Specializing Cache Structures for High Performance and Energy Conservation in Embedded Systems

Michael J. Geiger[1], Sally A. McKee[2], and Gary S. Tyson[3]

[1] ECE and CIS Departments, University of Massachusetts Dartmouth
North Dartmouth, MA 02747-2300
`mgeiger@umassd.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. We examine a new multilateral cache organization, replacing 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. In applications with small heap footprints, we save about 85% of the total cache energy. In the remaining applications, we employ a small cache for frequently accessed heap data and a larger cache for low locality data, achieving an energy savings of 80%.

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., they exhibit 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 total chip power [1]: as an alternative to general caching designs, specializing memory structures to better match data usage characteristics can 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 represents the most difficult region of memory to manage well in a cache structure. We propose a simple modification to demonstrate the benefits of increased specialization: large and small heap caches. If the application exhibits a small heap footprint, we save energy by using the smaller structure and turning off the larger. For applications with larger footprints, we use both caches, but save energy by keeping highly used "hot" data in the smaller, faster, lower-energy cache. The compiler determines (through profiled feedback or heuristics) which data belong where, 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 to cache where without requiring additional bits in memory reference instructions or complex coherence mechanisms.

This architectural approach addresses dynamic or switching energy consumption 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 are available. Drowsy caches save less power than many other leakage reduction techniques, but do not suffer the dramatic latency increases 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 minimal performance loss for these by using a smaller cache. We show energy savings of about 85% for both normal and drowsy reduced heap caches.
- For applications with large heap footprints that 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 splitting the heap cache into two structures. Using a small cache for hot data and a large cache for the remaining data, we show energy savings of up to 79% using non-drowsy split-heap caches and up to 80% using drowsy split heap caches.

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, increasing average observed L1 latency. Horizontal partitioning divides entities at a given level in the memory hierarchy to create multiple entities at that level. For instance, cache sub-banking [9][10] divides cache lines into smaller segments. 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 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 those 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 region-based paradigm can be extended to other areas of the memory system; for example, Lee and Ballapuram [12] propose partitioning the data TLB by semantic region.
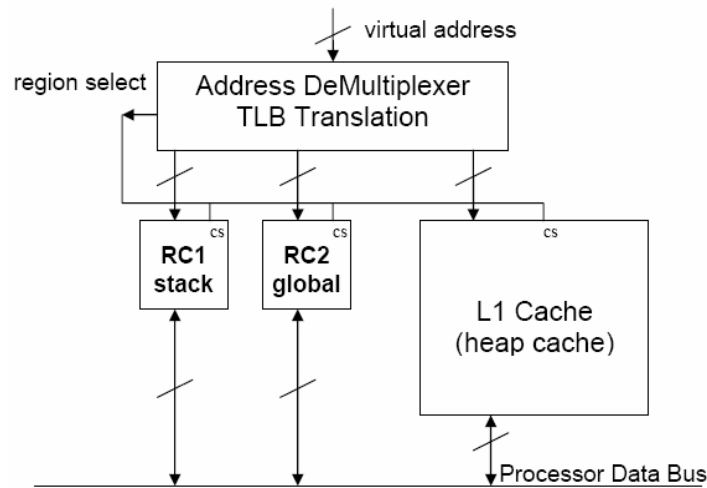

Fig. 1. Memory design for region-based caching

Other approaches to partitioning data by region aim to improve cache performance, not energy consumption. Several efforts focus on stack data, the most localized and frequently referenced data in most applications. Early machines like the HP3000 Series II [13] and the CRISP processor [14][15] contain caches specifically for stack references; in both cases, the stack cache is the only data cache in the processor. Cho et al. [16] decouple stack accesses from the rest of the data reference stream and route each stream to its own cache. The stack value file [17] and specialized stack cache [18] are two examples of cache-like structures customized to exploit characteristics of stack references. Techniques specifically addressing non-stack data are less common. In their analysis of the CRISP stack cache [14], Ditzel and McLellan note that global variables are also well suited to caching. The pointer cache [19] stores mappings between heap pointers and targets, but the structure a prefetching aid, not a data cache.

The downside to region-based caching is that increasing total cache capacity leads to higher static energy dissipation. Drowsy region-based caching [2] attacks this problem by implementing a leakage energy reduction technique [5][6][7] 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. Drowsy caches usually update cache line states after a given interval: lines may switch state from active to drowsy. Depending on the policy, lines accessed within an interval may remain active, and non-accessed lines become or continue to be drowsy. Geiger et al. [2] show that combining drowsy and region-based caching yields more benefits than either alone because each technique improves the performance of the other. Drowsy caching effectively eliminates static power increases due to additional region caches, while the partitioning strategy used in region-based caching permits more aggressive drowsy policies. The drowsy interval of each region cache can be tuned according to reference characteristics of that region, allowing highly active regions to be less drowsy than inactive regions.

The primary advantage of drowsy caching over other leakage reduction techniques is that drowsy cache lines preserve their state in low-leakage mode. Techniques that turn off unused cache lines by gating supply voltage [10][20][21] provide larger reductions in leakage energy. In these schemes, however, inactive lines must reload their state from L2 cache when accessed, and resulting performance penalties often outweigh power savings. Parikh et al. [22] dispute the notion that state-preserving techniques are superior, noting that the cost of drowsy caching depends on the latency of the transition between leakage modes.

Geiger et al. [2] introduce split heap caches. They find that for their application suite the ideal drowsy interval for the heap cache is the same as for the stack cache, a result that implies at least some heap data have locality similar to stack data. The authors observe that moving high-locality data to a separate structure allows a more aggressive drowsy caching policy for low-locality data, further reducing 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 show an average energy reduction of 71.7% over a unified drowsy L1 data cache.

3 Characteristics of heap data

In this section, we analyze characteristics of heap cache accesses in applications from the MiBench [8] benchmark suite to determine best heap caching strategies for each program. We begin by assessing the significance of the heap region within each target application, looking at overall size and number of accesses relative to the other semantic regions. Table 1 provides this information. The second and third columns of the table show number of unique block addresses accessed in the heap cache and number of accesses to those addresses, respectively. Since our simulations assume 32B cache blocks, 1KB 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). Several cases bear out our 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

Table 1. Characteristics of heap cache accesses in MiBench applications, including total footprint size, total number of accesses, and relative contribution of heap data to overall data footprint and reference count

Benchmark	# unique addresses	Accesses to heap cache	% total unique addresses	% total accesses
adpcm.encode	69	39971743	27.6%	39.9%
adpcm.decode	68	39971781	27.0%	39.9%
basicmath	252	49181748	61.2%	4.5%
blowfish.decode	213	39190633	39.0%	10.2%
blowfish.encode	212	39190621	38.9%	10.2%
bitcount	112	12377683	42.7%	6.7%
jpeg.encode	26012	10214537	99.2%	29.3%
CRC32	90	159955061	41.1%	16.7%
dijkstra	347	44917851	19.7%	38.3%
jpeg.decode	1510	7036942	90.2%	62.9%
FFT	16629	15262360	99.2%	8.6%
FFT.inverse	16630	14013100	99.2%	6.3%
ghostscript	59594	56805375	98.0%	15.3%
ispell	13286	28000346	96.5%	6.4%
mad	2123	40545761	82.3%	36.4%
patricia	110010	16900929	99.9%	6.6%
pgp.encode	298	252620	7.4%	1.9%
pgp.decode	738	425414	44.9%	1.5%
quicksort	62770	152206224	66.7%	12.9%
rijndael.decode	229	37374614	31.0%	21.7%
rijndael.encode	236	35791440	40.0%	19.6%
rsynth	143825	104084186	99.2%	21.4%
stringsearch	203	90920	18.2%	6.2%
sha	90	263617	20.9%	0.7%
susan.corners	18479	9614163	97.1%	63.6%
susan.edges	21028	22090676	99.1%	62.3%
susan.smoothing	7507	179696772	97.0%	41.7%
tiff2bw	2259	57427236	92.1%	98.5%
tiffdither	1602	162086279	83.1%	62.8%
tiffmedian	4867	165489090	53.0%	79.8%
tiff2rgba	1191987	81257094	100.0%	98.5%
gsm.encode	302	157036702	68.0%	11.7%
typeset	168075	153470300	98.0%	49.0%
gsm.decode	285	78866326	55.6%	21.5%
AVERAGE			65.7%	29.8%

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

We see that about half the applications have a 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 32KB L1 heap cache is likely far larger than these applications need; if we use a smaller heap cache, we dissipate less dynamic power per access with minimal effects on performance. Since heap cache accesses still comprise a significant percentage of total data accesses, this change should have a noticeable effect on dynamic energy consumption of these benchmarks. Shrinking the heap cache will also reduce static energy consumption. Previous resizable caches disable unused ways [23][24] or sets [24][25] 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 effects of this optimization on energy and performance.

Shrinking the heap cache may reduce the energy consumption for benchmarks with large heap footprints, but resulting performance losses may be intolerable for those applications. However, we can still 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 strive to choose the most frequently accessed lines. The access count gives a 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 2.1% of the cache lines cover 50% of the accesses. Although the rate of coverage decreases somewhat as you add more blocks—in other words, the first N blocks account for more accesses than the next N blocks—we still only need 5.8% to cover 75% of the accesses, 13.2% to cover 90% of the accesses, 24.5% to cover 95% of the accesses, and 45.5% 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.

Table 2. Number of unique addresses required to cover different fractions of accesses to the heap cache in MiBench applications. The data show that a small number of lines account for the majority of heap cache accesses, indicating that some of these lines possess better locality than previously believed. This trend is more apparent in applications with large heap cache footprints

Benchmark	# unique addresses	% unique addresses needed to cover given percentage of heap cache accesses				
		50%	75%	90%	95%	99%
adpcm.encode	69	1.4%	2.9%	2.9%	2.9%	2.9%
adpcm.decode	68	1.5%	1.5%	1.5%	1.5%	1.5%
basicmath	252	4.0%	25.4%	48.0%	55.6%	61.9%
blowfish.decode	213	0.9%	1.4%	2.3%	26.8%	55.9%
blowfish.encode	212	0.9%	1.4%	2.4%	26.9%	56.1%
bitcount	112	0.9%	1.8%	2.7%	3.6%	3.6%
jpeg.encode	26012	0.1%	0.6%	2.9%	38.2%	87.3%
CRC32	90	2.2%	3.3%	4.4%	4.4%	4.4%
dijkstra	347	0.3%	18.2%	39.2%	49.6%	63.1%
jpeg.decode	1510	4.8%	12.3%	31.9%	44.1%	59.5%
FFT	16629	0.0%	0.1%	4.8%	40.7%	85.3%
FFT.inverse	16630	0.0%	0.1%	13.0%	44.0%	86.5%
ghostscript	59594	0.0%	0.0%	0.6%	6.6%	57.5%
ispell	13286	0.1%	0.2%	0.5%	0.7%	1.3%
mad	2123	1.3%	2.6%	9.7%	14.9%	24.5%
patricia	110010	0.0%	0.1%	0.3%	36.6%	86.0%
pgp.encode	298	0.7%	1.0%	3.7%	6.7%	26.8%
pgp.decode	738	0.3%	0.4%	1.1%	2.3%	29.7%
quicksort	62770	0.0%	0.0%	0.2%	22.1%	49.1%
rijndael.decode	229	1.3%	2.2%	6.6%	31.4%	57.2%
rijndael.encode	236	1.3%	3.0%	7.6%	32.6%	56.8%
rsynth	143825	0.0%	0.0%	0.0%	1.3%	77.3%
stringsearch	203	17.2%	42.9%	59.6%	65.5%	72.9%
sha	90	1.1%	2.2%	3.3%	3.3%	8.9%
susan.corners	18479	0.0%	3.0%	11.0%	14.9%	32.7%
susan.edges	21028	0.0%	4.9%	15.1%	20.2%	30.4%
susan.smoothing	7507	0.0%	0.1%	13.7%	30.3%	44.1%
tiff2bw	2259	10.3%	15.4%	24.3%	29.4%	37.1%
tiffdither	1602	9.4%	19.6%	25.7%	29.6%	40.8%
tiffmedian	4867	4.0%	10.9%	16.7%	20.8%	47.8%
tiff2rgba	1191987	0.0%	0.1%	57.4%	78.7%	95.7%
gsm.encode	302	2.3%	4.0%	6.0%	7.6%	10.6%
typeset	168075	5.5%	15.4%	25.5%	33.0%	60.1%
gsm.decode	285	0.7%	1.4%	4.2%	6.0%	30.5%
AVERAGE (all apps)		2.1%	5.8%	13.2%	24.5%	45.5%
AVERAGE (>1k unique addrs)		2.0%	4.8%	14.1%	28.1%	55.7%

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 [26] 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 [27] and route the N most frequently accessed lines to the cache, varying N by powers of 2. We use optimal replacement to minimize conflict misses and give a sense of when the cache is filled to capacity; the actual miss rate for our direct-mapped hot heap cache will be higher.

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. As these tables show, 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. Note, however, that the miss rates remain tolerable for all applications for N values up to 256, regardless of cache size. The miss rate alone does not establish the suitability of a given caching scheme for heap data. Applications in which these accesses comprise a significant percentage of the total data references are less likely to be able to tolerate a higher miss rate. 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 that effectively incorporates miss rate estimates as well as footprint size and access percentages would likely yield better results.

Table 3. Miss rates for a fully-associative 2 KB cache using optimal replacement for different numbers of input addresses, N. These results establish a lower bound for the miss rate when caching these data. Applications shown either have a large heap footprint, which we define as a footprint of at least 1000 unique addresses, 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%

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 hot data in the smaller structure, we save dynamic energy. In all cases, we further lower static energy consumption by making the caches drowsy.

Our simulations use a modified version of the SimpleScalar ARM target [28]. We use Wattch [29] for dynamic power modeling and Zhang et al.'s HotLeakage [30] for static power modeling. HotLeakage contains a detailed drowsy cache model, which was used by Parikh et al. [22] 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]; the configuration details are shown in Table 6. We assume 130 nm technology with a 1.7 GHz clock frequency.

Table 6. Processor model parameters for heap caching simulations. All simulations use an in-order model based on the Intel StrongARM SA-110 [1]

Memory system	
Line size (all caches)	32 bytes
L1 stack/global cache configuration	4 KB, direct-mapped, single-ported, 1 cycle hit latency
L1 heap cache configuration	32KB, direct-mapped, single-ported, 2 cycle hit latency
L1 instruction cache configuration	16 KB, 32-way set associative, 1 cycle hit latency
L2 cache configuration	512 KB, 4-way set-associative, unified data/inst., 12 cycle hit latency
Main memory latency	88 cycles (first chunk) 3 cycles (inter chunk)

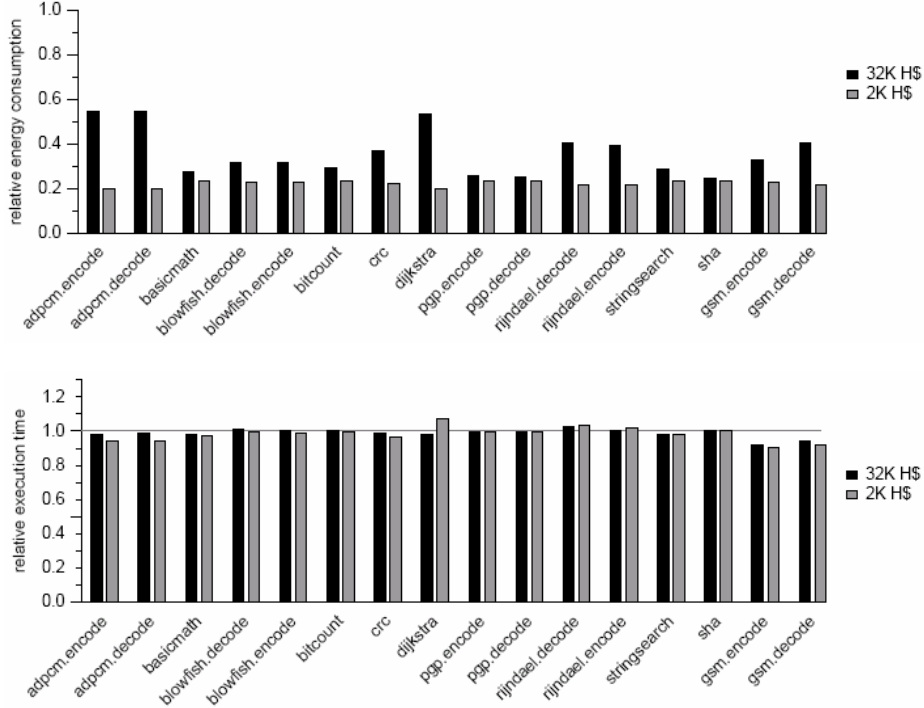| Execution engine	
Fetch/decode/ issue/commit width	1/1/1/1
IFQ size	8
Branch predictor	not taken
Integer ALU/multiplier	1 each
FP ALU/multiplier	1 each
Memory port(s) to CPU	1

Fig. 2. Energy (top graph) and performance (bottom graph) results for MiBench applications with small heap footprints (less than 1000 unique addresses) using region-based caches with large and small unified heap caches. The baseline is a 32KB direct-mapped unified L1 data cache. Speedups for the large heap cache are due to reduced conflicts between regions

Figures 2 and 3 show 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. We consider the most effective configuration to be the cache organization with the lowest energy-delay product ratio [31].

For applications with a heap footprint under 1000 lines, the split cache is unnecessary. Fig. 2 shows the results from these applications. Fig. 3, which shows applications with large heap footprints, adds the energy and performance numbers for the split cache. As expected, using the small heap cache and disabling the large offers the best energy savings across the board. Most applications consume close to 80% less energy in this case; however, some applications suffer significant performance losses, most notably *susan.corners* and *susan.edges*. 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
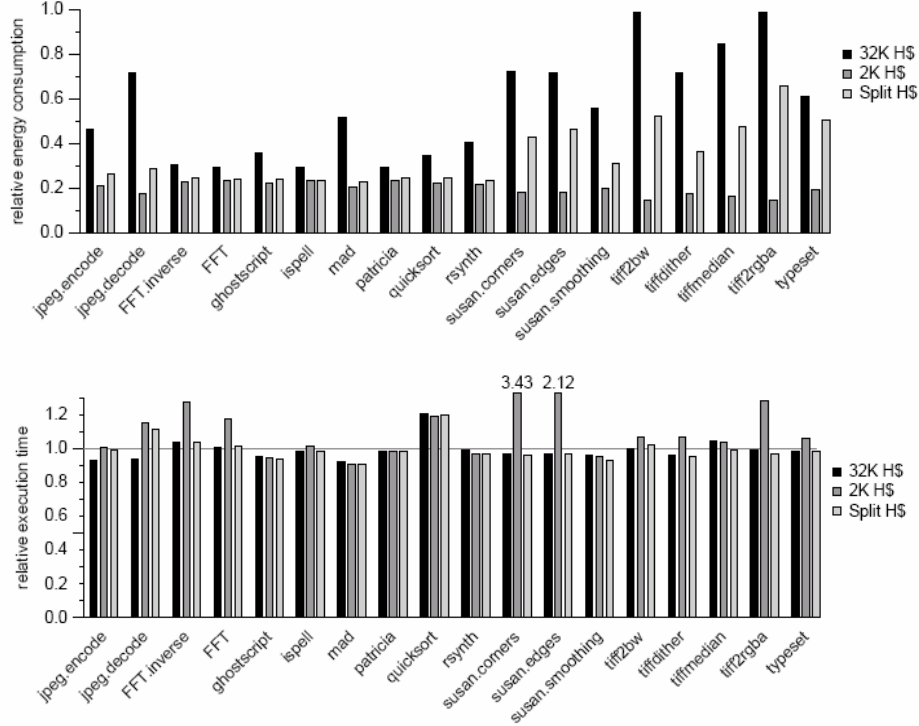
Fig. 3. Energy (top graph) and performance (bottom graph) results for MiBench applications with large heap footprints (greater than 1000 unique addresses) using three different heap cache configurations: a large unified heap cache, a small unified heap cache, and a split heap cache employing both large and small caches. The baseline is a 32KB direct-mapped unified L1 data cache. Speedups for the large heap cache are due to reduced conflicts between regions

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 global 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 86%. Several applications actually experience small speedups, a result of reduced conflict between regions and the lower hit latency for the smaller cache.

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 close to 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.
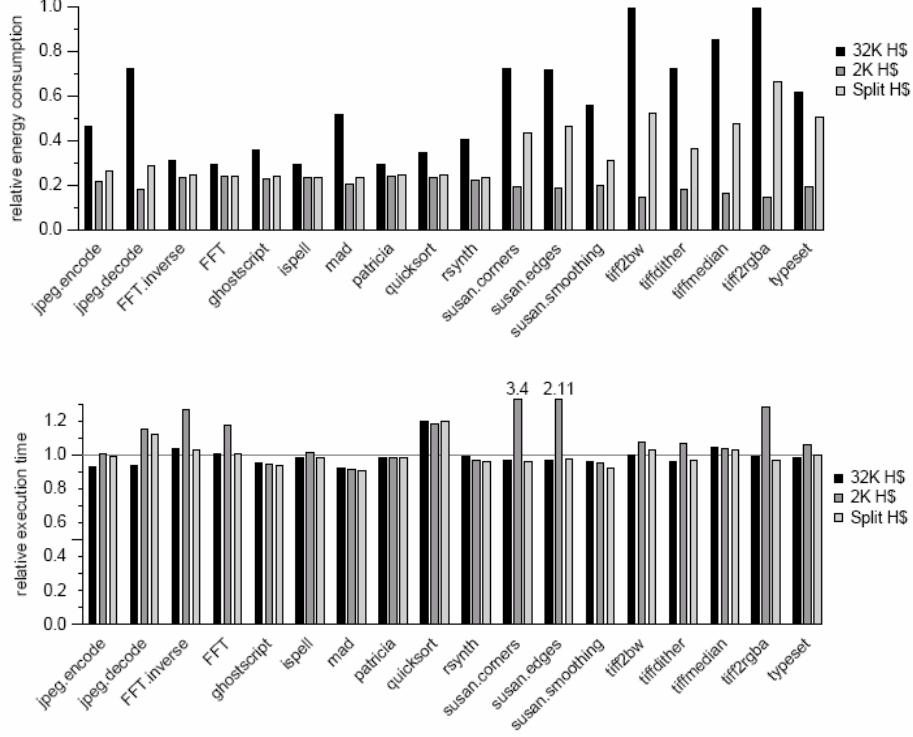
Fig. 4. Energy (top graph) and performance (bottom graph) results for MiBench applications with large heap footprints (greater than 1000 unique addresses) using three different heap cache configurations: a large unified heap cache, a small unified heap cache, and a split heap cache employing both large and small caches. The baseline is a 32KB direct-mapped unified L1 data cache with a 512-cycle drowsy interval

Fig. 4 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. We assume a 1 cycle latency for transitions to and from drowsy mode. 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 energy reduction offered by drowsy caching, this technique has the most profound effect on the split heap caches. Since the applications with small heap footprints do not require a split cache, the figure only shows the larger benchmarks. 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. Drowsy split heap caches save up to 80% of the
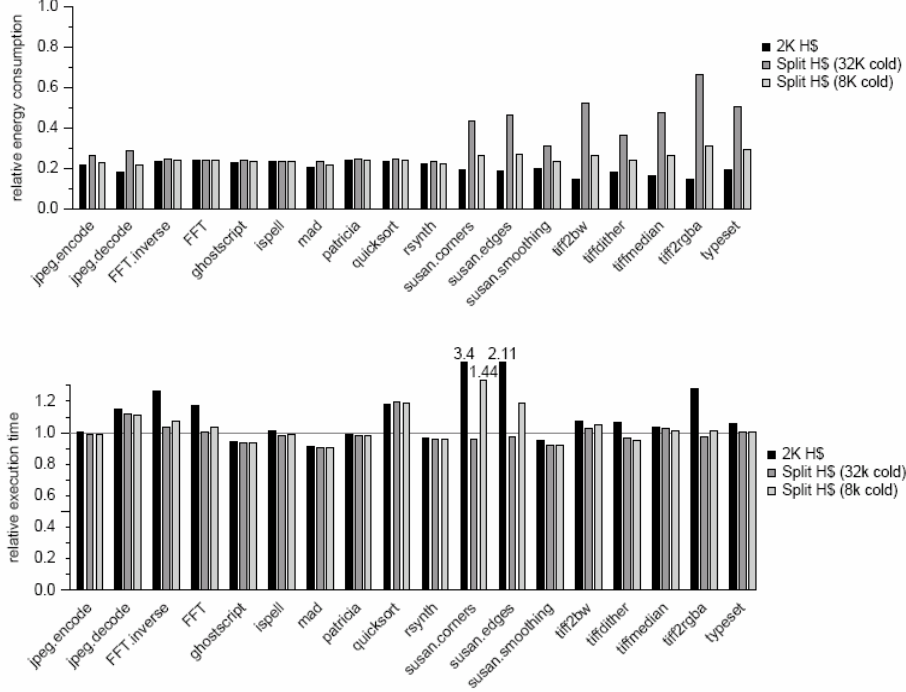
Fig. 5. Energy (top graph) and performance (bottom graph) results for MiBench applications with large heap footprints (greater than 1000 unique addresses) using three different heap cache configurations: a small unified heap cache, and split heap caches using either a 32KB cache or an 8KB cache for low-locality heap data. The baseline is a 32KB direct-mapped unified L1 data cache with a 512-cycle drowsy interval

total energy, while the small caches alone save between 76% and 85%. Because drowsy caching has a minimal performance cost, the runtime numbers are similar to those shown in the previous figure. 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. Fig. 5 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 cold heap cache narrows the gap between unified and split heap configurations. Applications such as *susan.corners* 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 36% less energy with the smaller cold heap cache. Overall, these applications save between 69% and 79% of the total energy.

5 Instruction-Centric Heap Caching

To this point, we have focused on analyzing heap data to determine how best to cache them. When only a subset of the data displays good locality, we use access frequency to identify hot data to store in a smaller cache. We now approach the same problem from a different angle—rather than looking at the locality characteristics of a particular line, we examine the references themselves. One advantage is that an instruction-based profile is often virtually independent of the program input. Although the data may affect how often a particular instruction executes, most programs follow the same general execution and therefore display the same relative behavior. Choosing hot data through their referencing instructions exploits locality in a different manner. Regularly accessed cache lines have high temporal locality. We cannot necessarily say the same about the targets of frequently executed memory instructions, as each instruction can access many addresses. However, this method effectively leverages spatial locality, as a single load often accesses sequential locations. Tight inner loops of program kernels display this behavior when accessing arrays or streams.

Table 7. Number of memory instructions that reference the heap required to cover different fractions of accesses to the heap cache in MiBench applications. As with the data itself, a small number of loads and stores account for the majority of heap cache accesses.

| Benchmark	# memory instructions	% memory instructions needed to cover given percentage of heap cache accesses				
		50%	75%	90%	95%	99%
adpcm.encode	171	1.2%	1.8%	1.8%	1.8%	1.8%
adpcm.decode	173	1.2%	1.7%	1.7%	1.7%	1.7%
basicmath	373	1.1%	4.8%	8.6%	10.5%	18.2%
blowfish.decode	325	1.2%	2.2%	2.5%	2.8%	2.8%
blowfish.encode	325	1.2%	2.2%	2.5%	2.8%	2.8%
bitcount	244	0.4%	0.8%	1.2%	1.6%	1.6%
jpeg.encode	1406	1.1%	3.0%	6.5%	8.9%	15.3%
CRC32	329	0.9%	1.2%	1.5%	1.5%	1.5%
dijkstra	383	0.8%	1.0%	1.3%	5.7%	14.4%
jpeg.decode	1192	1.2%	2.6%	4.9%	6.8%	11.8%
FFT	329	5.2%	11.2%	17.0%	20.4%	24.3%
FFT.inverse	327	4.9%	11.3%	17.7%	21.4%	25.1%
ghostscript	7501	0.2%	0.3%	1.5%	4.1%	13.5%
ispell	649	2.3%	4.2%	7.1%	10.8%	18.3%
mad	1043	2.9%	4.5%	7.2%	11.2%	15.0%
patricia	420	3.3%	10.7%	20.5%	23.8%	26.7%
pgp.encode	1119	0.4%	0.5%	2.1%	4.8%	22.5%
pgp.decode	1022	0.4%	0.6%	1.2%	2.7%	17.7%
quicksort	337	2.4%	5.9%	10.4%	12.5%	14.8%
rijndael.decode	540	13.7%	22.2%	27.4%	29.3%	31.1%
rijndael.encode	617	11.2%	18.3%	22.5%	24.1%	25.3%
rsynth	889	2.2%	4.2%	5.5%	7.6%	15.0%
stringsearch	210	1.9%	7.1%	11.9%	15.7%	21.9%
sha	276	1.1%	1.4%	1.8%	1.8%	8.0%
susan.corners	691	4.2%	8.0%	15.6%	18.5%	21.1%
susan.edges	878	6.8%	13.4%	21.5%	25.3%	28.6%
susan.smoothing	517	0.4%	0.6%	0.6%	0.6%	0.6%
tiff2bw	1036	0.4%	0.7%	1.1%	1.4%	1.8%
tiffdither	1314	0.6%	0.9%	2.8%	4.3%	6.8%
tiffmedian	1359	0.7%	1.0%	1.5%	1.8%	3.0%
tiff2rgba	1154	0.8%	1.6%	2.6%	2.9%	3.1%
gsm.encode	736	3.0%	5.3%	7.1%	8.2%	13.5%
typeset	17235	0.6%	1.9%	3.8%	6.7%	16.2%
gsm.decode	555	0.9%	1.4%	2.9%	3.4%	7.6%
AVERAGE		2.4%	4.7%	7.2%	9.0%	13.3%
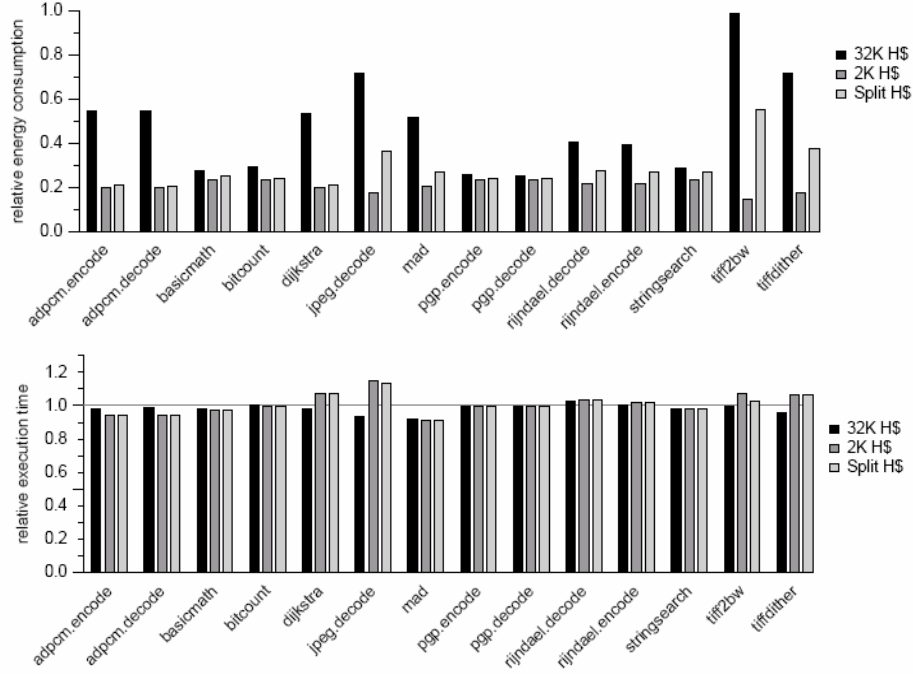
Fig. 6. Energy (top graph) and performance (bottom graph) for a subset of MiBench applications using different non-drowsy heap cache configurations. The baseline is a 32 KB direct-mapped unified L1 data cache. The hardware configurations are the same as in Figures 2 and 3, but in the split heap cache, data are routed to the hot heap cache based on the frequency of the accessing instructions, not references to specific blocks

In Table 2, we showed that a small number of blocks are responsible for the majority of heap accesses. This trend is even more apparent for memory instructions, as shown in Table 7. Just 2.4% of the loads and stores to the heap cache cover 50% of the accesses—a similar figure to the 2.1% of heap addresses required to cover the same percentage of accesses. The numbers do not increase greatly as we look at different coverage points, with approximately 13% of the memory instructions accounting for 99% of the heap references. These results reflect the oft-quoted maxim that programs spend 90% of their time in 10% of the code. Note that the number of instructions accessing the heap cache remains fairly consistent across applications, unlike the size of the heap data footprint. Our studies show that a small percentage of loads and stores access multiple regions.

The data suggest that we can treat heap references in a similar manner to heap data when determining how to cache this region. Because a small number of instructions account for most accesses, we can move their targets to a smaller cache, maintaining a larger cache for the remaining references. Note that only identifying the most frequently executed memory instructions will not sufficiently capture the appropriate accesses. Other memory references that share the same targets must also access the hot heap cache. Choosing appropriate instructions involves an iterative routine that ceases when the set of target addresses overlaps with no remaining references.
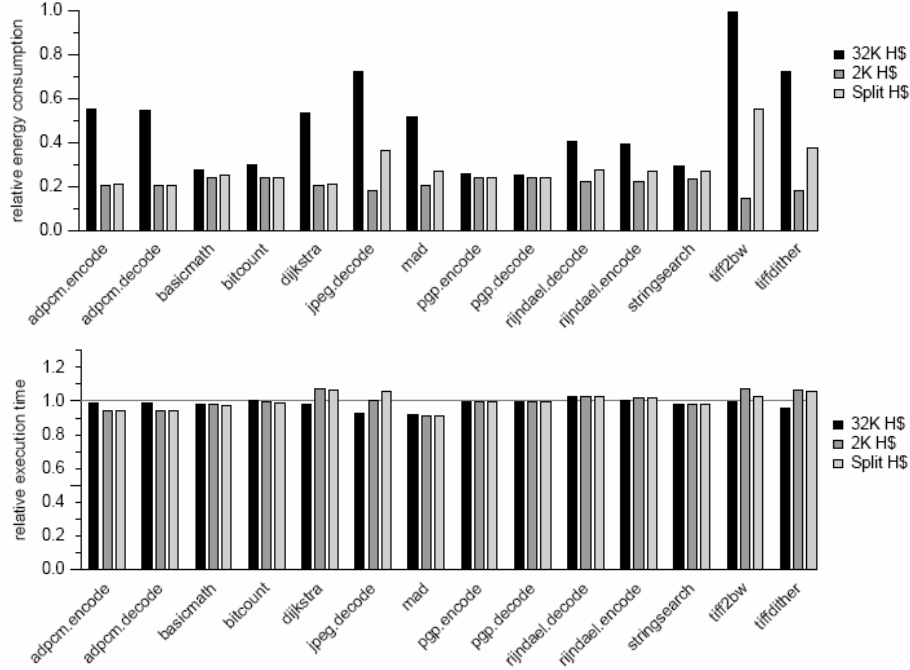
Fig. 7. Energy (top graph) and performance (bottom graph) results for a subset of MiBench applications using different drowsy heap cache configurations. The baseline is a 32 KB direct-mapped unified L1 data cache with a 512-cycle drowsy interval. The hardware configurations are the same as in Fig. 4, but in the split heap cache, data are routed to the hot heap cache based on the frequency of the accessing instructions, not references to specific blocks

Fig. 6 shows some preliminary results from this approach. As in Figures 2 and 3, we compare three non-drowsy cache configurations: a large (32 KB) unified heap cache, a small (2 KB) unified heap cache, and a split cache employing both large and small caches. We use the 128 most executed load instructions as a starting point for routing data between the caches. The table shows a subset of the MiBench applications, as the space requirements for the memory instruction profiles currently prevents the execution of some of the larger applications. Therefore, the small unified heap cache unsurprisingly represents the ideal design point for the benchmarks shown. For the most part, the results for the split heap configuration are similar to those shown in Figures 2 and 3, with energy savings ranging between 45% and 79%.

In Fig. 7, we evaluate drowsy heap cache configurations using the same routing methodology. As with the data shown in Fig. 4, the addition of drowsy caching significantly improves the energy consumption of the split cache, leading to comparable results for the small cache and split cache in several cases. Energy savings range from 45% to 79% in the split heap caches for the applications shown. These reductions match the reductions shown in the non-drowsy case; however, recall that the drowsy baseline already provides a significant energy savings over the non-drowsy baseline.

6 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 varies 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 86%. 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 79% 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 76% and 85%, while activating both heap caches at once allows us to save up to 80% of the total energy.

In the future, we plan to further explore a few different aspects of this problem. We believe that the instruction-based selection of hot heap data may ultimately hold more promise than the initial approach, and plan to explore this topic further through more detailed analysis of memory reference behavior. 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*, 9(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. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pp. 120-127, November 2000.
4. H-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 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. *Proc. 35th International Symposium on Microarchitecure*, 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. 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. H.S. Lee and C.S. Ballapuram. Energy Efficient D-TLB and Data Cache using Semantic-Aware Multilateral Partitioning. *Proc. International Symposium on Low Power Electronics and Design*, pp. 306-311, August 2003.
13. R.P. Blake. Exploring a stack architecture. *IEEE Computer*, 10 (5):30-39, May 1977.
14. D.R. Ditzel and H.R. McLellan. Register Allocation for Free: The C Machine Stack Cache. *Proc. 1st International Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 48-56, March 1982.
15. A.D. Berenbaum, B.W. Colbry, D.R. Ditzel, R.D. Freeman, H.R. McLellan, K.J. O'Connor, and M. Shoji. CRISP: A Pipelined 32-bit Microprocessor with 13-kbit of Cache Memory. *IEEE Journal of Solid-State Circuits*, SC-22(5):776-782, October 1987.
16. S. Cho, P-C. Yew, and G. Lee. Decoupling Local Variable Accesses in a Wide-Issue Superscalar Processor. In *Proc. 26th International Symposium on Computer Architecture*, pp. 100-110, May 1999.
17. H-H. S. Lee, M. Smelyanskiy, C.J. Newburn, and G.S. Tyson. Stack Value File: Custom Microarchitecture for the Stack. *Proc. 7th International Symposium on High Performance Computer Architecture*, pp. 5-14, January 2001.
18. M. Huang, J. Renau, S.-M. Yoo, and J. Torellas. L1 Data Cache Decomposition for Energy Efficiency. *Proc. International Symposium on Low Power Electronics and Design*, pp. 10-15, August 2003.
19. J. Collins, S. Sair, B. Calder, and D.M. Tullsen. Pointer Cache Assisted Prefetching. *Proc.35th IEEE/ACM International Symposium on Microarchitecture*, pp. 62-73, November 2002.
20. S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: Exploiting Generational Behavior to Reduce Cache Leakage Power. *Proc. 28th International Symposium on Computer Architecture*, pp. 240-251, June 2001.
21. 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. *Proc. International Symposium on Low Power Electronics and Design*, pp. 90-95, July 2000.
22. 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.
23. D.H. Albonesi. Selective Cache Ways: On-Demand Cache Resource Allocation. *Proc. 32nd International Symposium on Microarchitecture*, pp. 248–259, November 1999.
24. S.-H. Yang, M. Powell, B. Falsafi, and T.N. Vijaykumar. Exploiting Choice in Resizable Cache Design to Optimize Deep-Submicron Processor Energy-Delay. *Proc. 8th International Symposium on High-Performance Computer Architecture,* pp. 147-158, February 2002.

25. 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.

26. 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.

27. L.A. Belady. A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM Systems Journal*, 5(2):78-101, 1966.

28. T. Austin. SimpleScalar 4.0 Release Note. http://www.simplescalar.com/.

29. D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. *Proc. 27th International Symposium on Computer Architecture*, pp. 83-94, June 2000.

30. 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.

31. R. Gonzales and M. Horowitz. Energy Dissipation In General Purpose Microprocessors. *IEEE Journal of Solid State Circuits,* 31(9):1277-1284, September 1996.