

Beyond Basic Region Caching: Specializing Cache Structures for High Performance and Energy Conservation

Michael J. Geiger¹, Sally A. McKee², and Gary S. Tyson³

¹ Advanced Computer Architecture Lab, The University of Michigan,
Ann Arbor, MI 48109-2122
geigerm@eecs.umich.edu

² Computer Systems Lab, Cornell University
Ithaca, NY 14853-3801
sam@csl.cornell.edu

³ Department of Computer Science, Florida State University
Tallahassee, FL 32306-4530
tyson@cs.fsu.edu

Abstract. Increasingly tight energy design goals require processor architects to rethink the organizational structure of microarchitectural resources. In this paper, we examine a new multilateral cache organization that replaces a conventional data cache with a set of smaller region caches that significantly reduces energy consumption with little performance impact. This is achieved by tailoring the cache resources to the specific reference characteristics of each application.

1 Introduction

Energy conservation continues to grow in importance for everything from high performance supercomputers down to embedded systems. Many of the latter must simultaneously deliver both high performance and low energy consumption. In light of these constraints, architects must rethink system design with respect to general versus specific structures. Consider memory hierarchies: the current norm is to use very general cache structures, splitting memory references only according to instructions versus data. Nonetheless, different kinds of data are used in different ways (i.e., exhibiting different locality characteristics), and even a given set of data may exhibit different usage characteristics during different program phases. On-chip caches can consume over 40% of a chip's overall power [1]: as an alternative to general caching designs, further specialization of memory structures to better match usage characteristics of the data they hold can both improve performance and significantly reduce total energy expended.

One form of such heterogeneous memory structures, *region-based caching* [2][3][4], replaces a single unified data cache with multiple caches optimized for global, stack, and heap references; this approach works well precisely because these types of references exhibit different locality characteristics. Furthermore, many ap-

plications are dominated by data from a particular region, and thus greater specialization of region structures should allow both quantitative (in terms of performance and energy) and qualitative (in terms of security and robustness) improvements in system operation. This approach slightly increases required chip area, but using multiple, smaller, specialized caches that together constitute a given “level” of a traditional cache hierarchy and only routing data to a cache that matches those data’s usage characteristics provides many potential benefits: faster access times, lower energy consumption per access, and the ability to turn off structures that are not required for (parts of) a given application.

Given the promise of this general approach, in this paper we first look at the heap cache, the region-based memory structure that most data populate for most applications (in fact, the majority of a unified L1 cache is generally populated by heap data). Furthermore, the heap has always represented the most difficult region of memory to manage well in a cache structure. We propose a simple modification to demonstrate the benefits of further specialization: large and small heap caches. If the application exhibits a small heap footprint, we save energy by using the smaller structure and turn off the larger. For applications with larger footprints, we use both structures, but save energy by keeping highly used “hot” data in the smaller, faster, lower-energy cache. The compiler determines (either through profiled feedback or heuristics) which data belong in which cache, and it conveys this information to the architecture via two different `malloc()` functions that allocate data structures in two disparate regions of memory. This allows the microarchitecture to determine what data are to be cached where without the need for additional bits in memory reference instructions and without complex coherence mechanisms.

The architectural approach we describe above addresses one kind of energy consumption—dynamic or switching energy—via smaller caches that reduce the cost of each data access. The second kind of energy consumption that power-efficient caching structures must address is static or leakage energy; for this, we add *drowsy caching* [5][6][7], an architectural technique exploiting dynamic voltage scaling. Reducing supply voltage to inactive lines lowers their static power dissipation. When a drowsy line is accessed, the supply voltage must be returned to its original value before the data may be accessed. Drowsy caches save less power than many other leakage reduction techniques, but do not suffer the dramatically increased latencies of other methods.

Using the MiBench suite [8], we study application data usage properties and the design space for split heap caches. The contributions of this paper are:

- We perform a detailed analysis of heap data characteristics to determine the best heap caching strategy and necessary cache size for each application.
- We show that a significant number of embedded applications do not require a large heap cache and demonstrate significant energy savings with a minimal performance loss for those applications by using a smaller cache. We show energy savings of up to 79% using non-drowsy reduced heap caches and up to 84% using drowsy reduced heap caches.
- For applications that do have a large heap footprint and require a bigger cache, we demonstrate that we can still achieve significant energy savings by identifying a subset of data responsible for the majority of accesses to the heap region and split-

ting the heap cache into two structures, a small cache for hot data and a large cache for the remaining data. We show energy savings of up to 45% using non-drowsy split-heap caches and up to 78% using drowsy split heap caches.

The remainder of this paper is organized as follows. In Section 2, we present related work on energy-saving techniques in caches, focusing on drowsy and region-based caching and the combination of the two to reduce both dynamic and static energy. Section 3 discusses the caching requirements of heap data, focusing on the footprint size and locality of the heap region. Section 4 describes our experimental setup and presents our results. Section 5 offers conclusions and directions for future work.

2 Related Work

Since dissipated power per access is proportional to cache size, partitioning techniques reduce power by accessing smaller structures. Cache partitioning schemes may be vertical or horizontal. Vertical partitioning adds a level between the L1 and the processor; examples include line buffers [9][10] and filter caches [11]. These structures provide low-power accesses for data with temporal locality, but typically incur many misses and therefore increase average observed L1 latency. Horizontal partitioning divides entities such as cache lines into smaller segments, as in cache sub-banking [9][10]. Memory references are routed to the proper segment, reducing dynamic power per data access.

Fig. 1 illustrates a simple example of region-based caching [3][4], a horizontal partitioning scheme that replaces a unified data cache with heterogeneous caches optimized for global, stack, and heap references. Any non-global, non-stack reference is directed to a normal L1 cache, but since most non-global, non-stack references are to heap data, (with a small number of accesses to text and read-only regions), this L1 is

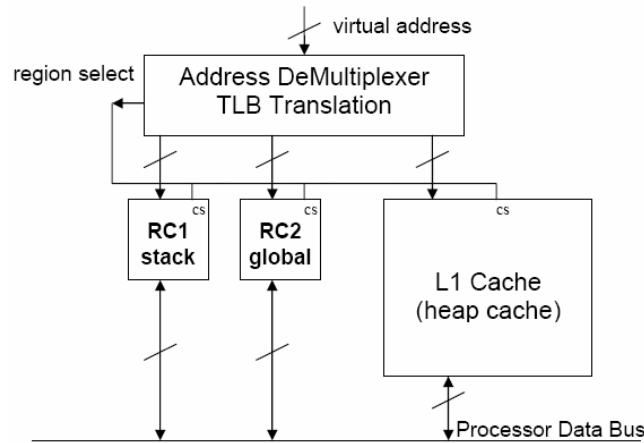


Fig. 1. Memory design for region-based caching (from Lee [4])

referred to as the *heap cache*. On a memory reference, only the appropriate region cache is activated and draws power. Relatively small working sets for stack and global regions allow their caches to be small, dissipating even less power on hits. Splitting references among caches eliminates inter-region conflicts, thus each cache may implement lower associativity, reducing complexity and access time.

The downside to region-based caching is that increased cache capacity leads to higher static energy dissipation. Drowsy region-based caching [2] attacks this problem by implementing drowsy caching [5][6][7], a leakage energy reduction technique, within the region-based caches. In a drowsy cache, inactive lines use a reduced supply voltage; an access to a drowsy line must wait for the supply voltage to return to its full value. After a given interval, lines may switch state from active to drowsy; depending on the policy, lines accessed within the interval may remain active. Geiger et al [2] show that the combination of drowsy and region-based caching yields more benefits than either alone because each technique improves the performance of the other. Drowsy caching all but eliminates the static power increase due to additional region caches, while the partitioning strategy used in region-based caching allows for more aggressive drowsy policies. The drowsy interval of each region cache can be tuned according to the reference characteristics of that region, allowing highly active regions to be less drowsy than inactive regions.

Geiger et al [2] first discussed the idea of split heap caches. They noted that the ideal drowsy interval for the heap cache was the same as that of the stack cache, a result that implied at least some of the heap data had locality similar to that of stack data. The authors observed that moving the highly local data to a separate structure would allow more aggressive drowsy caching of the low-locality data, further reducing the static energy consumption of region-based caches. Using a 4KB cache for hot heap data and maintaining a 32KB cache for low-locality heap data, they showed an average energy reduction of 71.7% over a unified drowsy L1 data cache.

3 Characteristics of heap data

In this section, we analyze the characteristics of heap cache accesses in applications from the MiBench [8] benchmark suite to determine the best heap caching strategy for each program. We begin by assessing the significance of the heap region within our target applications, looking at its overall size and number of accesses relative to the other semantic regions. This information is provided in Table 1. The second and third columns of the table show the number of unique block addresses accessed in the heap cache and the number of accesses to those addresses, respectively. Since our simulations assume 32B cache blocks, 1 KB of data contains 32 unique block addresses. The fourth and fifth columns show this same data as a percentage of the corresponding values for all regions (i.e., the fourth column shows the ratio of unique data addresses in the heap region to all unique data addresses in the application). We can see several cases that bear out the previous assertions about heap data: they have a large footprint and low locality. In these applications, the heap cache accesses occupy a much larger percentage of the overall footprint than of the total accesses. The most extreme cases are applications such as *FFT.inverse* and *patricia* in which heap accesses account for

over 99% of the unique addresses accessed throughout the programs but comprise less than 7% of the total data accesses. This relationship holds in most applications; heap accesses cover an average of 65.67% of the unique block addresses and account for 29.81% of the total data accesses. In some cases, we see a correlation between footprint size and number of accesses—applications with few heap lines and few accesses, like *pgp.encode*, and applications with a large percentage of both cache lines and accesses, like *tiff2rgba*. A few outliers buck the trend entirely, containing frequently accessed heap data with a relatively small footprint; *dijkstra* is one example.

We see that about half of the applications have a fairly small number of lines in the heap, with 16 of the 34 applications containing fewer than 1000 unique addresses. The *adpcm* application has the smallest footprint, using 69 and 68 unique addresses in the encode and decode phases, respectively. The typical 32 KB L1 heap cache is likely far larger than these applications need; if we use a smaller heap cache, we can dissipate less dynamic power per access with a minimal effect on performance. Since heap cache accesses still comprise a significant percentage of the overall data accesses, this change should have a noticeable effect on the dynamic energy consumption of these benchmarks. Shrinking the heap cache will also reduce its static energy

Table 1. Characteristics of heap cache accesses in MiBench applications

Benchmark	# unique addresses	Accesses to heap cache	% total unique addresses	% total accesses
adpcm.encode	69	39971743	27.60%	39.88%
adpcm.decode	68	39971781	26.98%	39.88%
basicmath	252	49181748	61.17%	4.52%
blowfish.decode	213	39190633	39.01%	10.17%
blowfish.encode	212	39190621	38.90%	10.17%
bitcount	112	12377683	42.75%	6.75%
jpeg.encode	26012	10214537	99.16%	29.35%
CRC32	90	159955061	41.10%	16.69%
dijkstra	347	44917851	19.74%	38.31%
jpeg.decode	1510	7036942	90.20%	62.91%
FFT	16629	15262360	99.16%	8.56%
FFT.inverse	16630	14013100	99.17%	6.29%
ghostscript	59594	56805375	97.97%	15.29%
ispell	13286	28000346	96.45%	6.43%
mad	2123	40545761	82.25%	36.41%
patricia	110010	16900929	99.86%	6.59%
pgp.encode	298	252620	7.44%	1.93%
pgp.decode	738	425414	44.92%	1.50%
quicksort	62770	152206224	66.67%	12.89%
rijndael.decode	229	37374614	30.99%	21.70%
rijndael.encode	236	35791440	40.00%	19.62%
rsynth	143825	104084186	99.23%	21.43%
stringsearch	203	90920	18.17%	6.21%
sha	90	263617	20.93%	0.72%
susan.corners	18479	9614163	97.07%	63.56%
susan.edges	21028	22090676	99.12%	62.28%
susan.smoothing	7507	179696772	97.03%	41.72%
tiff2bw	2259	57427236	92.09%	98.50%
tiffdither	1602	162086279	83.09%	62.82%
tiffmedian	4867	165489090	53.03%	79.82%
tiff2rgba	1191987	81257094	99.99%	98.51%
gsm.encode	302	157036702	68.02%	11.66%
typeset	168075	153470300	97.97%	49.00%
gsm.decode	285	78866326	55.56%	21.50%
AVERAGE			65.67%	29.81%

consumption. Previous resizable caches disable unused ways [12][13] or sets [13][14] in set-associative caches; we can use similar logic to simply disable the entire large heap cache and route all accesses to the small cache when appropriate. In Section 4, we show the effects of this modification on energy and performance.

Shrinking the heap cache may reduce the energy consumption of the remaining benchmarks, but the resulting performance loss may be too great to tolerate for applications with a large heap footprint. However, we can still gain some benefit by identifying a small subset of addresses with good locality and routing their accesses to a smaller structure. Because we want the majority of references to dissipate less power, we should choose the most frequently-accessed lines. The access count gives some sense of the degree of temporal locality for a given address.

Usually, a small number of blocks are responsible for the majority of the heap accesses, as shown in Table 2. The table gives the number of lines needed to cover different percentages—50%, 75%, 90%, 95%, and 99%—of the total accesses to the heap cache. We can see that, on average, just 2.14% of the cache lines cover 50% of the accesses. Although the rate of coverage decreases somewhat as you add more

Table 2. Number of unique addresses required to cover different fractions of accesses to the heap cache in MiBench applications

Benchmark	# unique addresses	% unique addresses needed to cover given percentage of heap cache accesses				
		50%	75%	90%	95%	99%
adpcm.encode	69	1.45%	2.90%	2.90%	2.90%	2.90%
adpcm.decode	68	1.47%	1.47%	1.47%	1.47%	1.47%
basicmath	252	3.97%	25.40%	48.02%	55.56%	61.90%
blowfish.decode	213	0.94%	1.41%	2.35%	26.76%	55.87%
blowfish.encode	212	0.94%	1.42%	2.36%	26.89%	56.13%
bitcount	112	0.89%	1.79%	2.68%	3.57%	3.57%
jpeg.encode	26012	0.10%	0.65%	2.91%	38.19%	87.28%
CRC32	90	2.22%	3.33%	4.44%	4.44%	4.44%
dijkstra	347	0.29%	18.16%	39.19%	49.57%	63.11%
jpeg.decode	1510	4.77%	12.32%	31.85%	44.11%	59.47%
FFT	16629	0.05%	0.14%	4.82%	40.67%	85.33%
FFT.inverse	16630	0.05%	0.14%	13.02%	44.00%	86.51%
ghostscript	59594	0.01%	0.04%	0.56%	6.64%	57.49%
ispell	13286	0.09%	0.23%	0.46%	0.68%	1.29%
mad	2123	1.32%	2.64%	9.70%	14.88%	24.54%
patricia	110010	0.02%	0.06%	0.32%	36.64%	86.03%
pgp.encode	298	0.67%	1.01%	3.69%	6.71%	26.85%
pgp.decode	738	0.27%	0.41%	1.08%	2.30%	29.67%
quicksort	62770	0.02%	0.04%	0.15%	22.08%	49.13%
rijndael.decode	229	1.31%	2.18%	6.55%	31.44%	57.21%
rijndael.encode	236	1.27%	2.97%	7.63%	32.63%	56.78%
rsynth	143825	0.00%	0.00%	0.01%	1.28%	77.33%
stringsearch	203	17.24%	42.86%	59.61%	65.52%	72.91%
sha	90	1.11%	2.22%	3.33%	3.33%	8.89%
susan.corners	18479	0.03%	3.02%	11.04%	14.87%	32.66%
susan.edges	21028	0.02%	4.92%	15.13%	20.22%	30.42%
susan.smoothing	7507	0.01%	0.09%	13.72%	30.25%	44.11%
tiff2bw	2259	10.27%	15.41%	24.26%	29.39%	37.05%
tiffdither	1602	9.43%	19.60%	25.72%	29.59%	40.76%
tiffmedian	4867	4.03%	10.89%	16.72%	20.81%	47.83%
tiff2rgba	1191987	0.04%	0.11%	57.39%	78.69%	95.73%
gsm.encode	302	2.32%	3.97%	5.96%	7.62%	10.60%
typeset	168075	5.55%	15.41%	25.53%	33.02%	60.12%
gsm.decode	285	0.70%	1.40%	4.21%	5.96%	30.53%
AVERAGE (all apps)		2.14%	5.84%	13.20%	24.49%	45.47%
AVERAGE (>1k unique addr)		1.99%	4.76%	14.07%	28.11%	55.73%

blocks—in other words, the first N blocks account for more accesses than the next N blocks—we still only need 5.84% to cover 75% of the accesses, 13.2% to cover 90% of the accesses, 24.49% to cover 95% of the accesses, and 45.47% to cover 99% of the accesses. The percentages do not tell the whole story, as the footprint sizes are wildly disparate for these applications. However, the table also shows that in applications with large footprints (defined as footprints of 1000 unique addresses or more), the percentage of addresses is lower for the first two coverage points (50% and 75%). This statistic implies that we can identify a relatively small subset of frequently accessed lines for all applications, regardless of overall footprint size.

Since a small number of addresses account for a significant portion of the heap cache accesses, we can route these frequently accessed data to a smaller structure to reduce the energy consumption of the L1 data cache. Our goal is to maximize the low-power accesses without a large performance penalty, so we need to judiciously choose which data to place in the hot heap cache. To estimate performance impact, we use the Cheetah cache simulator [15] to find a lower bound on the miss rate for a given number of input data lines. We simulate fully-associative 2 KB, 4 KB, and 8 KB caches with optimal replacement [16] and route the N most frequently accessed lines to the cache, varying N by powers of 2. We recognize that the true miss rate will be higher since the hot heap cache is actually direct-mapped; optimal replacement minimizes conflict misses and gives a sense of when the cache is filled to capacity.

Tables 3, 4, and 5 show the results of these simulations for 2 KB, 4 KB, and 8 KB caches, respectively. We present only a subset of the applications, omitting programs with small heap footprints and a worst-case miss rate less than 1% because they will perform well at any cache size. These tables show a couple of clear trends. The first is that the miss rate rises precipitously for small values of N , but levels off around $N = 512$ or 1024 in most cases. This result reflects the fact that the majority of accesses are concentrated at a small number of addresses. The second is that the miss rates

Table 3. Miss rates for a fully-associative 2 KB cache using optimal replacement for different numbers of input addresses. Applications shown either have a large heap footprint or a worst-case miss rate above 1%

Benchmark	Miss rate for given N value						
	128	256	512	1024	2048	4096	8192
jpeg.encode	0.2%	0.8%	1.8%	2.5%	2.5%	2.5%	2.5%
dijkstra	4.2%	8.0%	8.0%	8.0%	8.0%	8.0%	8.0%
jpeg.decode	0.4%	0.9%	1.7%	2.8%	2.8%	2.8%	2.8%
FFT	0.1%	0.1%	0.2%	0.3%	0.4%	0.8%	1.4%
FFT.inverse	0.1%	0.1%	0.2%	0.3%	0.5%	0.8%	1.5%
ghostscript	0.0%	0.2%	0.3%	0.5%	0.6%	0.6%	0.8%
ispell	0.2%	0.4%	0.4%	0.4%	0.4%	0.4%	0.4%
mad	0.7%	1.6%	2.4%	2.4%	2.4%	2.4%	2.4%
patricia	0.7%	1.3%	1.8%	1.9%	2.0%	2.0%	2.1%
quicksort	0.0%	0.0%	0.1%	0.1%	0.1%	0.2%	0.2%
rsynth	0.0%	0.0%	0.0%	0.0%	0.1%	0.1%	0.1%
stringsearch	1.8%	2.0%	2.0%	2.0%	2.0%	2.0%	2.0%
susan.corners	0.0%	0.0%	0.0%	0.0%	0.1%	0.1%	0.2%
susan.edges	0.0%	0.0%	0.0%	0.0%	0.1%	0.1%	0.2%
susan.smoothing	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
tiff2bw	2.5%	3.8%	4.7%	5.7%	5.7%	5.7%	5.7%
tiffdither	0.4%	0.8%	1.3%	1.6%	1.6%	1.6%	1.6%
tiffmedian	0.5%	1.2%	2.0%	3.4%	3.5%	3.4%	3.4%
tiff2rgba	2.5%	3.8%	4.6%	6.1%	7.1%	7.1%	7.1%
typeset	1.4%	2.6%	2.7%	3.0%	3.4%	4.0%	5.0%

Table 4. Miss rates for a fully-associative 4 KB cache using optimal replacement for different numbers of input addresses. Applications are the same set shown in Table 3

Benchmark	Miss rate for given N value						
	128	256	512	1024	2048	4096	8192
jpeg.encode	0.0%	0.3%	0.9%	1.4%	1.5%	1.5%	1.5%
dijkstra	0.0%	2.7%	2.7%	2.7%	2.7%	2.7%	2.7%
jpeg.decode	0.0%	0.3%	0.7%	1.4%	1.5%	1.5%	1.5%
FFT	0.0%	0.0%	0.1%	0.1%	0.3%	0.6%	1.3%
FFT.inverse	0.0%	0.0%	0.1%	0.2%	0.4%	0.7%	1.4%
ghostscript	0.0%	0.0%	0.0%	0.1%	0.2%	0.3%	0.4%
ispell	0.0%	0.1%	0.1%	0.1%	0.1%	0.1%	0.1%
mad	0.0%	0.8%	1.6%	1.6%	1.6%	1.6%	1.6%
patricia	0.0%	0.3%	0.5%	0.6%	0.6%	0.6%	0.7%
quicksort	0.0%	0.0%	0.0%	0.1%	0.1%	0.1%	0.2%
rsynth	0.0%	0.0%	0.0%	0.0%	0.0%	0.1%	0.1%
stringsearch	0.2%	0.5%	0.5%	0.5%	0.5%	0.5%	0.5%
susan.corners	0.0%	0.0%	0.0%	0.0%	0.0%	0.1%	0.1%
susan.edges	0.0%	0.0%	0.0%	0.0%	0.0%	0.1%	0.1%
susan.smoothing	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
tiff2bw	0.0%	2.5%	3.9%	5.0%	5.0%	5.0%	5.0%
tiffdither	0.0%	0.5%	1.1%	1.3%	1.3%	1.3%	1.3%
tiffmedian	0.0%	0.8%	1.3%	2.9%	3.0%	3.0%	3.0%
tiff2rgba	0.0%	2.5%	3.1%	4.6%	5.8%	5.8%	5.8%
typeset	0.0%	0.1%	0.2%	0.5%	0.9%	1.4%	2.3%

Table 5. Miss rates for a fully-associative 8 KB cache using optimal replacement for different numbers of input addresses. Applications shown are the same set shown in Table 3

Benchmark	Miss rate for given N value						
	128	256	512	1024	2048	4096	8192
jpeg.encode	0.0%	0.0%	0.2%	0.6%	0.6%	0.7%	0.7%
dijkstra	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
jpeg.decode	0.0%	0.0%	0.2%	0.7%	0.7%	0.7%	0.7%
FFT	0.0%	0.0%	0.0%	0.1%	0.3%	0.6%	1.2%
FFT.inverse	0.0%	0.0%	0.0%	0.1%	0.3%	0.7%	1.4%
ghostscript	0.0%	0.0%	0.0%	0.0%	0.1%	0.1%	0.2%
ispell	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
mad	0.0%	0.0%	0.8%	0.9%	0.9%	0.9%	0.9%
patricia	0.0%	0.0%	0.1%	0.2%	0.3%	0.3%	0.3%
quicksort	0.0%	0.0%	0.0%	0.0%	0.1%	0.1%	0.1%
rsynth	0.0%	0.0%	0.0%	0.0%	0.0%	0.1%	0.1%
stringsearch	0.2%	0.2%	0.2%	0.2%	0.2%	0.2%	0.2%
susan.corners	0.0%	0.0%	0.0%	0.0%	0.0%	0.1%	0.1%
susan.edges	0.0%	0.0%	0.0%	0.0%	0.0%	0.1%	0.1%
susan.smoothing	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
tiff2bw	0.0%	0.0%	2.4%	3.6%	3.7%	3.7%	3.7%
tiffdither	0.0%	0.0%	0.6%	0.8%	0.8%	0.8%	0.8%
tiffmedian	0.0%	0.0%	0.2%	1.9%	2.0%	2.0%	2.0%
tiff2rgba	0.0%	0.0%	0.5%	1.7%	3.2%	3.3%	3.3%
typeset	0.0%	0.0%	0.0%	0.0%	0.2%	0.6%	1.3%

remain tolerable for all applications for N values up to 256, regardless of cache size. In order to gain the maximum benefit from split heap caching, we would like to route as many accesses as possible to a small cache. These simulations indicate that varying the cache size will not have a dramatic effect on performance, so we choose the smallest cache size studied—2 KB—and route the 256 most accessed lines to that cache when splitting the heap. This approach should give us a significant energy reduction without compromising performance.

This approach for determining what data is routed to the small cache does require some refinement. In practice, the compiler would use a profiling run of the application

to determine the appropriate caching strategy, applying a well-defined heuristic to the profiling data. We use a simple heuristic in this work to show the potential effectiveness of our caching strategies; a more refined method would likely yield better results.

4 Experiments

The previous section motivates the need for two separate heap caches, one large and one small, to accommodate the needs of all applications. As shown in Table 1, many applications have small heap footprints and therefore do not require a large heap cache; in these cases, we can disable the large cache and place all heap data in the smaller structure. This approach will reduce dynamic energy by routing accesses to a smaller structure and reduce static energy by decreasing the active cache area. Applications with large heap footprints are more likely to require both caches to maintain performance. We showed in Table 2 that most heap references access a small subset of the data; by keeping this hot data in the smaller structure, we can save dynamic energy. In all cases, we can further lower static energy consumption by making the caches drowsy.

Our simulations use a modified version of the SimpleScalar ARM target [17]. We use Watch [18] for dynamic power modeling and Zhang et al.’s HotLeakage [19] for static power modeling. HotLeakage contains a detailed drowsy cache model, which was used in [20] to compare state-preserving and non-state-preserving techniques for leakage control. HotLeakage tracks the number of lines in both active and drowsy modes and calculates leakage power appropriately. It also models the power of the additional hardware required to support drowsy caching. All simulations use an in-order processor model similar to the Intel StrongARM SA-110 [1].

Table 6 shows simulation results for region-based caches using three different heap cache configurations: a large (32KB) unified heap cache, a small (2KB) unified heap cache, and a split heap cache using both the large and small caches. We present normalized energy and performance numbers, using a single 32KB direct-mapped L1 data cache as the baseline. Because all region-based caches are direct-mapped to minimize energy consumption, we use a direct-mapped baseline to ensure a fair comparison. The shaded cells indicate the best heap caching method for each application. To choose the most effective configuration, we calculate the energy-delay product ratio [21] for each configuration, using the cache organization that yields the lowest value.

As expected, using the small heap cache and disabling the large offers the best energy savings across the board. Most applications consume over 70% less energy in this case; the only exceptions are the *susan.corners* and *susan.edges* applications, which suffer the worst performance losses of any applications under this configuration. 18 of the 34 applications in the MiBench suite experience performance losses of less than 1%, including *ghostscript*, *mad*, *patricia*, *rsynth*, and *susan.smoothing*—all applications with large heap footprints. This result suggests that heap data in these applications have good locality characteristics and are frequently accessed while present in the cache. Another application, *quicksort*, suffers significant performance losses for all configurations due to an increased number of stack misses, and therefore still

benefits most from using the small heap cache. In all of these cases, we gain substantial energy savings with virtually no performance loss, reducing overall energy consumption by up to 79%. Several applications actually experience small speedups, as indicated by the negative values for performance loss. These speedups result from reduced conflict between regions.

For those applications that suffer substantial performance losses with the small cache alone, the split heap cache offers a higher-performance alternative that still saves some energy. The most dramatic improvements can be seen in *susan.corners* and *susan.edges*. With the large heap cache disabled, these two applications see their runtime more than double; with a split heap cache, they experience small speedups. Other applications, such as *FFT* and *tiff2rgba*, run over 30% slower with the small cache alone and appear to be candidates for a split heap cache. However, the energy required to keep the large cache active overwhelms the performance benefit of splitting the heap, leading to a higher energy-delay product.

Table 6. Energy and performance results for various non-drowsy heap caching configurations: a large unified heap cache, a small unified heap cache, and a split heap cache employing both large and small caches. The baseline is a 32KB direct-mapped unified L1 data cache. Shaded cells indicate the most effective heap caching method for each application

Benchmark	32KB heap cache		2KB heap cache		Split heap cache (2KB hot/32KB cold)	
	Normalized total energy	Execution time increase	Normalized total energy	Execution time increase	Normalized total energy	Execution time increase
adpcm.encode	0.92	0.40%	0.25	0.40%	0.80	0.40%
adpcm.decode	0.89	0.26%	0.25	0.26%	0.75	0.26%
basicmath	0.64	0.90%	0.25	1.08%	0.64	1.08%
blowfish.decode	0.62	0.39%	0.24	0.40%	0.58	0.40%
blowfish.encode	0.62	0.40%	0.24	0.41%	0.58	0.41%
bitcount	0.75	0.00%	0.26	0.00%	0.75	0.00%
jpeg.encode	0.77	-0.13%	0.25	4.89%	0.70	3.66%
CRC32	0.65	-0.38%	0.24	-0.38%	0.57	-0.38%
dijkstra	0.78	0.58%	0.24	6.84%	0.62	6.84%
jpeg.decode	0.88	-0.12%	0.23	11.70%	0.65	9.99%
FFT	0.72	7.32%	0.29	32.26%	0.71	7.64%
FFT.inverse	0.69	5.11%	0.28	22.45%	0.69	5.33%
ghostscript	0.65	-0.21%	0.24	0.11%	0.60	-0.11%
ispell	0.62	-0.03%	0.25	2.83%	0.61	0.36%
mad	0.77	-0.26%	0.23	0.28%	0.61	0.10%
patricia	0.68	0.20%	0.25	0.63%	0.68	0.57%
pgp.encode	0.63	0.04%	0.25	0.04%	0.64	0.04%
pgp.decode	0.62	0.00%	0.25	0.02%	0.64	0.00%
quicksort	0.93	22.47%	0.29	21.42%	0.92	22.62%
rijndael.decode	0.66	0.94%	0.24	1.29%	0.55	1.29%
rijndael.encode	0.64	0.49%	0.24	0.84%	0.55	0.84%
rsynth	0.66	0.73%	0.24	0.97%	0.56	0.92%
stringsearch	0.68	-0.09%	0.25	0.03%	0.68	0.04%
sha	0.66	0.09%	0.26	0.09%	0.69	0.09%
susan.corners	0.90	-2.21%	0.50	250.85%	0.73	-2.14%
susan.edges	0.90	-0.98%	0.35	115.70%	0.76	-0.89%
susan.smoothing	0.78	-0.04%	0.23	0.67%	0.63	0.00%
tiff2bw	1.09	-0.04%	0.21	2.11%	0.83	0.86%
tiffdither	0.93	-0.19%	0.24	7.19%	0.75	1.51%
tiffmedian	1.00	2.88%	0.22	4.54%	0.79	3.17%
tiff2rgba	1.08	-0.60%	0.24	32.04%	0.89	-0.33%
gsm.encode	0.62	0.00%	0.24	0.03%	0.58	0.03%
typeset	0.80	-0.16%	0.23	4.78%	0.75	0.56%
gsm.decode	0.76	0.01%	0.25	0.04%	0.68	0.04%

Table 7 shows simulation results for drowsy heap caching configurations. In all cases, we use the ideal drowsy intervals derived in [2]—for the unified heap caches, 512 cycles; for the split heap cache, 512 cycles for the hot heap cache and 1 cycle for the cold heap cache. The stack and global caches use 512 and 256 cycle windows, respectively. Note that drowsy caching alone offers a 35% energy reduction over a non-drowsy unified cache for this set of benchmarks [2].

Although all caches benefit from the static power reduction offered by drowsy caching, this technique has the most profound effect on the split heap caches. Drowsy caching all but eliminates the leakage energy of the large heap cache, as it contains rarely accessed data with low locality and is therefore usually inactive. Since the small cache experiences fewer conflicts in the split heap scheme than by itself, its lines are also less active and therefore more conducive to drowsy caching. Both techniques are very effective at reducing the energy consumption of these benchmarks.

Table 7. Energy and performance results for various drowsy heap caching configurations: a large unified heap cache, a small unified heap cache, and a split heap cache employing both large and small caches. The baseline is a 32KB direct-mapped unified L1 data cache with a 512 cycle drowsy interval. Shaded cells indicate the most effective heap caching method for each application

Benchmark	32KB heap cache		2KB heap cache		Split heap cache (2KB hot/32KB cold)	
	Normalized total energy	Execution time increase	Normalized total energy	Execution time increase	Normalized total energy	Execution time increase
adpcm.encode	0.58	0.79%	0.21	0.79%	0.26	0.45%
adpcm.decode	0.57	0.65%	0.21	0.65%	0.25	0.48%
basicmath	0.29	1.01%	0.23	1.20%	0.26	1.11%
blowfish.decode	0.33	0.44%	0.23	0.45%	0.24	0.20%
blowfish.encode	0.33	0.47%	0.23	0.48%	0.24	0.23%
bitcount	0.33	0.00%	0.23	0.00%	0.27	-0.04%
jpeg.encode	0.48	-0.04%	0.21	4.83%	0.29	3.59%
CRC32	0.38	-0.36%	0.22	-0.36%	0.24	-0.73%
dijkstra	0.55	0.82%	0.21	6.91%	0.24	5.99%
jpeg.decode	0.73	-0.10%	0.19	11.70%	0.31	9.76%
FFT	0.33	7.27%	0.23	31.99%	0.27	7.35%
FFT.inverse	0.31	5.08%	0.23	22.28%	0.27	5.11%
ghostscript	0.37	-0.11%	0.22	0.22%	0.26	-0.40%
ispell	0.31	0.01%	0.23	2.84%	0.25	0.19%
mad	0.53	-0.11%	0.21	0.47%	0.26	0.09%
patricia	0.32	0.23%	0.23	0.78%	0.27	0.62%
pgp.encode	0.27	0.04%	0.23	0.04%	0.26	0.00%
pgp.decode	0.27	0.00%	0.23	0.02%	0.26	-0.03%
quicksort	0.39	22.43%	0.23	21.38%	0.29	22.48%
rijndael.decode	0.42	1.52%	0.22	1.99%	0.24	1.81%
rijndael.encode	0.40	0.98%	0.22	1.50%	0.24	1.29%
rsynth	0.41	0.87%	0.22	1.13%	0.25	0.65%
stringsearch	0.31	0.10%	0.23	0.22%	0.26	0.20%
sha	0.27	0.14%	0.23	0.14%	0.26	0.14%
susan.corners	0.73	-2.21%	0.22	250.32%	0.45	-1.96%
susan.edges	0.73	-0.98%	0.20	115.61%	0.48	-0.67%
susan.smoothing	0.57	-0.04%	0.20	0.76%	0.33	-0.39%
tiff2bw	1.00	-0.05%	0.16	2.78%	0.55	1.15%
tiffdither	0.73	0.11%	0.19	7.44%	0.39	2.43%
tiffmedian	0.86	2.91%	0.17	4.99%	0.50	3.36%
tiff2rgba	1.00	-0.60%	0.16	32.04%	0.68	0.05%
gsm.encode	0.34	0.01%	0.22	0.05%	0.24	-0.14%
typeset	0.62	-0.12%	0.20	4.99%	0.52	2.16%
gsm.decode	0.43	0.01%	0.22	0.05%	0.25	-0.17%

Drowsy split heap caches save up to 76% of the total energy, while the small caches alone save between 77% and 84%. Because drowsy caching has a minimal performance cost, the runtime numbers are similar to those shown in the previous table. The small cache alone and the split heap cache produce comparable energy-delay values for several applications; *ispell* is one example. In these cases, performance-conscious users can employ a split heap cache, while users desiring lower energy consumption can choose the small unified heap cache.

Shrinking the large heap cache further alleviates its effect on energy consumption. The data remaining in that cache is infrequently accessed and can therefore tolerate an increased number of conflicts. Table 8 shows simulation results for two different split heap configurations—one using a 32KB cache for cold heap data, the other using an 8KB cache—as well as the 2KB unified heap cache. All caches are drowsy. The unified cache is still most efficient for the majority of applications, but shrinking the

Table 8. Energy and performance results for different drowsy heap caching configurations: a small heap cache alone, and split heap caches using 32KB and 8KB caches for low-locality heap data. The baseline is a 32KB direct-mapped unified L1 data cache with a 512 cycle drowsy interval. Shaded cells indicate the most effective heap caching method for each application

Benchmark	2KB heap cache		Split heap cache (2KB hot/32KB cold)		Split heap cache (2KB hot/8KB cold)	
	Normalized total energy	Execution time increase	Normalized total energy	Execution time increase	Normalized total energy	Execution time increase
adpcm.encode	0.21	0.79%	0.26	0.45%	0.22	0.45%
adpcm.decode	0.21	0.65%	0.25	0.48%	0.22	0.48%
basicmath	0.23	1.20%	0.26	1.11%	0.24	1.11%
blowfish.decode	0.23	0.45%	0.24	0.20%	0.23	0.20%
blowfish.encode	0.23	0.48%	0.24	0.23%	0.23	0.23%
bitcount	0.23	0.00%	0.27	-0.04%	0.24	-0.04%
jpeg.encode	0.21	4.83%	0.29	3.59%	0.24	3.63%
CRC32	0.22	-0.36%	0.24	-0.73%	0.22	-0.73%
dijkstra	0.21	6.91%	0.24	5.99%	0.22	5.99%
jpeg.decode	0.19	11.70%	0.31	9.76%	0.22	9.99%
FFT	0.23	31.99%	0.27	7.35%	0.25	11.79%
FFT.inverse	0.23	22.28%	0.27	5.11%	0.24	8.20%
ghostscript	0.22	0.22%	0.26	-0.40%	0.24	-0.35%
ispell	0.23	2.84%	0.25	0.19%	0.24	0.68%
mad	0.21	0.47%	0.26	0.09%	0.22	0.10%
patricia	0.23	0.78%	0.27	0.62%	0.24	0.63%
pgp.encode	0.23	0.04%	0.26	0.00%	0.24	0.00%
pgp.decode	0.23	0.02%	0.26	-0.03%	0.24	-0.03%
quicksort	0.23	21.38%	0.29	22.48%	0.25	21.73%
rijndael.decode	0.22	1.99%	0.24	1.81%	0.22	1.81%
rijndael.encode	0.22	1.50%	0.24	1.29%	0.23	1.29%
rsynth	0.22	1.13%	0.25	0.65%	0.22	0.64%
stringsearch	0.23	0.22%	0.26	0.20%	0.24	0.19%
sha	0.23	0.14%	0.26	0.14%	0.24	0.14%
susan.corners	0.22	250.32%	0.45	-1.96%	0.27	48.34%
susan.edges	0.20	115.61%	0.48	-0.67%	0.27	22.48%
susan.smoothing	0.20	0.76%	0.33	-0.39%	0.24	-0.38%
tiff2bw	0.16	2.78%	0.55	1.15%	0.27	1.92%
tiffdither	0.19	7.44%	0.39	2.43%	0.25	2.57%
tiffmedian	0.17	4.99%	0.50	3.36%	0.26	4.01%
tiff2rgba	0.16	32.04%	0.68	0.05%	0.31	3.61%
gsm.encode	0.22	0.05%	0.24	-0.14%	0.23	-0.14%
typeset	0.20	4.99%	0.52	2.16%	0.29	3.30%
gsm.decode	0.22	0.05%	0.25	-0.17%	0.23	-0.17%

cold heap cache narrows the gap between unified and split heap configurations. Applications such as *jpeg.decode* and *tiff2rgba*, which contain a non-trivial number of accesses to the cold heap cache, see the greatest benefit from this modification, with *tiff2rgba* consuming 37% less energy with the smaller cold heap cache. Overall, these applications save between 69% and 78% of the total energy.

5 Conclusions

In this paper, we have evaluated a new multilateral cache organization designed to tailor cache resources to the individual reference characteristics of an application. We examined the characteristics of heap data for a broad suite of embedded applications, showing that the heap data cache footprint vary widely. To ensure that all applications perform well, we maintain two heap caches: a small, low-energy cache for frequently accessed heap data, and a larger structure for low-locality data. In the majority of embedded applications we studied, the heap footprint is small and the data possesses good locality characteristics. We can save energy in these applications by disabling the larger cache and routing data to the smaller cache, thus reducing both dynamic energy per access and static energy. This modification incurs a minimal performance penalty while reducing energy consumption by up to 79%. Those applications that do have a large heap footprint can use both heap caches, routing a frequently-accessed subset of the data to the smaller structure. Because a small number of addresses account for the majority of heap accesses, we can still reduce energy with both heap caches active—using up to 45% less energy—while maintaining high performance across all applications. When we implement drowsy caching on top of our split heap caching scheme, we can achieve even greater savings. With drowsy heap caches, disabling the larger structure allows for energy reductions between 77% and 84%, while activating both heap caches at once allows us to save up to 78% of the total energy.

In the future, we plan to further explore a few different aspects of this problem. We believe there is room for improvement in the heuristic used to determine which data belongs in which cache; we intend to refine that process. Also, the studies we ran using Cheetah suggest we can significantly lower the heap cache miss rate by reducing conflicts within it. We plan to investigate data placement methods as a means of ensuring fewer conflicts and better performance.

References

1. J. Montanaro, et al. A 160-MHz, 32-b, 0.5-W CMOS RISC Microprocessor. *Digital Technical Journal*, (1):49-62, January 1997.
2. M.J. Geiger, S.A. McKee, and G.S. Tyson. Drowsy Region-Based Caches: Minimizing Both Dynamic and Static Power Dissipation. *Proc. ACM International Conference on Computing Frontiers*, pp. 378-384, May 2005.
3. H.S. Lee and G.S. Tyson. Region-Based Caching: An Energy-Delay Efficient Memory Architecture for Embedded Processors. *Proc. ACM/IEEE International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pp. 120-127, November 2000.

4. H.S. Lee. Improving Energy and Performance of Data Cache Architectures by Exploiting Memory Reference Characteristics. Doctoral thesis, The University of Michigan, 2001.
5. K. Flautner, N.S. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy Caches: Simple Techniques for Reducing Leakage Power. *Proc. 29th IEEE/ACM International Symposium on Computer Architecture*, pp. 147-157, May 2002.
6. N.S. Kim, K. Flautner, D. Blaauw, and T. Mudge. Drowsy Instruction Caches: Leakage Power Reduction using Dynamic Voltage Scaling and Cache Sub-bank Prediction. *35th IEEE/ACM International Symposium on Microarchitecture*, pp. 219-230, November 2002.
7. N.S. Kim, K. Flautner, D. Blaauw, and T. Mudge. Circuit and Microarchitectural Techniques for Reducing Cache Leakage Power. *IEEE Transactions on VLSI*, 12(2):167-184, February 2004.
8. M.R. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. *Proc. 4th IEEE Workshop on Workload Characterization*, pp. 3-14, December 2001.
9. K. Ghose and M.B. Kamble. Reducing Power in Superscalar Processor Caches using Sub-banking, Multiple Line Buffers and Bit-Line Segmentation. *Proc. ACM/IEEE International Symposium on Low Power Electronics and Design*, pp. 70-75, August 1999.
10. C-L. Su and A.M. Despain. Cache Designs for Energy Efficiency. *Proc. 28th Hawaii International Conference on System Sciences*, pp. 306-315, January 1995.
11. J. Kin, M. Gupta, and W.H. Mangione-Smith. Filtering Memory References to Increase Energy Efficiency. *IEEE Transactions on Computers*, 49(1):1-15, January 2000.
12. D.H. Albonesi. Selective Cache Ways: On-Demand Cache Resource Allocation. *32nd IEEE/ACM International Symposium on Microarchitecture*, pp. 248-259, November 1999.
13. S.-H. Yang, M. Powell, B. Falsafi, and T.N. Vijaykumar. Exploiting Choice in Resizable Cache Design to Optimize Deep-Submicron Processor Energy-Eelay. *Proc. 8th International Symposium on High-Performance Computer Architecture*, pp. 147-158, February 2002.
14. 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. *Proc. 7th International Symposium on High-Performance Computer Architecture*, pp. 147-158, Jan. 2001.
15. R.A. Sugumar and S.G. Abraham. Efficient Simulation of Multiple Cache Configurations using Binomial Trees. Technical Report CSE-TR-111-91, CSE Division, University of Michigan, 1991.
16. L.A. Belady. A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM Systems Journal*, 5(2):78-101, 1966.
17. T. Austin. SimpleScalar 4.0 Release Note. <http://www.simplescalar.com/>.
18. D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. *Proc. 27th IEEE/ACM International Symposium on Computer Architecture*, pp. 83-94, June 2000.
19. Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan. HotLeakage: A Temperature-Aware Model of Subthreshold and Gate Leakage for Architects. Technical Report CS-2003-05, University of Virginia Department of Computer Science, March 2003.
20. D. Parikh, Y. Zhang, K. Sankaranarayanan, K. Skadron, and M. Stan. Comparison of State-Preserving vs. Non-State-Preserving Leakage Control in Caches. *Proc. 2nd Workshop on Duplicating, Deconstructing, and Debunking*, pp. 14-24, June 2003.
21. R. Gonzales and M. Horowitz. Energy Dissipation In General Purpose Microprocessors. *IEEE Journal of Solid State Circuits*, 31(9):1277-1284, September 1996.