Improving Cache Efficiency via Resizing + Remapping

Technical Report: GIT-CERCS-07-18
Subramanian Ramaswamy, Sudhakar Yalamanchili
Computational Architectures and Systems Laboratory
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332
ramaswamy@gatech.edu, sudha@ece.gatech.edu

Abstract

In this paper we propose techniques to dynamically downsize or upsize a cache accompanied by cache set/line shutdown to produce efficient caches. Unlike previous approaches, resizing is accompanied by a non-uniform remapping of memory into the resized cache, thus avoiding misses to sets/lines that are shut off. The paper first provides an analysis into the causes of energy inefficiencies revealing a simple model for improving efficiency. Based on this model we propose the concept of "folding" - memory regions mapping to disjoint cache resources are combined to share cache sets producing a new placement function. Folding enables powering down cache sets at the expense of possibly increasing conflict misses. Effective folding heuristics can substantially increase energy efficiency at the expense of acceptable increase in execution time. We target the L2 cache because of its larger size and greater energy consumption. Our techniques increase cache energy efficiency by 20%, and reduce the EDP (energy delay product) by up to 45% with an IPC degradation of less than 4%. The results also indicate opportunity for improving cache efficiencies further via cooperative compiler interactions.

1 Introduction

The shift from scaling frequency to scaling the number of cores continues the trend of stressing off-chip memory bandwidth and reliance on on-chip caches. However the costs of larger caches are significant and growing. They typically occupy 40–60% of the chip area [14] and with leakage power exceeding switching power at sub-micron technologies [11, 10], they are dominant consumers of energy. Furthermore, analysis of benchmarks have shown that cache utilization is typically low - below 20% for a majority of benchmarks [8, 4] with performance efficiency averaging 4.7% and energy efficiency averaging 0.17%!(Section 2). This is in large part due to the fact that the majority of the cache (especially L2 and L3) is idle most of the time but contributes significantly to leakage power. The state of the practice in making caches more energy efficient has been to power down cache components such as cache lines, sets or ways - turn them off or maintain them in a low voltage state (examples include Albonesi [3], Abella *et al*.[2], Kaxiras *et al*.[9], Powell *et al*.[12], Zhang. C *et al*.[19], Zhang M. and Asanovic [20], Zhou *et al*.[21], Flautner *et al*. [7], and, Abella and González [1].) Strategies focus on *when* to turn of *which* components. Poor decisions lead to expensive misses and power up events and therefore strategies tend to be conservative.

In this paper we propose an evolution to the dynamic scaling of cache resources for improving both cache performance and power efficiency. Resource scaling differs from prior approaches such as those identified earlier in that the subset of active cache resources represent fully functional caches - segments that are turned "off" will not be referenced, and neither is the scaling uniform, i.e., certain memory lines will have relatively less cache resources allocated. We view resource scaling as a natural extension to the prior techniques for intelligently turning off portions of the cache for short periods of time, and which requires solving a basic problem of mapping all of memory into the scaled down cache or a scaled up cache i.e., computing a new placement function. We combine the abstraction of *conflict sets* [13] with run-time reference counts to realize simple (hardware) techniques to dynamically resize the cache to a subset of active components and recompute the placement function. We target the L2 cache because of its larger size and consequently greater impact on energy consumption.

The paper analyzes the causes of energy inefficiencies revealing a simple model for improving energy efficiency. Based on this model we propose the concept of *folding* - memory regions that normally map to disjoint cache resources are combined to share one or more cache sets pro-

ducing a new placement function. Folding enables powering down cache sets at the expense of possibly increasing conflict misses. The concepts of resizing and folding can be applied in different ways; for example, sizing and folding can be applied statically if memory access patterns are known at compile-time as happens with many scientific computing kernels. In this paper, we focus on applying folding strategies at run-time. Effective folding heuristics can substantially increase energy efficiency at the expense of an acceptable increase in execution time. Section 2 describes the model and metrics for computing cache efficiency and the application to a set of benchmarks. The resulting insights lead into several folding heuristics described in Section 3. The proposed heuristics are evaluated in Section 4 and the paper concludes with summary remarks and directions for future research.

2 An Efficiency Model for Caches

2.1 The Model

At a clock cycle, a cache line may be *active* (powered) or *inactive* (turned off). Thus, in the absence of energy management, all lines are active. Further, an active cache line is *live* on a clock cycle if it contains data that will be reused prior to eviction, and is *dead* otherwise [4, 9]. Thus, on any clock cycle, a cache line will be live, dead, or inactive. Thus the $(L*T)$ cycles expended for a L line cache over T cycles is the sum of *live cycles*, *dead cycles*, and *inactive cycles*, with each cache line contributing one live, dead, or, inactive cycle per clock cycle.

Cache utilization,η_u is the average percentage of cache lines containing live data at a clock cycle [4, 9], and is computed per Equation 1. The effectiveness of the cache, E, is computed as the percentage of cache cycles devoted to live or inactive lines as shown in Equation 2 where $(total\_cycles)$ is the program execution time. This metric captures the effectiveness of strategies for programmed shutdown - the higher the value the greater the percentage of the active cache that retains live data. The most effective scheme is one where all cache cycles are either live or inactive. In the absence of any energy management - effectiveness is equivalent to utilization. Equation 2 and Equation 3 are equivalent because the number of active cycles is the sum of the number of dead cycles and live cycles.

$$\eta_u = \frac{\sum_{i=0}^{i=L-1} live\_cycles_{line_i}}{\sum_{i=0}^{i=L-1} active\_cycles_{line_i}} \tag{1}$$

$$E = 1.0 - \frac{\sum_{i=0}^{i=L-1} dead\_cycles_{line_i}}{total\_cycles * L} \tag{2}$$

$$E = \frac{\sum_{i=0}^{i=L-1} (live\_cycles_{line_i} + inactive\_cycles_{line_i})}{\sum_{i=0}^{i=L-1} (active\_cycles_{line_i} + inactive\_cycles_{line_i})} \tag{3}$$

Effectiveness as defined can be increased at the expense of a high miss rate. For example, shutting down all except one line in a direct mapped cache can produce high effectiveness for structured accesses. An efficient cache should be effective with minimal compromises in execution performance. Cache *performance efficiency,* η_p, is defined in Equation 4 as the product of effectiveness and a scaling factor shown below, where t_c is the cache access time, t_p is the miss penalty, and, m is the miss rate. A cache has 100% performance efficiency if it does not contribute any dead cycles, and has a 100% hit rate.

$$\eta_p = E * \frac{t_c}{t_c + m * t_p} \tag{4}$$

Energy efficiency, η_e, is the ratio of *useful work* to total work. We regard useful work as the cache switching energy expended in servicing a cache hit. The total work is the sum of the switching energy consumed during all cache accesses (hits and misses), and leakage energy. A cache will have an energy efficiency of unity, if all energy consumed by the cache equals the switching energy consumed during cache hits. Energy efficiency is defined in Equation 5, where sw_{energy} represents the switching energy, and $leak_{energy}$ represents the leakage energy.

$$\eta_e = \frac{sw_{energy} * num_{hits}}{sw_{energy} * (num_{hits} + num_{misses}) + leak_{energy}} \tag{5}$$

Although cache sets/lines may be turned off to reduce leakage energy losses, if additional misses are created as a result, program execution time increases. The increase in execution cycles leads to greater energy consumption by all active lines and therefore the choice of lines to turn off are critical. Effectiveness and utilization can be expected to drop as misses generally increase the percentage of dead cycles due to conflict misses with a live cache line. An important question now becomes the line size - will smaller line size enable a finer degree of control over energy dissipation? Can we exercise this control effectively? The preceding model provides a simple classification of cache cycles that can be used to reason about cache efficiency, especially, as it relates to energy efficiency. This paper does not explore the impact of cache parameters on the model parameters. Rather we use this model to identify good folding heuristics. An analysis of benchmark kernels towards that end is provided in the following section.

2.2 Empirical Analysis

We simulated the execution of benchmarks from the SPEC2000 [18], Olden [15] and DIS [6] suites using the $SimpleScalar$ [17] simulator which was modified to obtain cache efficiency. We used simulation windows for SPEC programs given in the study performed by Sair and Charney [16]. Our energy estimates were derived using $Cacti$ 4.2 [5] for $70nm$ technology. We assume the L2 cache access latency to be fixed at 15 cycles independent of the size and associativity of the cache. Varying the cache latency affected execution time by less than 2%. Our definition assumes that the switching energy for a read is the same as that of a write. This artificially increases energy efficiency because switching energy for a write is lower than that for a read, as only one bank has to be accessed for a write compared to all banks for a read. Leakage constitutes 95% of the total cache power at $70nm$ for cache sizes at 256KB or greater [5], and cache writes constitute a small fraction of the total number of accesses, therefore this assumption affected efficiency by less than 1%. Finally, the energy was calculated with the cache operating at the highest expected frequency as given by $Cacti$.

Cache utilization for a 256KB 8-way L2 cache with 128-byte lines averages 24% and that for the L1 cache averages 12% as seen from Figure 1. The utilization for the L2 cache is higher than the L1 cache because the L2 cache is able to house data structures that conflict in the L1 due to its larger size for some of the applications (gzip, field etc.) The utilization numbers suggest that both cache levels maintain more *dead* lines than *live* lines, and therefore the majority of cache costs are spent in maintaining data that will not be re-used. Performance efficiency is shown in Figure 2 and averages 4.7% for the L2 cache. Cache energy efficiency with current designs averages 0.17% as observed from Figure 3 with leakage being the primary source of inefficiency.

Utilization captures the temporal residency of live data in the cache. Performance efficiency captures how well this residency in the cache is exploited. Thus, a live line that is accessed 10 times during its period of residence is more efficient than if it was accessed only twice during its period of residence. Energy efficiency simply captures the percentage of overall energy that is devoted to useful work which in this paper is regarded as servicing cache hits.

The sensitivity of energy and performance efficiencies to cache sizes and associativity averaged across all applications is shown in Figures 4 and 5. Efficiencies drop with cache size, and increase slightly with associativity. Larger cache sizes can increase utilization if application footprints fit in the cache, however as cache sizes are increased further, the percentage of dead lines increases which lowers efficiency. Thus, we see a drop in efficiency for higher caches. When associativity is increased, miss rates are lowered with a resultant drop in execution time which improves

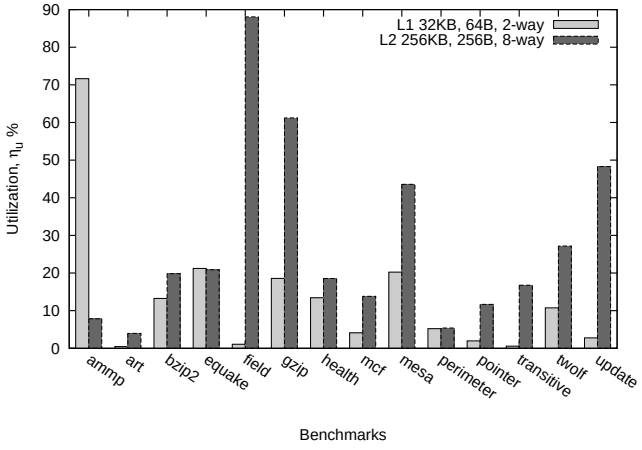

Figure 1. Cache utilization with current designs

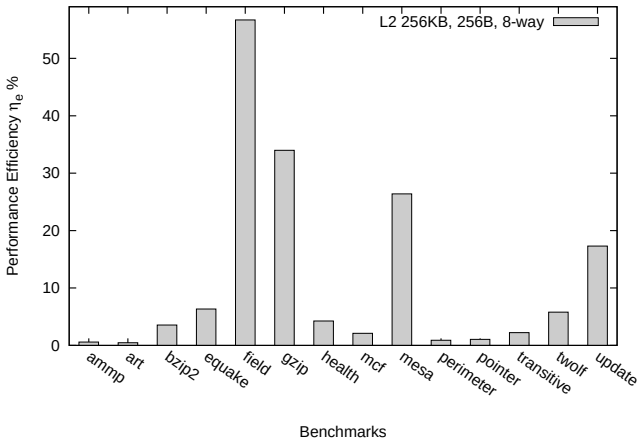

Figure 2. Cache performance efficiency with current designs

efficiency. However, each *dead* cache line stays longer in the cache due to the deeper LRU stack and the added dead cycles can compensate for any utilization gains. Accordingly, performance efficiency remains almost flat with associativity. Energy efficiency improves slightly with associativity due to switching energy increases and reductions in execution time, and is an artifact of the manner we define energy efficiency. Cache line sizes also had a limited impact on efficiencies and varied less than 5% for a range of 128 to 512 byte line sizes.

The primary source of inefficiency from an energy per-

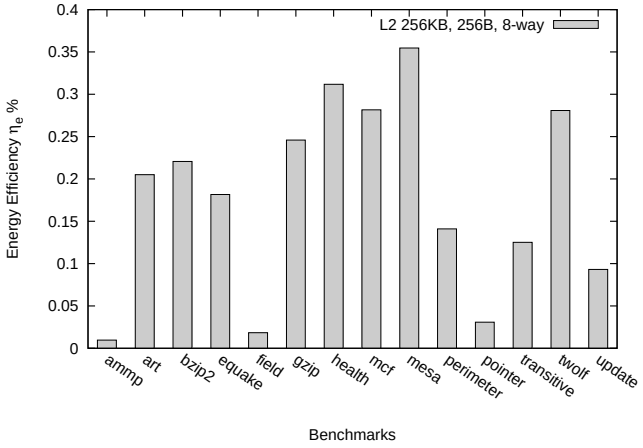

Figure 3. Cache energy efficiency with current designs

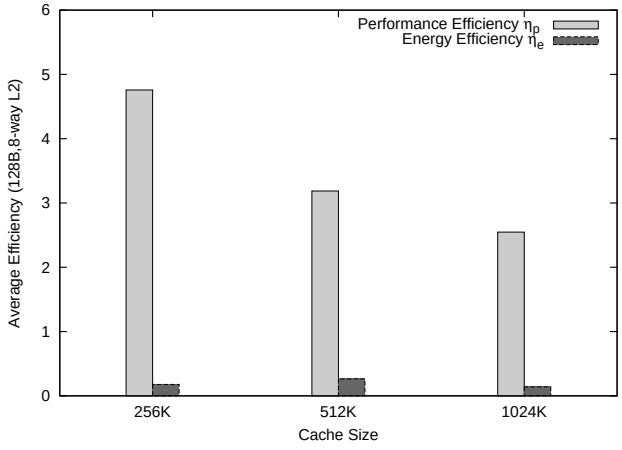

Figure 4. Efficiencies as a function of cache size

spective is that the number of accesses/cycle to an L2 is very low. Detailed analysis of the reference behavior of benchmark kernels [2] have identified inter-reference intervals from thousands to tens of thousands of clock cycles - during which time millions of cache transistors remain powered up. The goal now is to dramatically scale back the cache size while remapping memory to the smaller cache. This scaling (up or down) happens periodically to match the program reference behavior that is modeled as described in the following section.

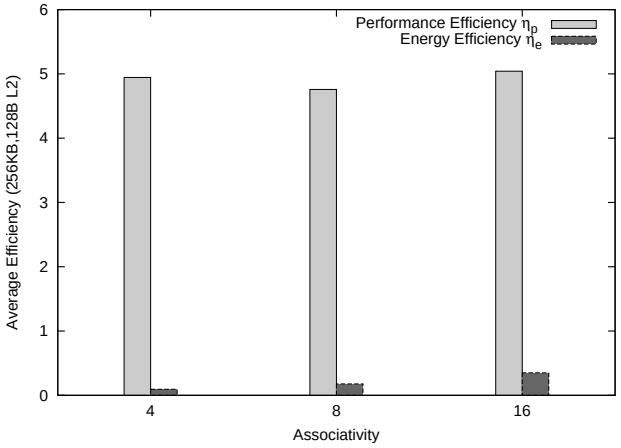

Figure 5. Efficiencies as a function of associativity

2.3 Improving Efficiency

The sharing of cache resources by memory lines is represented by the construction of *conflict sets*, where a *conflict set* is the set of main memory lines that is mapped to a cache set. The conflict sets in modern caches are constructed using *modulo* placement, where a memory line at address L is placed in the cache set L mod S, with S sets in the cache.

The key to improving cache energy and performance efficiencies is sharing the cache among memory lines efficiently. For this, we advocate extending the notion of liveness of program variables to main memory lines. A memory line is *live* if any of the variables resident in the memory line are live. Ideally, during the execution of a procedure or function, if the set of live memory lines are resident in the cache all remaining cache lines can powered off. The cache resizing implementation would ideally track the size of the set of live memory lines at any point during the program execution. This paper captures our initial application of this view where we apply heuristics to resize the cache in an attempt to track the live range as represented by the generational model of caches [8].

The concepts of liveness, resizing, folding and conflict sets can be applied at multiple program execution points. They may be applied statically, if information regarding access behavior is available at compile-time, or they may be applied as a one-time reconfiguration technique for embedded processors that run pre-defined workloads, etc. In this paper, we apply these techniques at runtime to improve cache energy and performance efficiency.

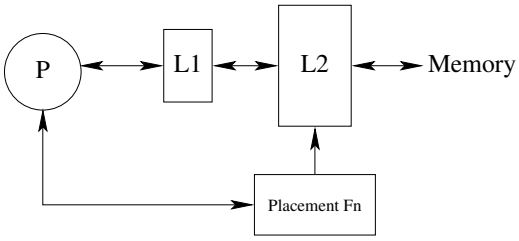

Figure 6. Architecture Model

3 Cache Resizing

If two conflict sets have non-overlapping live ranges, the two may be merged and a cache set corresponding to one of those sets can be turned off without any increase in the number of misses. This merging of conflict sets is referred to as *folding*. Folding requires remapping a set of main memory lines to a new set. Conversely, *splitting* is the reverse process and is accompanied by a corresponding upsizing of the cache. In this paper splitting is confined to folded conflict sets.

3.1 Customizing Cache Placement

Re-constructing conflict sets alters the mapping from memory lines to cache sets, i.e., the cache placement is altered. Conflict set folding, splitting and cache sizing customize the placement for energy and performance. All conflict sets do not necessarily have the same cardinality as opposed to conflict sets in traditional caches which have the same cardinality. That is, the number of memory lines contained in a conflict set may vary over time using customized placement. A newly created conflict set by merging original conflict sets has a higher cardinality than other conflict sets which were not altered.

Associated with each set is a reference counter that is used to approximate the "liveness" of the lines in a cache set. This reference count is used in the heuristics described later in this section.

3.2 Architecture Model

Customized placement is implemented for the L2 cache in an uni-processor architecture with L1 and L2 caches as shown in Figure 6. The L2 access is overlapped with the L1 access - details for the individual heuristics are provided in the following section. Operationally, when a down sizing or upsizing operation is be performed a remapping table must be reloaded with address translations - this is performed in software.

Finally, we assume the existence of the ability to turn off cache lines using the *Gated-Vdd* approach proposed by Powell *et al.*[12] which enables turning off the supply voltages to caches lines, and has an additional area cost of 3%. A wide transistor in the supply voltage or the ground path of the cache's SRAM cells is introduced which reduces leakage because of the stacking effect of the self reverse-biasing series-connected transistors.

3.3 Folding Heuristics

3.3.1 Decay Resizing

This technique is an extension to the cache decay strategy proposed by Kaxiras *et al.*[8]. Based on a 4-bit counter value, cache sets are turned off. If a cache set is accessed while it is turned off, a cache miss results and the cache set is powered back on during the servicing of the miss.

Our technique extends this by folding conflict sets that that have long periods of inactivity or little activity. Fine grained control at the granularity of cache sets is possible through this approach. The hardware implementation consists of a look up table which contains the new set indexes as shown in Figure 7. On an L2 cache access, the lookup table is accessed for the new set index, which may be different from the original set index if the conflict set corresponding to the original set index was one of those chosen for merging. The look up table access can be performed in parallel with L1 access to save latency, however, the serial access add only a single cycle which increases execution time by only 1%.

Parameters that can be tuned include the frequency at which the counter is sampled and the number of bits in the counter. We experimented with two and four bit counters, and turn off intervals were varied from 128K to 512K cycles. Our technique merges all unused conflict sets to two or four conflict sets, which is again a design parameter that was explored. Our technique splits an original conflict set from the merged conflict set if there are multiple accesses (four) to the original conflict set within the sampling period. Eager write-backs of dirty lines are required whenever the cache is resized. Two lines with the same tag can now map to the same cache set. Therefore to ensure unique tags across all memory lines that map to a cache set, the new tags are comprised of the old tags concatenated with the index bits. For a 256KB 8-way L2 with 128 byte lines, the extended tag adds another 8 bits per cache set or a total of 256 bytes additional overhead to the entire cache which is insignificant compared to the cache size.

3.3.2 Power of Two Resizing

A simple version of customized placement is downsizing or sizing the cache upwards in powers of two. Powell *et al.*[12] adopt such a mechanism for instruction caches where the active cache size is increased or decreased based

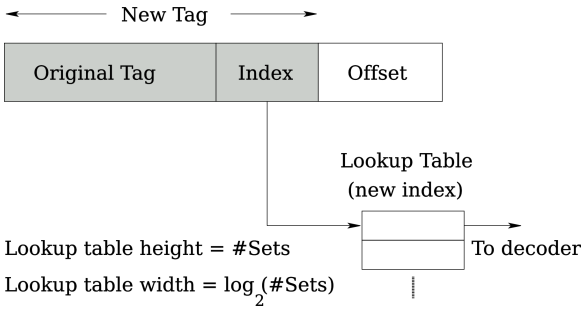

Figure 7. Cache Decay Resizing

on miss rates being within a certain bound determined by profiling. The advantage of this strategy is the simple address translation mechanism which uses simple bit masking as shown in Figure 8. We adopt this scheme to unified L2 caches. The number of references for a time interval is used as our basis for sizing the cache. If the references within a particular window are lower than a preset threshold, the cache can be downsized, and if it is greater, the cache can be resized upwards, with eager write-backs of dirty lines on both occasions. Parameters that can be varied include the thresholds and the turn off intervals, or using the hit count in lieu of the reference count. This scheme creates a smaller number of uniformly sized denser conflict sets upon downsizing.

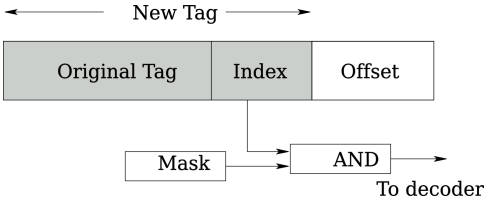

Figure 8. Power of Two Resizing

3.3.3 Segment Resizing

In this approach, the cache is split into segments that are a power of two, where a segment is a group of contiguous cache sets. For example, a cache with 256 sets can be split into 8 segments of 32 sets each, or 16 segments of 16 sets, and so on. Conflict sets from one segment are folded with those in another segment based on reference count within a time interval. For example, if there are eight segments and 256 conflict sets originally, conflict sets belonging to two segments can be folded resulting in 224 conflict sets, allowing 32 cache sets to be turned off.

This strategy sizes the cache on utilization. For example, if the utilization was 12.5%, and if the cache was divided into eight segments, ideally only one of the eight segments has to be turned on if the application footprint in the cache is contiguous. On the other hand, if the application footprint was small but non-contiguous, finer grained approaches are better suited.

The incoming address is split into a segment offset and the segment index. The lookup table is indexed using the original segment index to identify the new segment. The new segment index concatenated with the segment offset gives the new cache set index to be sent to the cache decoder. A simple implementation is to merge all segments that are under-referenced to one segment in the cache. More complicated schemes which merge segments according to conflict set live ranges are possible and are a focus of future efforts. The new tag is the original tag appended with the segment index to ensure unique tags. Parameters that were varied include thresholds, the number of segments and time intervals for sampling the reference counters.

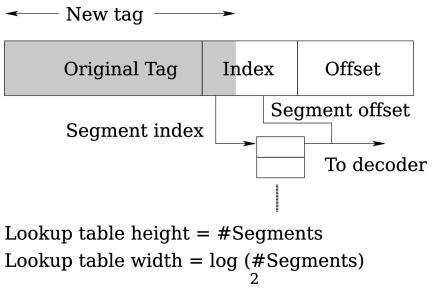

Figure 9. Cache Segment Resizing

4 Evaluation and Results

4.1 Simulation Setup

Our simulation infrastructure is the same as described in Section 2. We performed our studies on a subset of applications from the SPEC2000 and Olden benchmark suites. (Some were not included due to compilation or simulation issues). Cache configurations similar to those in existing Intel Pentium 4 architectures were studied, with 256KB L2 caches and a 32KB 64-byte, 2-way L1. If larger cache sizes were chosen, the efficiencies of caches that were sized will be further skewed upwards as many applications will fit in the cache enabling customized placement to turn off a larger number of cache sets without affecting performance. Our evaluation metrics included effectiveness, energy and performance efficiency, execution time and energy-delay product (EDP).

The decision models used for folding were discussed in Section 3.3. The models are invoked at a fixed time interval, 512K cycles in the following discussion. The decision models themselves used up less than 1% of this time frame

to perform the cache set remapping necessary for folding. The philosophy is similar to dynamic optimizers which are invoked by traps.

4.2 Results and Analysis

We evaluate the heuristics described earlier and compare it with the original cache decay technique [8], and observe that for the L2 caches, sampling the access counters every 512K cycles yields the best results. The heuristics shown in the plots are: i) *decay* representing the cache decay technique with a sample interval of 512K cycles, ii) *dec-res:4b:2s*, representing the decay resizing heuristic with a 4-bit counter with all folded conflict sets merged to two sets with the other *dec-res* heuristics representing changed parameters, iii) *pow-res* represents the power of two resizing technique with a threshold of 1000 accesses for downsizing, iv) *seg-res:125:8S* represents segment resizing with eight segments and a threshold of 125 accesses per segment per 512K cycles. The thresholds were chosen on the basis that on average access to an L2 cache line occurs once every 80000 cycles.

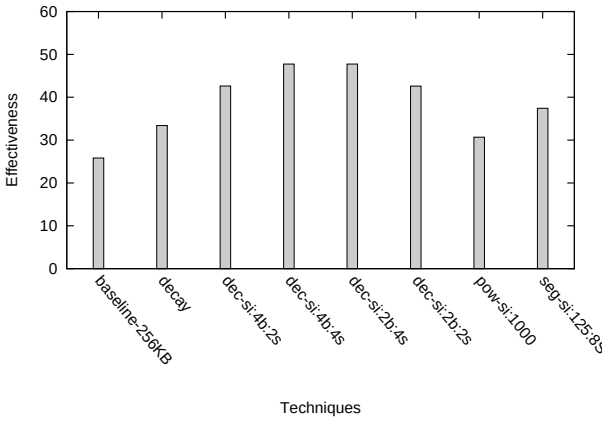

Figure 10. Utilization Comparison

Figures 10 and 11 capture the effectiveness and performance efficiency averaged over all benchmarks. Average energy efficiency is shown in Figure 12. The normalized average execution time (w.r.t. 256KB base cache) is shown in Figure 13 and the maximum increase is less than 4% for the heuristics, compared to a 5.5% increase for the cache decay technique.

The folding heuristics have each memory line mapped to an active cache set. This provides resilience against occasional accesses to a conflict set that was folded, because the active cache set can satisfy the request. This feature allows more conflict sets to be folded and more aggressive turn offs to be scheduled.

Performance efficiency improves indicating that the heuristics folded conflict sets with large inactivity periods

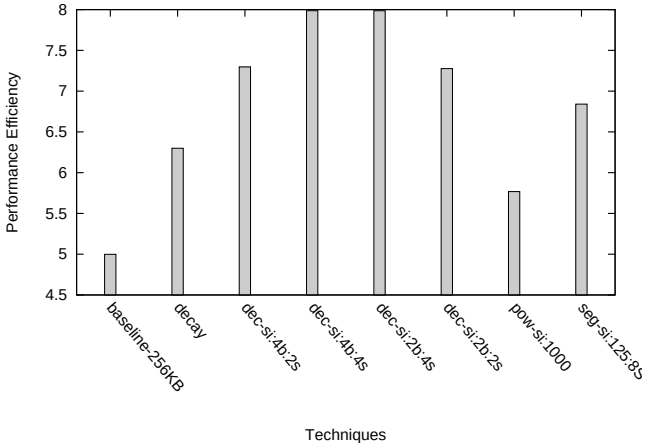

Figure 11. Performance Efficiency Comparison

and therefore cache misses were kept low. Energy efficiency increases by about 20% relative to the base line. If half the cache was turned off to halve leakage energy was halved, the number of hits must stay constant for energy efficiency to double. Every added miss will add to execution time, and to leakage energy increasing the denominator and lowering the numerator in the energy efficiency equation. These effects provide the context for evaluating the efficiency improvement of 20%. This improvement is captured more effectively in the drop in EDP shown in Figure 14 - *the EDP improves by up to 45% compared to the base line cache*.

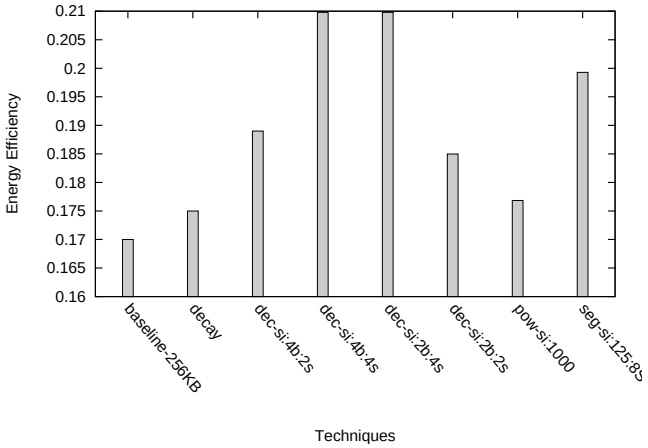

Figure 12. Energy Efficiency Comparison

Among the heuristics studied, the 2-bit hierarchical counters for the decay resizing heuristic provide the largest

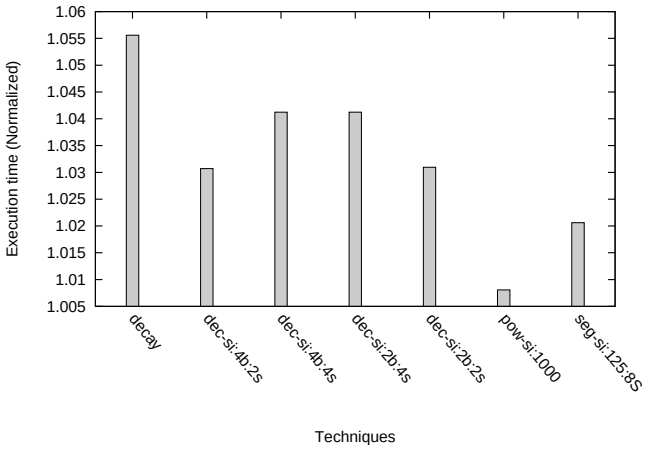

Figure 13. Execution Time Comparison

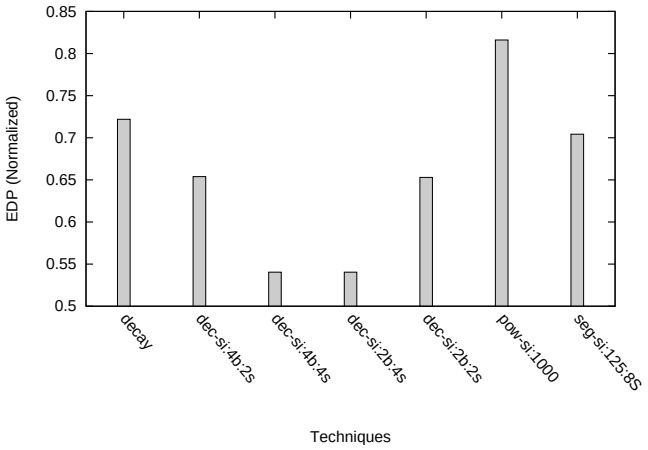

Figure 14. EDP Comparison

improvements in EDP, because of the aggressive turn offs folding all unused conflict sets into just two sets. The segment resizing technique, reduces the EDP by 30%, but affects the execution time by less than 1%. We varied the thresholds for the techniques (for example, from 1000 to 256 accesses per bank with 8 banks) and increased the number of banks to 16, however, this was not found to have a significant effect, affecting results by less than 10%. EDP improvements for individual benchmarks were significant, ranging from 40–60%, and had low variance from the average. The benchmark $bzip2$ however, did not have any appreciable drop in EDP–accesses and conflicts among the three major data structures prevented the technique from folding cache sets to improve energy efficiencies.

The power of two sizing heuristic lowered EDP by 20%, and performed worse than the original cache decay approach and the other heuristics. The reason for the underperformance is that it tries to create different "conventional caches", by resizing conflict sets uniformly, and therefore the sources of inefficiency remained.

The folding heuristics were found to be stable independent of the exact thresholds, counters or banks chosen, indicating that the techniques are robust to the variance in memory reference behaviors, as long as the parameters chosen were in the expected reference range. The IPC degradation followed the execution time pattern, and was limited to less than 4% for all the heuristics.

We also evaluated the hardware overhead our proposed heuristics require. For example, the decay resizing heuristic requires a lookup table of $256 * 8 = 1024$ bits for a 256 set cache. This 256 byte investment is less than 1% of a 256 KB cache, because no tag circuitry is required. As for energy consumption, if lookup table is accessed on every L1 access to minimize added latency, the switching energy for the lookup plus the leakage energy constitutes a negligible amount ($<< 1\%$.) Similarly, the extended tag that is required because of folding constitutes an insignificant overhead in terms of area and power, once again being less than 1%.

The results indicate a significant headroom for improving cache efficiency remains, notwithstanding the benefits gained by the heuristics. A more focused approach using inputs from and controlled by the compiler exploiting liveness and predictability of memory access patterns may be successful in improving efficiencies further. This is especially suited for scientific computation, where access patterns are often known and predictable.

Compilers may have to schedule accesses to the cache in a burst to minimize leakage energy in future. Conflict set construction with compiler inputs and compiler control is one of the extensions we are exploring.

5 Conclusion and Future Extensions

Modern caches have poor efficiencies and were the target of the work reported in this paper. Our hypothesis is that efficiency can be improved by extending the state of practice that involves intelligent shutdown of cache components to scaling caches by remapping main memory to a reduced size cache. We first proposed a model of cache behavior for quantifying operational aspects of efficiency. This model motivated the proposal of several heuristics based in the concept of folding to improve efficiency. We found that efficiencies and the EDP are increased by up to 20% and 40% using these simple heuristics. Significant headroom for improving efficiency still exists, and our current efforts are targeted towards refining our approach towards that end. Extensions include using compiler generated liveness analysis and folding-aware data layouts.

References

[1] J. Abella and A. González. Heterogeneous way-size cache. In *ICS*, pages 239–248, New York, NY, USA, 2006. ACM Press.

[2] J. Abella, A. González, X. Vera, and M. F. P. O'Boyle. IATAC: a smart predictor to turn-off l2 cache lines. *ACM Trans. Archit. Code Optim.*, 2(1):55–77, 2005.

[3] D. H. Albonesi. Selective cache ways: On-demand cache resource allocation. In *MICRO*, 1999.

[4] D. C. Burger, J. R. Goodman, and A. Kagi. The Declining Effectiveness of Dynamic Caching for General-Purpose Microprocessors. Technical Report UWMADISONCS CS-TR-95-1261, University of Wisconsin, Madison, January 1995.

[5] N. P. J. David Tarjan, Shyamkumar Thoziyoor. CACTI 4.0. Technical Report HPL-2006-86, HP Laboratories, 2006.

[6] DIS Stressmark Suite: Specifications for the Stressmarks of the DIS Benchmark Project v 1.0., 2000.

[7] K. Flautner, N. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy caches: Simple techniques for reducing leakage power. In *ISCA*, 2002.

[8] Z. Hu, M. Martonosi, and S. Kaxiras. Improving cache power efficiency with an asymmetric set-associative cache. In *Workshop on Memory Performance Issues*, 2001.

[9] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: exploiting generational behavior to reduce cache leakage power. In *ISCA*, pages 240–251, 2001.

[10] N. S. Kim, T. M. Austin, D. Blaauw, T. N. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. T. Kandemir, and N. Vijaykrishnan. Leakage current: Moore's law meets static power. *IEEE Computer*, 36(12):68–75, 2003.

[11] L. Li, I. Kadayif, Y.-F. Tsai, N. Vijaykrishnan, M. T. Kandemir, M. J. Irwin, and A. Sivasubramaniam. Managing leakage energy in cache hierarchies. *J. Instruction-Level Parallelism*, 5, 2003.

[12] 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 *ISLPED*, pages 90–95, 2000.

[13] S. Ramaswamy and S. Yalamanchili. Customizable fault tolerant caches for embedded processors. In *ICCD*, 2006.

[14] P. Ranganathan, S. Adve, and N. P. Jouppi. Reconfigurable caches and their application to media processing. In *ISCA '00*, pages 214–224, 2000.

[15] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM TOPLAS*, 17(2):233–263, March 1995.

[16] S. Sair and M. Charney. Memory behavior of the spec2000 benchmark suite. Technical Report RC-21852, IBM Thomas J. Watson Research Center, October 2000.

[17] The Simplescalar toolset version 3.0. Available online at http://www.simplescalar.com.

[18] The SPEC CPU2000 Benchmark suite. Information available at http://www.spec.org.

[19] C. Zhang, F. Vahid, and W. Najjar. A highly configurable cache architecture for embedded systems. In *ISCA*, pages 136–146, 2003.

[20] M. Zhang and K. Asanovi. Fine-grain CAM-tag cache resizing using miss tags. In *ISLPED*, pages 130–135, 2002.

[21] H. Zhou, M. C. Toburen, E. Rotenberg, and T. M. Conte. Adaptive mode control: A static-power-efficient cache design. *ACM TECS*, 2(3):347–372, 2003.