quality cache sizing decisions that are made dynamically as compared to an omniscient off-line technique. Nevertheless, the DRAM energy increases are still relatively small compared to the overall on-chip cache energy reductions.

5. Related Work

A large body of work exists on cache resizing. Selective cache ways [1] uses off-line profiling to drive disabling of cache ways for dynamic power savings. DRI caches [23, 33, 32] use cache-miss counts to detect over-provisioning, and resize across either cache sets or ways. In addition, DRI caches also gate the power supply to unused portions of cache, conserving both dynamic and static power. Madan *et al* [18] propose resizing L2 caches by dynamically extending their capacity into stacked DRAM. Malik *et al* [19] study selective ways in the MCore CPU. All of these prior studies consider resizing a single level of cache only whereas MCR addresses the problem of resizing multiple levels of cache. In particular, we develop novel algorithms for navigating the much larger configuration space in the multi-level case to efficiently find the best configurations.

Balasubramonian *et al* [3, 4] propose resizing two levels of cache, either the L1/L2 or the L2/L3, by partitioning a common pool of SRAM arrays to different caching levels. Because partitionings always utilize all of the available SRAM, only one cache's size is controlled independently. Hence, in this technique, it is impossible to optimize the balance point of different caching levels simultaneously as is done in MCR.

Besides resizing, researchers have studied other adaptive cache techniques as well. Dropsho *et al* [7] propose *accounting caches* which divide a cache's ways into primary and secondary groups. Each cache access searches the two groups sequentially, accessing the secondary only on a primary miss. This saves power if secondary accesses are infrequent. Zhang *et al* [35] propose *way concatenation* which permits flexible organization of cache banks to form direct-mapped, 2-way, or 4-way set-associative caches. Neither accounting caches nor way concatenation address capacity allocation across different levels of cache, the main focus of MCR.

Silva-Filho *et al* [26] and Gordon-Ross *et al* [9] study design-time techniques for optimizing 2-level cache hierarchies. This body of work tries to find the best block size and associativity–as well as cache capacity–for two caching levels. They consider a more complex design space than we do, and employ more costly search techniques that are suitable for design analysis only. In contrast, MCR is an architecture-level power management technique. It solves a more constrained problem, but provides algorithms suitable for runtime use. Similarly, Zhang and Vahid [34] search for the best cache architecture using a reconfigurable hardware platform. But they only consider optimizing a single level of cache.

Cache partitioning explicitly allocates shared cache across multiprogrammed workloads, providing cache to those programs that can best utilize it. The majority of techniques focus on performance [6, 24, 16, 27, 28, 31]. More recently, techniques have also tried to reduce power consumption [11, 29] by withholding allocation and shutting down portions of the shared cache, similar to cache resizing. Like MCR, cache partitioning also employs reuse distance profiles to drive allocation decisions. But cache partitioning is a "horizontal" allocation technique compared to MCR which is a "vertical" allocation technique. While both can save power, cache partitioning does so by optimizing utility across competing threads whereas MCR does so by optimizing balance between caching levels.

Finally, significant research has explored circuit-level techniques for reducing a cache's static power consumption. Multi-V_t techniques [2, 14] employ low-V_t devices along critical paths and high-V_t devices along non-critical paths to save power while still maintaining performance. Gated-V_{DD} [23] uses high-V_t devices to gate the power supply to unused portions of cache. Adaptive body bias [13, 22] controls the backgate voltage to place devices in a standby low-leakage mode when not in use, but then restores the devices to an active high-performance mode when the cache is accessed. Lastly, dynamic voltage scaling [8, 15] can similarly transition between standby and active modes by scaling the supply voltage. Similar to other cache resizing techniques [33, 32], MCR relies on Gated-V_{DD} to essentially eliminate leakage for unused portions of the cache.

6. Multicore Integration

MCR allocates resources "vertically" across different caching levels within a cache hierarchy. In this paper, we have focused on resizing the caches underneath a single core only in order to study the main effects. But MCR can also be integrated into multicore CPUs as well. The multicore cache hierarchy with the most natural fit to MCR is one in which all caches are private. In this case, MCR can be applied to each "vertical slice" of the chip's cache system. The cache coherence protocol, which would most likely maintain coherence across the private last-level caches (LLCs), would need to be aware of cache resizing and only track cache blocks within active ways of each private LLC. But the sizing control for different cores' caches would not need to coordinate.

Most multicore CPUs today, however, employ shared LLCs. Integrating MCR into a processor with a shared LLC is also easy if the CPU employs cache partitioning. As discussed in Section 5, cache partitioning horizontally allocates portions of a shared cache to cores, providing each core with some number of cache ways. MCR can be applied *after* the partitioner makes its partitioning decision. Then, all that is needed is for the partitioner to inform MCR how many ways of the shared LLC it can resize up to. In that case, again there would be no interaction between MCR sizing control for different cores.

An interesting question is whether cache partitioning could be more effective if its horizontal allocation decisions were coupled with MCR's vertical allocation decisions. For example, if the partitioner knew that a program running on a particular core could more aggressively down-size its upstream caches given a smaller number of ways in the shared LLC, it might try to allocate more ways to other cores. Such “global” optimization that considers horizontal and vertical allocation interactions may ultimately provide the best power efficiency. Exploring such issues is a natural direction for future work.

7. Conclusion

This paper presents MCR, an architecture-level power management technique that resizes all caches in a modern cache hierarchy simultaneously. Our work shows a static-optimal version of MCR applied to a 3-level cache hierarchy can reduce total energy dissipation by 58.9% while degrading performance by only 4.4% across 22 SPEC CPU2006 benchmarks. We find a non-trivial portion of this gain–$\frac{1}{3}rd$ for the SPECint benchmarks–comes from optimizing inter-cache interactions. Our work also proposes several dynamic MCR techniques that can find the best sizing configurations at runtime. We show dynamic MCR can achieve between 40–62% energy savings while degrading performance by 7–9%.

References

[1] D. H. Albonesi, “Selective Cache Ways: On-Demand Cache Resource Allocation,” in *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, November 1999, pp. 248–259.

[2] R. Bai, N.-S. Kim, D. Sylvester, and T. Mudge, “Total Leakage Optimization Strategies for Multi-Level Caches,” in *Proceedings of the 15th ACM Great Lakes Symposium on VLSI*, 2005, pp. Chicago, IL.

[3] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, “Dynamic Memory Hierarchy Performance Optimization,” in *Proceedings of the workshop on Solving the Memory Wall Problem*, June 2000.

[4] R. Balasubramonian, D. H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, “A Dynamically Tunable Memory Hierarchy,” *IEEE Transactions on Computers*, vol. 52, no. 10, pp. 1243–1258, October 2003.

[5] D. Burger and T. M. Austin, “The SimpleScalar Tool Set, Version 2.0,” University of Wisconsin-Madison, CS TR 1342, June 1997.

[6] J. Chang and G. S. Sohi, “Cooperative Cache Partitioning for Chip Multiprocessors,” in *Proceedings of the International Conference on Supercomputing*, Seattle, WA, June 2007.

[7] S. Dropsho, A. Buyuktosunoglu, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, G. Semeraro, G. Magklis, and M. L. Scott, “Integrating Adaptive On-Chip Storage Structures for Reduced Dynamic Power,” in *Proceedings of 11th Annual International Conference on Parallel Architectures and Compilation Techniques*, 2002.

[8] K. Flautner, nam Sung Kim, S. Martin, D. Blaauw, and T. Mudge, “Drowsy Caches: Simple Techniques for Reducing Leakage Power,” in *Proceedings of the International Symposium on Computer Architecture*, Anchorage, AK, May 2002.

[9] A. Gordon-Ross, F. Vahid, and N. Dutt, “Automatic Tuning of Two-Level Caches to Embedded Applications,” in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE 04)*, 2004.

[10] G. Hamerly, E. Perelman, J. Lau, and B. Calder, “SimPoint 3.0: Faster and More Flexible Program Analysis,” in *Proceedings of the Workshop on Modeling, Benchmarking and Simulation*, June 2005.

[11] K. Kedzierski, F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu, and M. Valero, “Power and Performance Aware Reconfigurable Cache for CMPs,” in *Proceedings of the Second International Forum on Next-Generation Multicore/Manycore Technologies*, Saint-Malo, France, June 2010.

[12] C. Kim, J.-J. Kim, S. Mukhopadhyay, and K. Roy, “A forward body-biased low-leakage SRAM cache: device, circuit and architecture considerations,” *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 13, no. 3, pp. 349–357, 2005.

[13] C. H. Kim and K. Roy, “Dynamic Vth Scaling Scheme for Active Leakage Power Reduction,” in *Proceedings of the International Symposium on Design, Automation, and Test in Europe*, 2002, pp. 163–167.

[14] N. S. Kim, D. Blaauw, and T. Mudge, “Leakage Power Optimization Techniques for Ultra Deep Sub-Micron Multi-Level Caches,” in *Proceedings of the International Conference on Computer-Aided Design*, 2003.

[15] N. S. Kim, K. Flautner, D. Blaauw, and T. Mudge, “Circuit and Microarchitectural Techniques for Reducing Cache Leakage Power,” *IEEE Transactions on Very Large Scale Integration*, vol. 12, no. 2, pp. 167–184, February 2004.

[16] S. Kim, D. Chandra, and Y. Solihin, “Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture,” in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2002.

[17] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick, “ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems.”

[18] N. Madan, L. Zhao, naveen Muralimanohar, A. Udipi, R. Balasubramonian, R. Iyer, S. Makineni, and D. Newell, “Optimizing Communication and Capacity in a 3D Stacked Reconfigurable Cache Hierarchy,” in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2009.

[19] A. Malik, B. Moyer, and D. Cermak, “A Low Power Unified Cache Architecture Providing Power and Performance Flexibility,” in *Proceedings of the International Symposium on Low Power Electronics and Design*, Rapallo, Italy, 2000.

[20] (2011) Itrs working group models, mastar, http://www.itrs.net/models.html.

[21] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, “Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0,” in *IEEE/ACM INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE*. IEEE Computer Society, 2007, pp. 3–14.

[22] 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 *Proceedings of the International Symposium on Low-Power Electronics and Design*, August 1998, pp. Monterey, CA.

[23] M. Powell, S.-H. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar, “Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories,” in *Proceedings of the IEEE/ACM International Symposium on Low Power Electronics & Design*, 2000, pp. 90–95.

[24] M. K. Qureshi and Y. N. Patt, “Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches,” in *Proceedings of the International Symposium on Microarchitecture*, 2006.

[25] N. Shukla, R. Singh, and M. Pattanaik, “Design and Analysis of a Novel Low-Power SRAM Bit-Cell Structure at Deep-Sub-Micron CMOS Technology for Mobile Multimedia Applications,” *International Journal of Advanced . . .* , 2011.

[26] A. G. Silva-Filho and F. R. Cordeiro, “A Combined Optimization Method for Tuning Two-Level Memory Hierarcnhy Considering Energy Consumption,” *EURASIP Journal on Embedded Systems*, vol. 2011, September 2010.

[27] G. E. Suh, S. Devadas, and L. Rudolph, “A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning,” in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2002.

[28] G. E. Suh, L. Rudolph, and S. Devadas, “Dynamic Partitioning of Shared Cache Memory,” *The Journal of Supercomputing*, vol. 28, no. 7-26, 2004.

[29] K. T. Sundararajan, V. Porpodas, T. M. Jones, M. P. Topham, and B. Franke, “Cooperative Partitioning: Energy-Efficient Cache Partitioning for High-Performance CMPs,” in *Proceedings of the 18th International Symposium on High-Performance Computer Architecture*, New Orleans, LA, February 2012.

[30] J. Tschanz, S. Narendra, Y. Ye, B. Bloechel, S. Borkar, and V. De, “Dynamic sleep transistor and body bias for active leakage power control of microprocessors,” *Solid-State Circuits, IEEE Journal of*, vol. 38, no. 11, pp. 1838–1845, 2003.

[31] K. Varadarajan, S. K. Nandy, V. Sharda, and A. Bharadwaj, “Molecular Caches: A Caching Structure for Dynamic Creation of Application-Specific Heterogeneous Cache Regions,” in *Proceedings of the International Symposium on Microarchitecture*, 2006.

[32] S.-H. Yang, M. D. Powell, B. Falsafi, and T. N. Vijaykumar, "Exploiting Choice in Resizable Cache Design to Optimize Deep-Submicron Processor Energy-Delay," in *Proceedings of the 29th International Symposium on Computer Architecture*, San Diego, CA, June 2003.

[33] 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," in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, 2001.

[34] C. Zhang and F. Vahid, "Cache Configuration Exploration on Prototyping Platforms," in *Proceedings of the 14th International Workshop on Rapid Systems Prototyping*, 2003.

[35] C. Zhang, F. Vahid, and W. Najjar, "A Highly Configurable Cache Architecture for Embedded Systems," in *Proceedings of the 30th International Symposium on Computer Architecture*, San Diego, CA, June 2003.