To appear in the Proceedings of the 8th International Symposium on High-Performance Computer Architecture

Exploiting Choice in Resizable Cache Design to Optimize Deep-Submicron Processor Energy-Delay

Se-Hyun Yang, Michael D. Powell[†], Babak Falsafi, and T. N. Vijaykumar[†]

Computer Architecture Laboratory
Carnegie Mellon University
{sehyun,babak}@ece.cmu.edu

[†]*School of Electrical and Computer Engineering*
Purdue University
{mdpowell,vijay}@ecn.purdue.edu

Abstract

Cache memories account for a significant fraction of a chip's overall energy dissipation. Recent research advocates using "resizable" caches to exploit cache requirement variability in applications to reduce cache size and eliminate energy dissipation in the cache's unused sections with minimal impact on performance. Current proposals for resizable caches fundamentally vary in two design aspects: (1) cache organization, *where one organization, referred to as selective-ways, varies the cache's set-associativity, while the other, referred to as selective-sets, varies the number of cache sets, and (2)* resizing strategy, *where one proposal statically sets the cache size prior to an application's execution, while the other allows for dynamic resizing both within and across applications.*

In this paper, we compare and contrast, for the first time, the proposed design choices for resizable caches, and evaluate the effectiveness of cache resizings in reducing the overall energy-delay in deep-submicron processors. In addition, we propose a hybrid selective-sets-and-ways cache organization that always offers equal or better resizing granularity than both of previously proposed organizations. We also investigate the energy savings from resizing d-cache and i-cache together to characterize the interaction between d-cache and i-cache resizings.

1 Introduction

The ever-increasing level of on-chip integration in CMOS technology has enabled phenomenal improvements in microprocessor performance but has also caused an increase in energy dissipation in a chip. High energy dissipation diminishes the utility of portable systems and reduces reliability, requires sophisticated cooling technology, and increases cost in all segments of the computing market including high-end servers [11]. In state-of-the-art microprocessor designs, cache memories account for a significant fraction of total power/energy dissipation. For instance, 16% of total power in Alpha 21264 [3] and 21% in Pentium Pro [6] is dissipated in on-chip caches.

Current circuit techniques to reduce energy dissipation in caches typically trade off speed for lower energy dissipation in less performance-critical cache structures. Instead of solely relying on circuit techniques, recent research also advocates using "resizable" caches to reduce energy dissipation especially in high-performance caches [1,13]. Resizable caches are based on the observation that cache utilization varies *within and across* application execution. These caches allow hardware/software to customize the cache size to fit an application's demands. By eliminating energy dissipation in the cache's unused sections, resizable caches significantly improve energy-efficiency with *minimal* impact on application performance.

Current proposals for cache resizing fundamentally differ in cache organization, resizing framework, and how they exploit variability in applications' cache utilization to save energy. One proposal [1] advocates a *selective-ways* cache organization which allows for varying the cache's set-associativity. Another proposal [13] advocates *selective-sets* cache organization which varies the number of cache sets. These cache organizations differ in (1) the offered range of cache sizes, (2) the offered resizing granularity — i.e., the distance between two adjacent offered sizes, (3) the allowable set-associativity at various resizings, and (4) the hardware complexity. The effectiveness of either organizations to reduce size and energy depends on the one hand on the application's demand for a specific size and set-associativity and on the other hand the cache's ability to meet the demands.

The two proposals also differ in the cache resizing strategy of "when" to resize. The proposal for selective-ways [1] advocates *static resizing* by setting the cache size prior to an application's execution, and exploits variation in cache utilization only across applications. The proposal for selective-sets [13] advocates a *dynamic resizing* based on monitoring cache miss ratio and resizes the cache to react to varying demand for cache size both within and across applications. The two resizing strategies differ in two respects: (1) the ability to resize the cache during an application's execution, and (2) the design complexity. The effectiveness of dynamic resizing depends on both the resizing opportunity within applications and the ability of the dynamic resizing mechanisms to seize the opportunity.

The previous studies on resizable caches focused on a single cache design of interest, and did not compare and contrast the design choices for resizable caches. In this paper we identify the opportunity for cache resizing in a spectrum of applications, exploit the various design choices for both instruction and data resizable caches, and evaluate their effectiveness in reducing the *overall* energy dissipation in processors. We use Wattch [3] and SPEC benchmarks to simulate and model energy-delay for state-of-the-art processors and their cache hierarchies. We present results for optimal energy-delay, but show that the impact on overall performance is less than 3% in most of the experiments and less than 6% in all of the experiments.

The contributions of this paper are:

- **Resizing organization:** Selective-sets allows for maintaining set-associativity while resizing and offers superior energy-delay over selective-ways for caches with set-associativity of less than or equal to four. Selective-ways offers a better range of sizes and benefits caches with set-associativity of eight and higher. We propose a *hybrid selective-sets-and-ways* organization that always equals or improves energy-delay over the best of selective-sets or selective-ways alone.
- **Resizing strategy:** On average, static resizing captures most of the opportunity for resizing and reducing processor energy-delay in applications as compared to a miss-ratio based dynamic resizing framework while simplifying design. Dynamic resizing exhibits clear advantages over static resizing *only* in two scenarios: (1) when cache misses directly lie on the execution's critical path — e.g., instruction cache misses or blocking data cache misses — and the application exhibits varying working set sizes benefiting from resizing at runtime, or (2) the application's required cache size lies in between two sizes offered by the organization; unlike static resizing, dynamic resizing switches between two sizes and "emulates" the required size.
- **Resizing both d-cache and i-cache:** Our results indicate that resizing L1 d-cache and i-cache simultaneously has minimal impact on the application's footprint in L2 and therefore the cache resizing and energy-delay savings from the two caches are "additive". In a four-way out-of-order processor with 32K 2-way static selective-sets d-cache and i-cache and a 512K L2 cache, we measure an overall processor energy-delay savings of 20%.

The rest of the paper is organized as follows. Section 2 describes the design space of resizable caches, and in Section 3, we present energy dissipation in state-of-the-art cache memories and energy savings of resizable caches. In Section 4, we describe the experimental methodology and results. Section 5 presents an overview of the related work. Finally, we conclude the paper in Section 6.

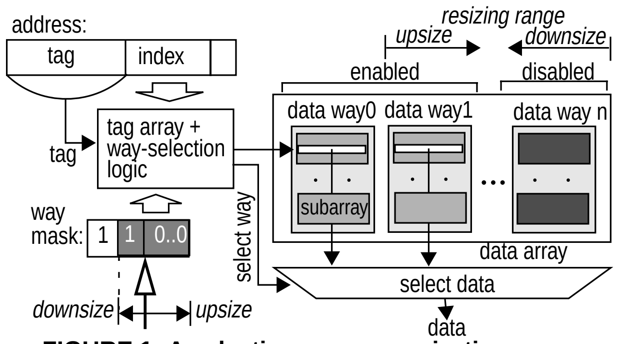


FIGURE 1: A selective-ways organization.

2 Resizable Caches

Resizable caches exploit the variability in cache size requirements in applications to save energy dissipation with minimal performance impact. Resizable caches save energy by enabling/disabling portions of the cache. To enable/disable cache sections, resizable caches exploit the cache subarrays, found in modern high-performance implementations. To optimize for cache access speed, cache designers divide the array of blocks into multiple subarrays of SRAM cell rows [12]. Resizing electrically isolates cache sections in multiple subarrays to save energy [1]. We will describe the details of energy savings in Section 3.

The basic cache organizations we study in this paper are derived from conventional RAM-tag caches, in which the tag and data arrays are organized as RAM structures. While CAM-tag caches (e.g., StrongARM [7]) have been shown to be more energy-efficient, they are typically limited to low-performance designs. While resizing in general is also applicable to CAM-tag caches, a study of resizable CAM-tag caches is beyond the scope of this paper.

Based on how they exploit the cache resizing opportunity in applications, resizable caches primarily differ in two respects: (1) cache resizing organization, dictating "which" cache dimensions are adjustable, and (2) resizing strategy (or time), dictating "when" the caches readjust these dimensions. In the rest of this section, we look at resizing organization and strategy one by one and also propose a hybrid organization.

2.1 Cache Resizing Organization

There are two proposals for resizable cache organizations, which we call *selective-ways* [1] and *selective-sets* [13]. Selective-ways allows enabling/disabling each individual associative way. Figure 1 depicts the basic structure of a selective-ways resizable cache. As in conventional set associative caches, at the higher level the data array is organized into cache ways. Each cache way consists of a number of subarrays. A *way-mask* allows enabling/disabling all the subarrays in a given way. Hardware or soft-

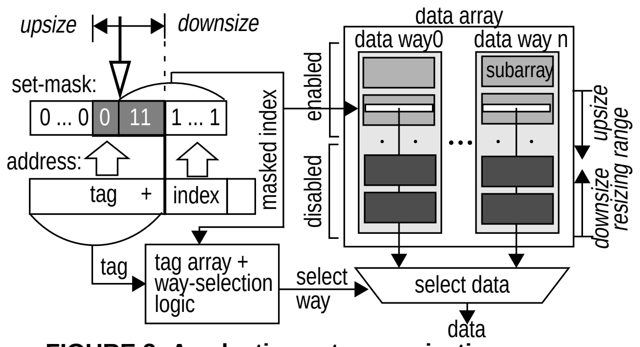


FIGURE 2: A selective-sets organization.

ware can adjust the number of ways the cache uses by setting the way-mask. The cache access logic uses the way-mask to identify which cache ways to access.

Alternatively, selective-sets allows enabling/disabling cache sets. Figure 2 depicts the anatomy of a selective-sets cache organization. In a conventional cache, the number of cache sets and the cache block size dictate the set of *index* and *tag* bits used to look up a cache block. Therefore, changing the number of cache sets changes both the required index and tag bits. Selective-sets provides a *set-mask* to allow varying the number of cache sets and the used index bits. Because enabling/disabling occurs in multiples of subarrays (Section 3), the minimum number of sets achievable is a single subarray per cache way.

There are fundamental differences between these organizations in their complexity and effectiveness. First, the two organizations differ in applicability and the range of cache sizes offered. Selective-ways changes cache size linearly in multiples of cache ways maintaining a constant resizing granularity. However, in high-performance caches (optimized for access time) which are often direct-mapped or use limited set-associativity, selective-ways is either not applicable or ineffective. Alternatively, selective-sets offers a better spectrum of sizes with low set-associativity. However selective-sets is limited when set-associativity is high, and is not applicable to fully associative caches.

Moreover, cache sizes offered by selective-sets are powers of two (due to the index-based set-mapping in conventional caches) allowing for fine-grain resizing only at smaller cache sizes. Therefore, selective-sets may be suboptimal when application working sets are large. Moreover, selective-ways changes set-associativity along with size and may miss the significant opportunity for resizing for memory reference streams with small working sets but high conflict miss rates. Selective-sets maintains set-associativity upon resizing increasing the opportunity for resizing for reference streams with high conflict miss rates.

A key advantage of selective-ways is its design simplicity. Selective-ways only requires an additional way-mask with corresponding logic. In contrast, selective-sets increases design complexity beyond the addition of a set-mask and its logic. Because resizing changes the number of tag bits, with smaller caches requiring a larger number of tag bits, selective-sets must use a tag array as large as that required by the smallest size offered. Therefore, using selective-sets, a cache of a given size requires a larger tag array which may be slower and dissipate more energy than selective-ways of the same size and set-associativity. Moreover, selective-ways does not change the set-mapping of cache blocks and as such obviates the need for flushing blocks in the enabled subarrays upon resizing. Selective-sets not only requires flushing modified blocks of disabling arrays, but also all blocks (clean or modified) for which set-mappings change upon enabling subarrays.

Size of each way	Set-associativity 4-way		3-way		2-way		dm
8K	32K	→	24K		16K		8K
4K	16K	↙ →	12K		8K		4K
2K	8K	↙ →	6K		4K		2K
1K	4K	↙ →	3K	→	2K	→	1K

Table 1: Enhanced resizing granularity using hybrid.

2.1.1 A Hybrid Organization

In this paper, we also propose and evaluate a hybrid selective-sets-and-ways organization for resizable caches. The key motivation behind a hybrid organization is that each of the resizable cache organizations offers a spectrum of cache sizes neither of which is a superset of the other. Selective-ways offers a spectrum of sizes that are multiples of a cache way size. Selective-sets offers a spectrum of sizes that are powers of two. A hybrid organization exploits the resizing granularity advantages of both organizations offering a richer spectrum of sizes than either organization alone, thereby optimizing energy savings by providing a size closest to the required demand for size by the application.

Table 1 illustrates cache size and set-associativities offered by a hybrid selective-ways-and-sets cache. For a 32K 4-way set associative cache and a subarray size of 1K, a hybrid cache offers all of 32K, 24K, 16K, 12K, 8K, 6K, 4K, 3K, 2K, and 1K sizes. Whereas, a selective-ways cache would only offer 32K, 24K, 16K, and 8K sizes (indicated by the first row) and a selective-sets cache would provide 32K, 16K, 8K, and 4K (indicated by the 4-way column). The table also depicts our simple resizing scheme. All the sizes between 32K and 3K simply go in steps between a 4-way and a 3-way configuration. For sizes less than 3K, the only configurations offered are those that further reduce set-associativity. This scheme follows the intuition that at higher cache sizes, capacity plays a bigger role than set-associativity while at lower cache sizes, set-associativity can significantly impact cache performance [5]. Downsizing from a 4-way 32K, our cache opts for a larger 24K size with a lower set-associativity of 3 ways rather than selecting a 4-way 16K cache as selective-sets would. Such an approach increases the resizing opportunity for applications with working set sizes closer to 24K than 16K.

Table 1 also indicates that a hybrid cache offers redundant sizes (shaded gray in the table). For instance, a 32K 4-way hybrid cache offers 16K with any of 4-way and 2-way set-associativities. In such cases, the hybrid cache offers the highest set-associativity to minimize miss ratio and optimize the utilization of block frames.

2.2 Cache Resizing Strategy

Besides organization, a key design choice in resizing is the strategy of "when" to resize. There are two proposals for resizing strategy. *Static resizing* [1] allows cache resizing prior to application execution, exploiting cache size variability across applications. Static resizing requires profiling an application's execution with different (static) cache sizes to determine the cache size with minimal energy dissipation and performance degradation. In static resizing, the application provides a cache size which the operating system loads into a programmable size mask (i.e., the way- or set-mask) prior to application's execution or upon a context switch.

Dynamic resizing [13] reacts to application demand for resizing to customize cache size and optimize energy savings during an application execution. Dynamic resizing uses extra hardware to monitor an application's execution and dynamically estimate performance and energy dissipation. When opportunity for resizing arises, dynamic resizing uses the cache size masks to resize the cache. In this paper, we evaluate a simple miss-ratio based dynamic resizing framework proposed in [13]. Hardware monitors the cache in fixed-length intervals measured in number of cache accesses. A miss counter counts the number of misses in each interval. At the end of each interval, hardware determines the need for cache resizing depending on whether the miss counter is higher or lower than a preset value, referred to as the *miss-bound*. To avoid thrashing, the framework prevents the cache from downsizing beyond a preset size, the *size-bound*. As in static resizing, the parameters are extracted offline through profiling.

Much like cache organization, there are fundamental differences in resizing strategy. Static resizing's key advantage is that it minimizes design complexity by fixing the size during an application's execution. When the application exhibits a fixed working set size, static resizing obviates the need for hardware monitoring and may achieve optimal energy reduction. Also, when cache miss latency is not exposed to the performance, regardless of

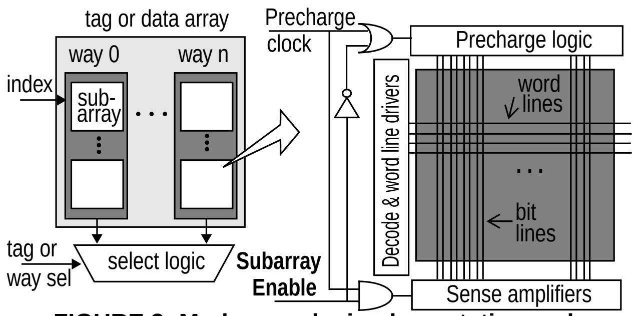


FIGURE 3: Modern cache implementation and energy saving technique.

the existence of working set size variation, the performance impact of misses created by static resizing's fixed size choice can be small and acceptable. Therefore, static resizing can downsize aggressively and save energy.

However, when there is working set variation within an application and the latency of additional misses directly affects performance, static resizing often fails to seize the opportunity and is suboptimal. In such a case, dynamic resizing may help optimize the energy savings and capitalize on its ability to capture the variation in working set size. Moreover, when the cache size required by an application lies between two sizes offered by the organization, dynamic resizing switches between the two sizes and emulates the required sizes with minimal impact on performance. Dynamic resizing, however, increases complexity and may require sophisticated hardware mechanisms to monitor and react to an application's change in behavior.

Dynamic resizing's effectiveness in reducing energy depends on the accuracy and timeliness of the mechanisms to react to an application behavior. In general, online estimation of opportunity for resizing is difficult when miss latency can be hidden and performance is *not* sensitive to simple cache performance metrics such as miss ratio. Inaccurate resizing may incur a large performance degradation due to large increases in the miss ratio. Dynamic resizing also incurs an increase in the miss ratio from flushing some of the cache blocks in the disabled/enabled subarrays upon resizing (Section 2.1). Furthermore, disabled subarrays may have included part of an application's primary working set, resulting in an increase in the miss ratio to bring the blocks back into the new enabled subarrays.

3 Energy Savings in Resizable Caches

In today's CMOS technology, the dominant source of power/energy is the switching energy dissipated in charging and discharging capacitive loads on bitlines. An increasingly important source of power/energy dissipation is the subthreshold leakage energy in future CMOS circuits [2] that aggressively scale down the transistor threshold voltage to reduce switching energy while maintaining high switching speeds. In this paper, we primarily focus on

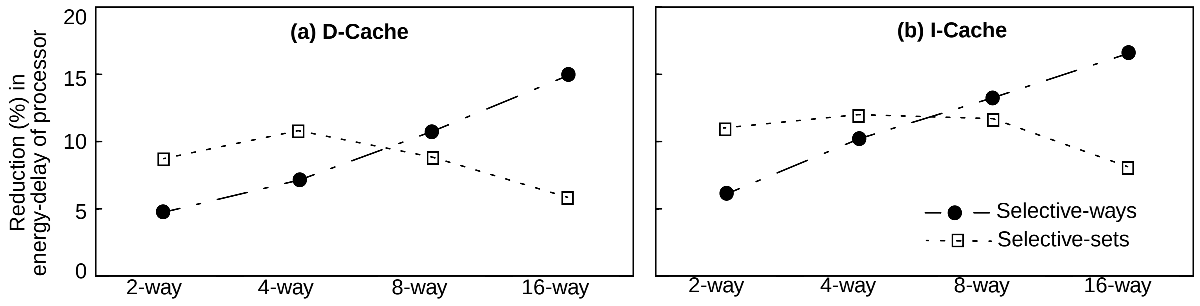


FIGURE 4: Resizable cache organizations and energy-delay reductions.

cache resizing as a technique to reduce switching energy dissipation. Because leakage energy dissipation is proportional to cache size [9], the results in this paper are also directly applicable to reducing leakage energy dissipation. However, a detailed analysis of the impact of cache resizing on leakage is beyond the scope of this paper.

Figure 3 depicts the structure of a modern cache implementation and the anatomy of a cache subarray. To optimize for access speed, cache designers divide the tag and data arrays into multiple subarrays of SRAM cell rows, each containing one or more cache blocks [12]. Modern high-performance cache designs precharge *all* the subarrays prior to a cache access, to overlap the precharging time with the address decode and wordline assertion. Unfortunately, in deep-submicron designs, precharged bitlines of *all* subarrays discharge through the pass transistors, even though only a small number of subarrays (equal to the cache's set-associativity) are actually accessed, leading to low energy efficiency. Precharging *only* the accessed subarrays to save energy requires either predicting the subarray to be accessed [8] or delaying the precharging until the address is available [4,7]. The latter, however, increases access time by as much as 30% as per CACTI simulations [12]. In this paper, we assume that all subarrays are precharged prior to access as modeled by Wattch [3]. A detailed study of techniques to trade off access time for energy is beyond the scope of this paper.

Instead, resizable caches select the appropriate cache size, disable the unused subarrays (Figure 3), and reduce switching energy by precharging only the enabled subarrays. Resizable caches are also able to eliminate unnecessary clock propagation to the disabled subarrays, achieving additional energy saving. Cache downsizing and (dynamic resizing's) block flushing between sense intervals increase activity in L2 and its energy consumption. However the increase is insignificant because energy per L2 access, that is less critical than L1 access, can be managed to be small using the techniques like delayed precharge. Additionally for selective-sets, L1 energy increases due to the extra resizing tag bits. This is also insignificant because the number of resizing tag bits is small (usually between 1 and 4) compared to the number of bit lines in a cache block (e.g., 256). In the result section, we report the energy consumption for the entire processor, so that we take all these factors into account.

Issue/decode width	4 intrs per cycle
ROB / LSQ	64 entries / 32 entries
Branch predictor	combination
writeback buffer / mshr	8 entries / 8 entries
Base L1 i-caches	32K 2-way; 1 cycle
Base L1 d-cache	32K 2-way; 1 cycle
L2 unified cache	512K 4-way; 12 cycles
Memory access latency	(80 + 5 per 8 bytes) cycles

Table 2: Base system configuration.

4 Results

In this section, we present the results in the comparisons of resizable cache's resizing organizations and strategies. We use Wattch 1.0 [3], which is an architecture-level power analysis tool built on top of SimpleScalar 3.0. Wattch reports both the execution time and the power/energy consumption of simulated processors. Table 2 shows our base system configuration parameters. We assume a 0.18µ technology and 1K subarray for L1 caches. Our base system has a power rating of 40.1W from Wattch.

We run SPEC benchmarks using reference inputs. We use *ammp, vortex,* and *vpr* from SPEC2000 and nine applications from SPEC95. For *gcc, ijpeg, m88ksim, su2cor, and tomcatv,* we simulate the entire runs; for the other applications we skip one billion instructions and run the next two billion instructions to reduce simulation turnaround time. On average, with our base configuration, d-cache accounts for 18.5% of total energy consumption for all these applications and i-cache accounts for 17.5%. Note that, unlike our average numbers here, reports on the power breakdown of commodity processors typically do not take into account activity factors of structures in the processors. Because there exist many structures that have smaller activity factors than caches, such as floating-point execution units, our average energy for overall cache structure appears larger than the numbers from such reports (e.g. 16% in Alpha 21264 [3].) With no activity

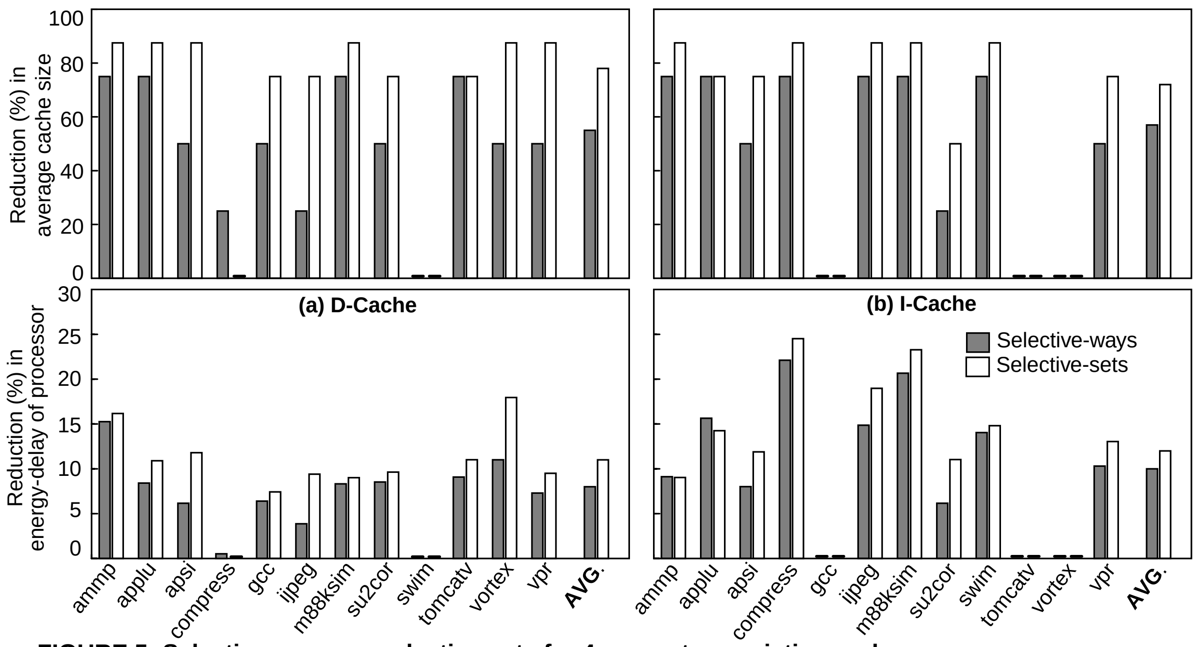


FIGURE 5: Selective-ways vs. selective-sets for 4-way set associative caches.

factor taken into consideration, our experiments also show that overall cache structure in the base system configuration accounts for only 18% of total processor power.

We use the energy-delay product of a processor to present the results because it is a well-established metric used in low-power research and ensures that both energy reduction and accompanying performance degradation are taken into account. For each design point, the relative energy-delay is obtained by normalizing its energy-delay with respect to that of the non-resizable cache with the same size and set-associativity. We always present the lowest energy-delay product achieved for each application regardless to the performance degradation. Nevertheless, all the lowest energy-delay products presented in this section are achieved within 6% of performance degradation and most of them, over 90% of the results presented, are achieved within 3% of degradation. Other configuration parameters are specified as they are varied in each section.

4.1 Cache Resizing Organization

In this section, we compare two resizing organizations, selective-ways and selective-sets. Based on our discussion in Section 2, we expect selective-sets to achieve the best relative energy-delay for high-performance low set associative caches by maintaining set-associativity for the applications with high conflict miss rate and by providing smaller minimum sizes for the applications with small size requirement. However, applications requiring finer granularity around the maximum size and having low conflict miss rate can benefit from selective-ways. Moreover, for highly associative caches, selective-ways provides larger spectrum of cache sizes in entire range and therefore is expected to achieve better energy-delay.

Figure 4 shows the reductions in processor's relative energy-delays of static selective-ways and selective-sets averaged for all the applications. Set-associativities of base caches range from 2-way to 16-way to include all the meaningful comparisons between two organizations of 32K size with 1K subarray. For d-caches, selective-ways reduces the energy-delays by 5% for 2-way, 8% for 4-way, 11% for 8-way and 15% for 16-way set associative cache, and selective-sets reduces by 9%, 11%, 9% and 6% in the same order. For i-caches, the numbers are 6%, 10%, 13% and 17% for selective-ways and 11%, 12%, 11% and 8% for selective-sets. The results indicate for both d-cache and i-cache that selective-sets achieves more reduction than selective-ways at low associativity but for 8-way or higher set associative caches, selective-ways is more effective.

Note that selective-sets achieves the best reduction at 4-way set associative cache, not at 2-way, although 2-way offers the best spectrum of cache sizes to this organization. It is well known that lower set associative caches produce more misses than higher set associative caches and the gap increases as the cache size decreases. Therefore, downsizing in higher set associative caches creates smaller number of misses, resulting in less performance impact and more aggressive downsizing. As far as the organization provides enough resizing granularity, higher set associative cache can downsize and save energy better, but selective-sets beyond 4-way does not offer enough granularity.

Also note that selective-sets on 2-way associative cache does not save as much energy as selective-ways on

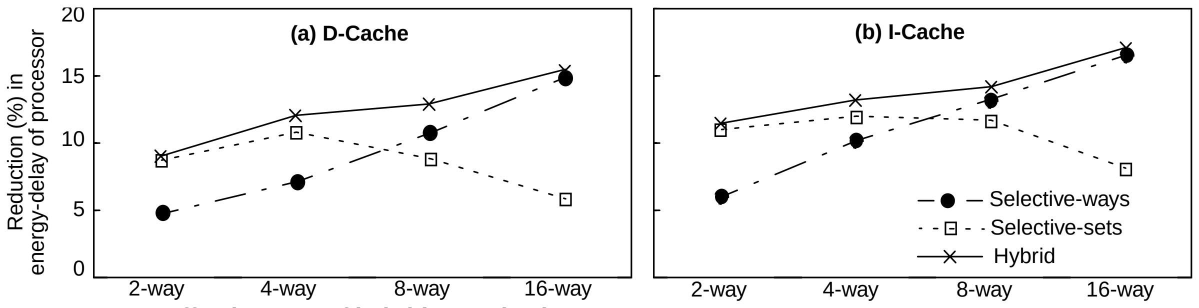


FIGURE 6: Effectiveness of hybrid organizations.

16-way cache does, although these two configurations have the best size spectrum on their own organizations. It is mainly due to different resizing granularity of two organizations. While selective-ways on 32K 16-way cache offers fine resizing granularity of 2K in entire range, selective-sets on 2-way offers fine grain resizing only at small sizes; no cache size is offered between 32K and 16K. Therefore, applications requiring cache sizes between 32K and 16K work more successfully with selective-ways. For instances, *compress* and *swim* for d-cache and *gcc*, *tomcatv* and *vortex* for i-cache belong to this type.

To investigate the resizing characteristics of each application on each organization, we present the reduction in cache sizes achieved by static selective-ways and selective-sets for 32K 4-way d- and i-caches in Figure 5. We include the average values at the end of each graph. To show the impact to the overall processor energy, we also present the reduction in processor's relative energy-delays. We use 4-way set associative cache because it provides a reasonable resizing granularity for both organizations. Specifically, selective-sets provides smaller minimum size (4K), while selective-ways offers better granularity between 32K and 16K. Note that although two applications have the same average cache size, their energy-delays would be different due to the difference in cache's energy contribution and resizing's performance impact.

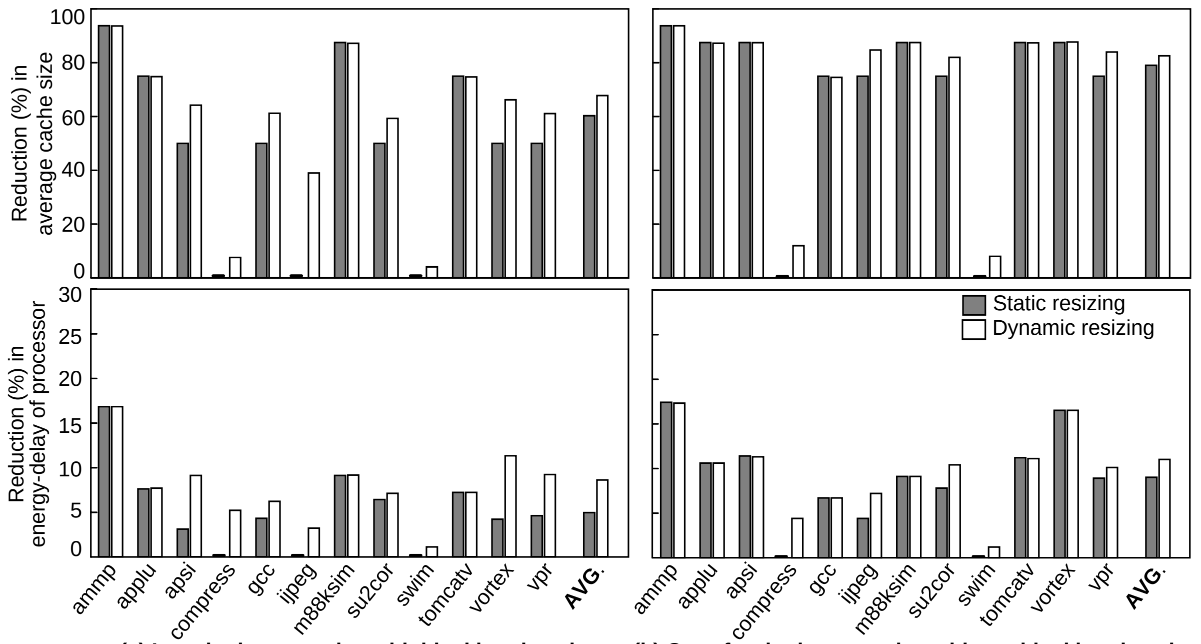
In d-caches, for ten applications out of twelve, selective-sets shows better energy-delay reduction than selective-ways. Six applications, *apsi*, *gcc*, *ijpeg*, *su2cor*, *vortex* and *vpr*, mainly benefit from selective-sets' ability to maintain set-associativity and prevent conflicts. Three applications, *ammp*, *applu* and *m88ksim*, require small cache sizes and take advantage of the smaller minimum size offered by selective-sets. For *compress*, selective-ways shows better energy-delay reduction than selective-sets, because the application requires granularity at large cache sizes offered by selective-ways but not by selective-sets. For *swim*, downsizing creates large amount of misses and large performance degradation, resulting in no downsizing for both organizations. *Tomcatv* reduces the cache size equally for both, but incurs larger performance impact with selective-ways due to more conflict misses.

For i-caches, *ammp*, *compress*, *ijpeg*, *m88ksim*, and *swim* require small cache sizes throughout execution and take advantage of the small minimum size available in selective-sets. *Apsi*, *su2cor* and *vpr* require set-associativity rather than cache size to keep the performance. Therefore, selective-sets exhibits better energy-delay reduction for them. For *applu*, selective-sets chooses the same cache sizes as selective-ways, but selective-ways dissipates less energy because lower set associative caches read fewer subarrays on each access (as many as set-associativity). *Gcc* and *tomcatv* have no cache downsizing because their working sets are larger than 32K and downsizing incurs large performance degradation.

4.1.1 Hybrid Organization

In this section, we investigate and evaluate hybrid organization. Figure 6 presents the average reduction in energy-delays for all three organizations including hybrid organization. The figure presents the set-associativities from 2-way to 16-way. The results show that hybrid organization achieves equal or better energy-delay reduction than both selective-ways and selective-sets in any set-associativities. On average, hybrid's energy-delay reductions for d-cache are 9% for 2-way, 12% for 4-way, 13% for 8-way and 15% for 16-way, and for i-cache, the numbers are 11%, 13%, 14% and 17%.

As we forecasted, there are two situations for which hybrid organization saves better than both the selective-ways and selective-sets. We do not show the result of each individual application, but for instances, for the applications like *compress*, *ijpeg*, *gcc*, *su2cor* in 4-way d-cache and *apsi*, *su2cor*, *ammp*, *swim*, *apsi* in 4-way i-cache, its better granularity plays a role and reduces energy-delays better. Hybrid offers better resizing granularity than either of selective-ways or selective-sets and therefore provides cache sizes closer to the actual cache size demand of the applications. The cache sizes utilized by these applications in the hybrid organization are supported by neither selective-ways nor selective-sets. Second, hybrid resizing offers small sizes less than the minimum sizes of selective-sets or selective-ways. For example, *ammp*, *applu*, and *m88ksim*, for 4-way d-cache, *ammp*, *compress*, *ijpeg*, *m88ksim*, and *swim* for 4-way i-cache exploit the smaller sizes.



(a) In-order issue engine with blocking d-cache (b) Out-of-order issue engine with nonblocking d-cache

FIGURE 7: D-cache resizing in two processor configurations.

4.2 Cache Resizing Strategy

In this section, we investigate static and dynamic resizing strategies based on two different processor configurations. Dynamic resizing presented here is a miss-ratio based strategy proposed in [13], implementation detail of which is described in Section 2.2. As we described, dynamic resizing is, in general, beneficial for applications with large variation in cache size requirement. Also, dynamic resizing has an advantage of emulating the cache sizes not offered by the organization.

Especially when miss latency is exposed to critical path of execution, suboptimal size chosen by static resizing that creates large number of misses during the execution incurs large performance degradation. Therefore, in this case, static resizing does not encourage cache downsizing. Instead, more accurate detection of working set variation, or using dynamic resizing, is required to achieve better energy savings without degrading performance.

However, when miss latency is hidden or has relatively less impact on performance, the cache resizing is more aggressive and downsizing is encouraged. Although static resizing incurs more misses from the program phases requiring larger cache sizes, these misses might not degrade performance significantly, therefore even static resizing downsizes aggressively, without hurting the performance. It, in turn, leaves smaller opportunity for dynamic resizing, and the effectiveness of dynamic resizing over static resizing is not as significant as when miss latency is highly exposed. Moreover, due to the misses possibly overlapped, miss ratio is not a good indicator of performance, and our miss-ratio based strategy is less effective to capture the cache size requirement.

To highlight the effect of miss latency exposure to the cache resizing, we compare cache resizings on two types of processor configuration: in-order issue engine with blocking d-cache and out-of-order issue engine with non-blocking d-cache. The former exposes d-cache miss latency to performance. Here, i-cache misses are relatively less critical to performance. However, the latter can highly exploit the instruction parallelism existing in applications to hide d-cache miss latency. Unlike d-cache, i-cache miss latency impacts the performance more directly in this configuration, being highly exposed to performance.

4.2.1 Resizing Data Caches

Figure 7 shows the reductions of energy-delay and average cache size by static and dynamic selective-sets for 2-way set associative d-cache on both types of processor configuration. We present only selective-sets because both organizations show similar results in this comparison. On average, with in-order issue processor, static resizing reduces 5% of total energy-delay, while dynamic resizing reduces 9%. Meanwhile, static resizing reduces 9% with out-of-order issue processor, and dynamic achieves 11%. In d-cache, cache resizing with out-of-order issue processor is more aggressive and achieves larger reductions.

With in-order issue engine and blocking d-cache, dynamic resizing exhibits larger reductions in cache sizes and energy-delays than static resizing, for eight applications. For these applications, the gap of average cache sizes between dynamic and static resizings is 16% on aver-

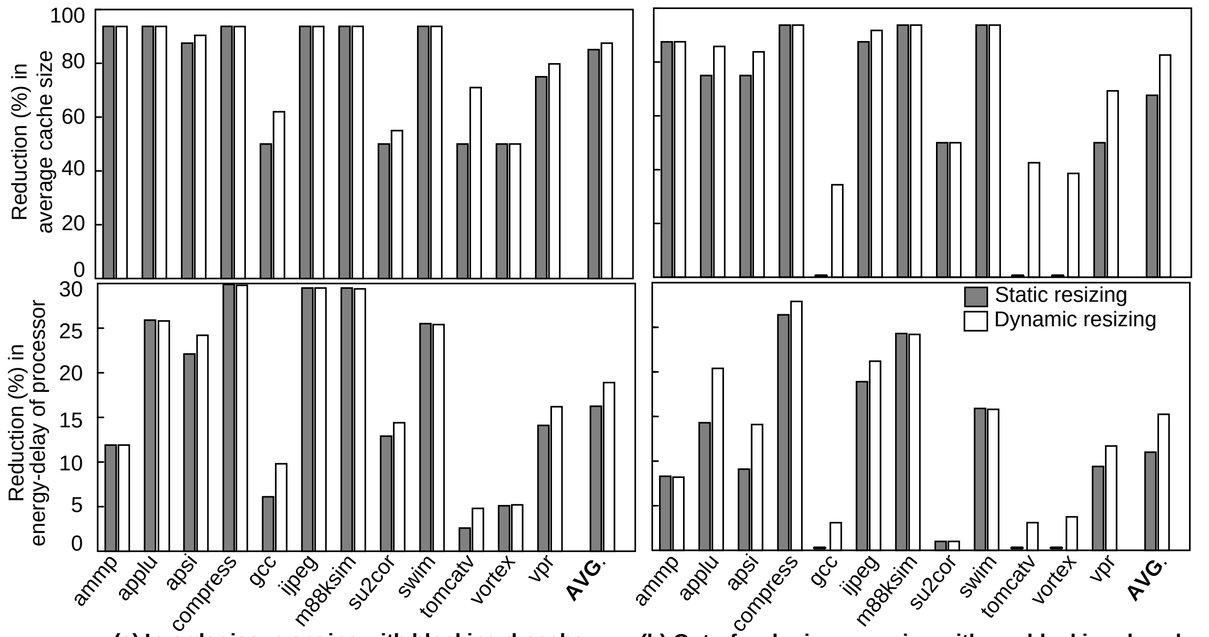


FIGURE 8: I-cache resizing in two processor configurations.

age with maximum of 38% in *ijpeg*. Also, the gap between energy-delays is 6% on average and as large as 8% in *vortex*. In this processor type, high exposure of d-cache miss latency to performance requires accuracy in capturing working set size variation. For *apsi*, *vortex* and *vpr* in in-order issue processor, static resizing achieves comparable cache size reduction to dynamic resizing — less than 15% gap between two but, interestingly, its energy-delay reductions are much less than half of dynamic resizing's results. It is because static resizing incurs relatively high performance impact that is close to 6%. It increases "delay" parts of the energy products relatively large, ending up with no significant reductions in energy-delay.

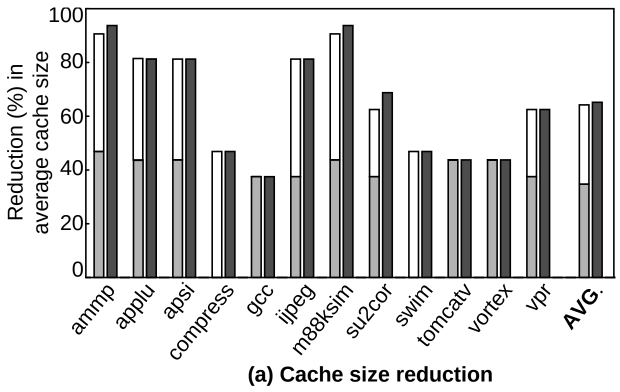
In contrast to the former processor type, the results with out-of-order issue engine and nonblocking d-cache show that dynamic resizing does not achieve significantly better savings. Aggressive superscalar engine with nonblocking d-cache exploits the parallelism between instructions and takes a lot of d-cache misses off of the critical path of application execution. As we mentioned, in such a case, aggressive downsizing is encouraged and static resizing possibly performs as good as dynamic resizing. We see even for the applications requiring variable cache sizes such as *apsi*, *gcc* and *vortex*, static resizing achieves as good reductions as dynamic resizing.

According to the dynamic resizing behavior, we group applications into three types. The simplest is those that have a constant size during the execution. For these applications, static and dynamic resizings show almost the same reductions. *Ammp*, *applu*, *m88ksim*, and *tomcatv* exhibit constant sizes. The next type of applications exhibits variation in working set size, indicated by changes in cache resizing behavior over many intervals. Examples of working-set variation in d-caches are *compress*, gcc, *vortex*, and *vpr. Su2cor* is an example of periodic variation in working set size as execution phases repeat. Dynamic resizing takes advantage of working-set size variation within these applications, especially with in-order issue engine and blocking d-cache.

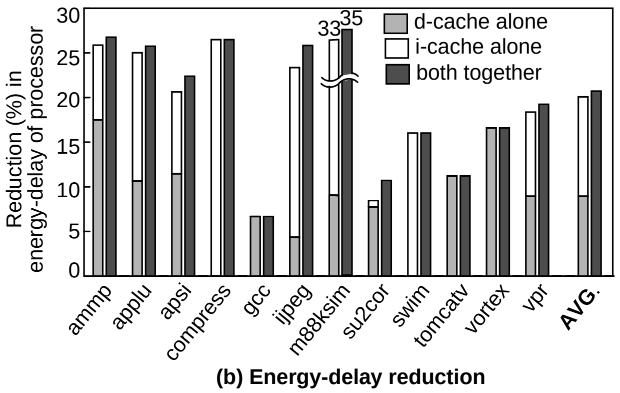
The third type is unavailable-size emulation, which occurs when the cache size required by the application is not offered. This type includes *apsi*, *compress*, *ijpeg*, and *swim* in in-order issue engine with blocking d-cache, and *compress* has the property of both second and third types. For the third type of applications, the resizable cache chooses cache sizes above and below the required size to achieve emulation. Unavailable-size emulation occurs because there is too much performance degradation at a smaller size but little degradation at a larger size. Unlike static resizing, dynamic resizing may be able to amortize the degradation by spending a while at the larger size. Additional sizes might be captured by using a hybrid selective-sets/selective-ways organization, but dynamic resizing's granularity is not limited by the organizations.

4.2.2 Resizing Instruction Caches

Figure 8 shows the reductions for 2-way set associative i-cache. With in-order issue engine and blocking d-cache, static resizing exhibits 16% of energy-delay reduction, while dynamic resizing reduces 18%. With out-of-order issue engine and nonblocking d-cache, static resizing reduces 11% of total energy-delay and dynamic reduces



(a) Cache size reduction



(b) Energy-delay reduction

FIGURE 9: Decoupled resizings on d-cache and i-cache.

by 15%. For i-cache, cache resizing with in-order issue processor achieves larger reductions.

In i-cache, in contrast to d-cache, dynamic resizing's ability to capture the working set size variation plays more important role with out-of-order issue engine and non-blocking d-cache, because i-cache miss latency is more exposed to performance when the processor exploits more parallelism on d-cache accesses. We see in out-of-order issue engine with nonblocking d-cache, dynamic resizing exhibits larger reductions in cache sizes and energy-delays than static resizing for seven applications. For these seven applications, the gap of average cache sizes between dynamic and static resizings is 31% on average and as large as 38% of *ijpeg*. Static resizing in in-order processor performs comparable to dynamic resizing, because i-cache misses in in-order processor exhibit less performance impact. Therefore static resizing can be aggressive and leaves smaller opportunity for dynamic resizing. Like d-cache resizing, i-cache resizing has three different types. The first type that exhibits a constant cache size throughout execution includes *ammp*, *compress*, *m88ksim*, *su2cor* and *swim*. For the second type, *applu*, *apsi* and *ijpeg* indicate periodic variation in i-cache working set size. The third type of behavior, unavailable size emulation, occurs in *gcc*, *tomcatv*, *vortex* and *vpr.*

Note that the same average cache size can result in different energy-delay reductions for two processor types because they have different breakdowns of the energy contribution. On average, energy contribution of i-cache in in-order issue processor is 21.5%, 4% larger than out-of-order issue processor.

4.3 Resizing Both Data and Instruction Caches

We have studied different aspects of resizable caches separately on d-cache and i-cache so far. In this section, we investigate the interaction between d-cache and i-cache resizings and the results of resizing them simultaneously.

In Figure 9, we present the reductions in average cache size and processor energy-delay achieved by resizing d-cache alone, i-cache alone and resizing both caches at the same time. As an example, we use static selective-sets with our base system configuration. In this figure, average cache size is normalized to the summation of base case d-cache and i-cache sizes. On average, simultaneous resizing reduces 20% of overall processor energy-delay. By stacking up the reductions from d-cache and i-cache resizings in one bar next to the result from simultaneous resizing, we easily see the additivity property; when we resize both caches together, the overall reductions in cache size and energy-delay are close to the summation of the reductions achieved by resizing each cache alone. Resizing both at the same time exhibits larger performance degradation, up to 10% but mostly less than 5%.

Moreover, there exist several applications exhibiting larger reduction from resizing both together than the sum of the reductions from resizing each individually. Downsizing one cache has an effect of shifting the bottleneck of overall performance close to itself, due to the additional misses in the cache. Therefore, the other cache can be downsized more aggressively resulting in less performance degradation with same downsizing than the case of downsizing itself alone.

Additivity property implies d-cache and i-cache resizings are decoupled from each other and we can study them separately expecting the additive energy savings when we apply resizing techniques for both at the same time.

5 Related Work

A number of previous studies have focused on architectural/microarchitectural techniques to reduce the energy dissipation in cache memories. Among them, recently there have been three proposals for cache resizing [1,10,13] two of which focus on reducing energy dissipation. Ranganathan, et al. [10], propose a statically resizing selective-ways d-cache and use it to partition the cache and use the unused part as auxiliary storage for instruction reuse information to increase performance. Albonesi [1] proposes a statically resizing a selective-ways cache, and evaluates the cache's effectiveness in reducing switching energy. Yang, et al. [13], propose a dynamically resizing selective-sets i-cache, and evaluate its effectiveness to reduce leakage energy dissipation. In this paper, we draw *key* resizing architectural design aspects from Albonesi's and Yang, et al.'s proposals, to evaluate effectiveness of and opportunity for cache resizing to reduce energy dissipation. We also consider overall energy dissipation including both switching and leakage energy, and propose a hybrid cache organization and that exploits advantages of both selective-ways and selective-sets.

6 Conclusions

Using a cycle-accurate performance and energy simulation tool, we studied and compared the merits of the resizable cache's two design aspects: cache resizing organization and resizing strategy. For organization, our results showed that selective-sets offers better energy-delay over selective-ways for caches with set-associativity of less than or equal to four, by maintaining set-associativity upon resizing. Meanwhile, selective-ways benefits caches with set-associativity of eight and higher. We proposed a hybrid selective-sets-and-ways organization that always equals or improves energy-delay over the best of selective-sets or selective-ways alone.

For cache resizing strategy, we showed that on average, static resizing captures most of the opportunity for resizing and reducing processor energy-delay in applications as compared to a miss-ratio based dynamic resizing framework with minimal design complexity incurred. Dynamic resizing exhibits clear advantages over static resizing only in two scenarios: (1) when cache misses directly lie on the execution's critical path and the application exhibits varying working set sizes benefiting from resizing at runtime, or (2) the application's required cache size lies in between two offered sizes by the cache organization; unlike static resizing, dynamic resizing switches between the two sizes and emulates the required size.

Our results also indicated that resizing L1 d-cache and i-cache simultaneously has minimal impact on L2's footprints and therefore the cache resizing and energy-delay savings from the two caches are uncorrelated and additive. In a four-way out-of-order processor with 32K 2-way static selective-sets d-cache and i-cache and a 512K L2 caches, we measured an overall processor energy-delay savings of 20% on average.

Acknowledgments

This research is supported in part by SRC under contract 2000-HJ-768. We would like to thank Yuen Chan and the anonymous reviewers for their useful comments.

References

[1] D. H. Albonesi. Selective cache ways: On-demand cache resource allocation. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 32)*, pages 248–259, Nov. 1999.

[2] S. Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, July 1999.

[3] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 83–94, June 2000.

[4] J. H. Edmondson, et al. Internal organization of the Alpha 21164, a 300-MHz 64-bit quad-issue CMOS RISC microprocessor. *Digital Technical Journal*, 7(1), 1995.

[5] M. D. Hill. A case for direct-mapped caches. *IEEE Computer*, 21(12):25–40, Dec. 1988.

[6] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: Speculation control for energy reduction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 132–141, June 1998.

[7] J. Montanaro, et al. A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor. *IEEE Journal of Solid-State Circuits*, 31(11):1703–1714, 1996.

[8] M. D. Powell, A. Agrawal, T. Vijaykumar, B. Falsafi, and K. Roy. Reducing set-associative cache energy via selective direct-mapping and way prediction. In *Proceedings of the 34th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 34)*, Dec. 2001.

[9] M. D. Powell, S.-H. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar. Gated-V_{dd}: A circuit technique to reduce leakage in cache memories. In *Proceedings of the 2000 International Symposium on Low Power Electronics and Design (ISLPED)*, pages 90–95, July 2000.

[10] P. Ranganathan, S. Adve, and N. P. Jouppi. Reconfigurable caches and their application to media processing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 214–224, June 2000.

[11] D. Singh and V. Tiwari. Power challenges in the internet world. Cool Chips Tutorial in conjunction with the 32nd Annual International Symposium on Microarchitecture, November 1999.

[12] S. J. E. Wilson and N. P. Jouppi. An enhanced access and cycle time model for on-chip caches. Technical Report 93/5, Digital Equipment Corporation, Western Research Laboratory, July 1994.

[13] 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 Seventh IEEE Symposium on High-Performance Computer Architecture*, Jan. 2001.