Tracking Object Life Cycle for Leakage Energy Optimization *

G. Chen, N. Vijaykrishnan, M. Kandemir, M. J. Irwin
Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802
{gchen, vijay, kandemir, mji}@cse.psu.edu

M. Wolczko
Sun Microsystems, Inc.
2600 Casey Ave
Mountain View, CA 94043
mario@eng.sun.com

ABSTRACT

The focus of this work is on utilizing the state of objects during their lifespan in optimizing the leakage energy consumed in the data caches when executing embedded Java applications. Our analysis reveals that a major portion of the leakage energy is actually wasted in retaining the objects beyond their last use. In order to eliminate this wastage, we investigate three approaches that use the garbage collector, escape analysis and last use analysis for reducing leakage energy. Finally, we track the access gap between successive object accesses to reduce leakage energy of live objects. A combination of these schemes is shown to provide 21% data cache leakage energy reduction in our default configuration.

Categories and Subject Descriptors

B.3.m [**Hardware**]: B.3 Memory Structures—*Miscellaneous*

General Terms

Algorithms, Experimentation

Keywords

Java, cache, leakage energy

1. INTRODUCTION

Optimizing power consumption has become important for a variety of systems ranging from high-performance systems to low end battery-operated devices. While optimizing dynamic power consumption has been the focus of most of the previous work, static power consumption due to leakage current is an important concern in future technologies [1]. Unlike dynamic energy consumption, static power is consumed independent of whether the component is accessed or not. Specifically, leakage power consumption is primarily dependent on the technology used and the number of transistors used for the design. As on-chip caches account for a major portion of the processor's transistor budget, they constitute a major portion of the energy budget of the processor. Leakage energy is projected to account for 70% of the cache power budget in 70nm technology [2].

We can reduce leakage by completely gating the supply voltage of cache lines. However, when the supply voltage is gated, the data stored in the cache line is lost (state-destroying leakage control) [3]. Hence, if another access is required to the same data, it should be fetched from lower levels of memory hierarchy, and this incurs a large performance penalty than using state-preserving leakage control. We refer to placing a cache line in low leakage mode as "turning off the cache line" and restoring the normal supply voltage as "turning on the cache line" in the rest of this paper. We use this term for both state-preserving and state-destroying modes and distinguish between them when relevant.

The effectiveness of the leakage reduction depends on how precisely the behavior of cache line can be tracked. While turning off a cache line later than the last use can waste energy consumption, prematurely turning off a cache line can incur energy/performance penalties when it needs to be accessed. Thus, deciding when to turn off a cache line is very important. In this work, we utilize the knowledge about the state of an object in its lifespan to direct the turning off cache lines. In particular, we identify different states in the lifetime of an object, when it is created, last-used, becomes garbage, and is collected by the garbage collector. It must be observed that the cache lines containing only objects beyond their last use waste leakage energy. Our analysis in this paper reveals that this wasted leakage energy contributes to a significant portion of data cache energy consumption. In order to control the turning off cache lines more precisely, we propose a series of object in its lifespan. Almost none of prior approaches (e.g., [4, 5]) utilizes the object lifetime information in optimizing leakage energy with the exception of [6] that manages the leakage in on-chip memory using the garbage collector. In this paper, in addition to the garbage collector, we further exploit object lifetime information by applying escape and last use analyses to find more leakage reduction chances. Further, as compared to the work in [6], we focus on the data cache rather than leakage control in memory where caching influences the potential of the scheme.

This work explores three different approaches that save cache leakage energy. In our first approach, the garbage collector is used to turn off the cache line(s) containing the collected objects. Note

*This research is supported in part by NSF Awards 0103583, 0130143; NSF CAREER Awards 0093082, 0093085; MARCO 98-DF-600 GSRC.

CODES+ISSS'03, October 1–3, 2003, Newport Beach, California, USA.

that in this case it is preferable to employ state-destroying leakage control as the object will never be accessed again. Our second technique exploits the observation that, for many objects, their lifetime is contained within the lifetime of a particular methods execution. We refer to such objects as "method local objects". Using a trace-based escape analysis technique, we identify the method local objects and turn the corresponding cache lines off when the method to which it is local completes its execution. Thus, this scheme can turn off cache lines much earlier than the garbage collector. However, only method local objects can benefit from this scheme. Our third technique attempts to be even more precise. We identify the last-use of the object by identifying the instruction that last uses it. We turn off the cache line containing the object immediately after executing this instruction. After these three optimization strategies, we focus our attention on object access times and show that it is beneficial to turn-off cache lines containing objects which have a large gap between their successive accesses.

In exploring these schemes, we focus on objects in embedded Java applications. Our choice is influenced by the wide adoption of Java in the energy-constrained mobile devices [7, 8]. The results of our evaluation of using ten embedded Java applications shows that the three proposed approaches provide significant savings in leakage energy consumption.

The remainder of this paper is organized as follows. In the next section, we introduce our experimental setup and the benchmark suite used. In Section 3, we show the energy consumption profile in the data cache based on object lifespan. Next, we evaluate our energy optimization schemes that use garbage collection, escape analysis and last-use analysis in Sections 4 through 6. In Section 7, we present how to optimize energy consumed by live objects. Finally, we provide conclusions in Section 8.

2. EXPERIMENTAL SETUP AND BENCHMARKS

We use ten applications that target embedded Java-based environments (Table 1). We believe these applications represent a good mix for typical Java-enabled personal devices. For obtaining detailed energy profiles, we have customized an energy simulator and analyzer using the Shade [9] (microSPARC instruction set simulator) tool-set and simulated the entire KVM (a small-footprint Java virtual machine designed for resource-constrained environments [10]) executing a Java code. As many of our optimization techniques are profile-based, we used a different training set to get the profiles and a different input set for evaluating the energy benefits. Shade is used to capture the cache access pattern of the application while the KVM was augmented with instrumentation code to track the lifespan of the objects. Figure 1 shows our experimental setup.

Once the cache access patterns are captured and the state of the cache lines is determined after applying the optimizations, we can evaluate the energy consumption. In our experiments, we assume that the processor has 32KB ICache and 32KB DCache. The dynamic energy for each cache access is 0.302nJ. When a cache line is in the active mode, it consumes 0.147pJ energy each cycle; while in the sleep mode, it does not consume leakage energy. These energy parameters are based on future 70nm CMOS technology. The cache data energy is obtained by using CACTI 3.0 [11]. Each cache line is either in *active* or *leakage control (sleep)* mode. A cache line needs to be in active mode to serve a cache access and consumes the maximum leakage energy. When a cache line is in the sleep mode, we assume state-destroying leakage control and assume zero leakage energy. Accessing a sleep cache line involves loading the data from main memory, which is similar to a cache miss. The wakeup delay of the sleep cache line is overlapped with the delay due to the main memory access.

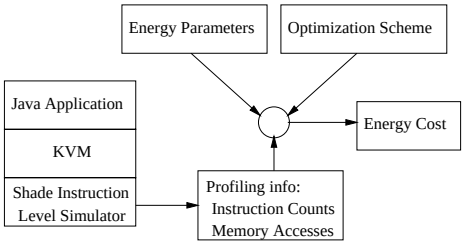

Figure 1: Experimental setup.

Benchmark	Brief Description	Heap Energy (mJ)		Cache Leakage Contribution
		Memory	Cache	
calculator	Arithmetic calculator for MIDP	0.84	4.21	84.9%
chess	User vs. computer chess game	16.26	87.38	80.4%
emailviewer	Light-weighted POP3 Client	2.65	16.29	86.3%
manyballs	Drawing bouncing balls	0.60	3.26	84.9%
mdoom	3D Shooting game	8.29	44.27	84.9%
mpg	Movie player for MIDP	7.16	35.64	80.9%
pushpuzzle	A conventional puzzle game	1.28	7.18	83.1%
scheduler	Personal monthly scheduler	0.67	8.08	94.3%
sfmap	Digital map for MIDP	3.47	17.03	85.9%
webviewer	WWW browser for MIDP	5.18	28.25	84.5%

Table 1: Our benchmarks.

In order to control the leakage state, we support a special instruction, *deactivate(address, length)*, that can turn off cache lines and place them in sleep mode. This instruction takes two parameters, the starting address and the number of bytes (length) from the starting address to place in sleep mode. The deactivate instruction compares the tag of each cache line for addresses in this region sequentially and turns off cache lines whenever the tags match. Note that the tag match is necessary as only a part of the object may be present in the cache. A *deactivate* instruction may take several cycles to execute based on the size of the region being turned off. In order not to create conflicts in address port with other data cache accesses, the *deactivate* instruction needs an extra address decoder to control the supply voltage to cache lines. It should be noted that the *deactivate* instruction does not need to access the data itself.

3. OBJECT LIFETIMES AND CACHE ENERGY CONSUMPTION PROFILE

Table 1 shows the energy consumed in the data cache and memory due to the heap accesses for the applications in our suite. The last column of the table indicates the contribution of leakage energy to the cache energy. We can observe that it is a major contributor to overall cache energy.

Figure 2 illustrates the lifetime of a Java object. An object is created, used a number of times, and then reaches its last use. Following this, it becomes garbage, and after some time, it is collected (by the garbage collector). It should be pointed out that this object can be in the cache during these different phases of its lifetime. In fact, even after it is garbage collected, it can still be in the cache. Obviously, energy spent on a cache line that keeps an object which has passed its last-use is wasted, and should be reduced as much as possible.

Since the leakage current can be reduced only at the granularity of cache lines and not at the granularity of objects, it is important to identify how objects are mapped to cache lines. For an object whose size is larger than one cache line, all the cache lines occupied by it are called *relevant cache lines (RCLs)* (Figure 3). Since the

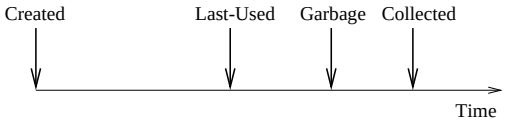

Figure 2: Lifetime of an object.

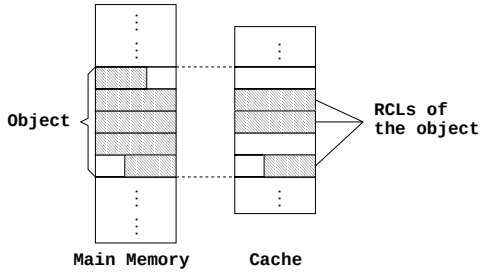

Figure 3: Relevant Cache lines of an object

objects are not necessarily aligned to the beginning of cache lines, it is possible for multiple objects to be contained in the same cache line. Further, these multiple objects can be in different phases of their lives. For example, two objects can share a cache line, and one of them might be live (i.e., its last use has not been reached yet), whereas the other one can be garbage. To capture these cases, we classify cache lines into the groups given in Table 2 according to their *states*. As an example, the state of a cache line is tagged as 'Last-Used' if it does not have a live object and has parts of (at least) one last-used object. It can also have objects that are garbage or collected. In this table, 'Not-Used' refers to a state where the cache line is not used by the application being executed.

Based on the discussion above, one can see that a cache line can be in one of the five states: live, last-used, garbage, collected, and not-used. Figure 4 shows the percentage data cache energy consumption breakdown as dynamic energy and static energy components. The static energy part is further divided into energies expended in different cache line states. This graph clearly shows that a large percentage of the cache energy is static energy consumed for maintaining cache lines that do not keep any useful data. In fact, a major portion of the leakage energy consumed in the caches is actually wasted because cache lines are not turned off when holding objects beyond their last use. Eliminating this energy consumption through object lifetime analysis can be very beneficial in practice.

Figure 4 also indicates the upper bounds on energy savings that can be achieved using optimal algorithms that target different phases of object lifetimes. For example, an optimization strategy that turns off cache lines holding collected objects can reduce overall cache energy by 10% for `emailviewer`. It should be noted that such a strategy can be implemented within the garbage collector itself. On the other hand, a different optimization strategy that turns off cache lines immediately after such lines transition to the garbage state can eliminate energies due to both cache lines in "garbage" and "collected" states. Finally, a strategy that turns off cache lines immediately after they transition to the last-used state can eliminate all wasted energy due to storing objects beyond their last use (saving 47% cache energy on the average). Up to this point, we have

	Objects that may be contained			
Status	Live	Last-Used	Garbage (Dead)	Collected
Live	Yes	Maybe	Maybe	Maybe
Last-Used	No	Yes	Maybe	Maybe
Garbage	No	No	Yes	Maybe
Collected	No	No	No	Yes
Not-Used	No	No	No	No

Table 2: Classification of cache lines into different states.

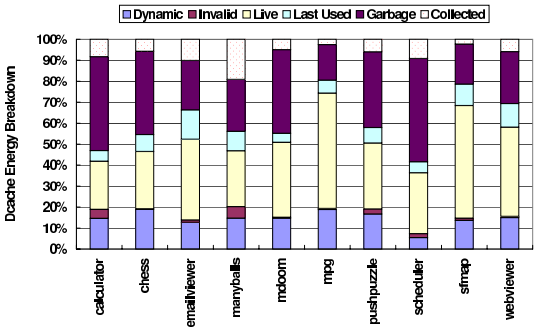

Figure 4: Energy breakdown in base case in data cache.

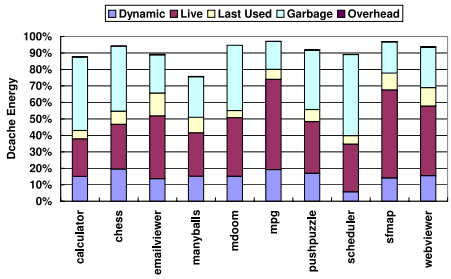

Figure 5: Data cache energy results for GC controlled cache line turnoff optimization (do this for other similar figures, too).

focused only on the wasted energy that is consumed after an object reaches its last use. However, objects that are live also cause their RCLs to consume leakage energy. If the gap between successive object accesses is large, it would be more beneficial to turn-off the cache lines rather than expend leakage energy during that interval. The focus of the rest of the paper is to implement schemes that can achieve close to the maximum energy savings possible while minimizing the overheads due to the implementation of the schemes themselves.

4. GARBAGE COLLECTION CONTROLLED CACHE LINE TURNOFF

KVM invokes the garbage collector(GC) when it runs out of heap memory. The collection is performed in two phases. In the mark phase, the collector traverses the reference graph from the roots to mark all the objects that are reachable from the roots. In the sweep phase, the collector collects all the objects that are not marked in the mark phase and merges them into free blocks. These free blocks are returned to the free memory pool for future allocation. Since the collected objects have been merged, we can turnoff the RCLs of several collected objects with contiguous addresses using one deactivate instruction. It should be noted that, during garbage collection, the header of each object (either live or dead) is accessed.

Figure 5 shows the impact of this garbage collector controlled cache line turnoff. All the energy values are normalized to the energy consumption of the cache without any optimization, as is shown in Table 1. The energy results for the optimized strategy also include the overhead energies consumed due to the execution of the deactivate instructions and for turning on the cache lines that are in sleep mode. Note that sleep cache lines need to turn-on when the next object maps onto the same cache line. On an average, this GC controlled cache line turnoff saves the overall data cache energy consumption by 9%. All the other schemes discussed in the rest of the paper use the GC controlled scheme.

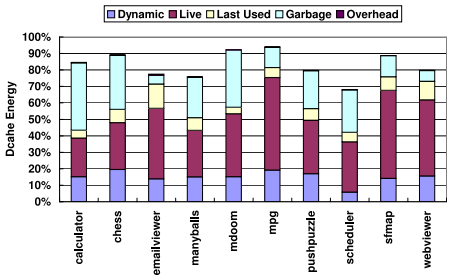

Figure 6: Data cache energy of escape analysis optimization. Also uses GC controlled leakage.

5. ESCAPE ANALYSIS

Prior research [12, 13] has shown that a big portion of objects do not survive beyond the lifetime of the method where these objects are created. These objects are called the "method local objects" since their lifetime is limited by that of their creating method. Using escape analysis [12, 13], we can identify the method local objects and turn off the cache lines occupied by them when the corresponding method returns. The key to escape analysis is to identify the local allocation sites, i.e., the bytecodes that allocate only local objects. In contrast to static analysis techniques employed in [12] and [13], we use a trace-based approach to identify method local objects. Specifically, we run each application with the training input sets using an instrumented virtual machine, which records the allocation site of each object. When a method returns, the instrumented virtual machine scans the heap to find out which objects that have been created within this method are still accessible. If an object created within the method is still alive, the allocation site of this object is marked as "global". When we are done with the training input sets, the instrumented virtual machine outputs the location of each global allocation site of each method. This information on the global allocation sites is then added to the corresponding class files using our class file annotation tool. During run time, when a class file is loaded, the virtual machine checks the annotations and marks each global allocation site. The unmarked allocation sites are regarded as local allocation sites.

We allocate all the objects created at the local allocation sites during the same invocation of each method together in a contiguous address space. When a method returns, all RCLs corresponding to its local objects can be turned off. The use of a contiguous address space helps to turn off all the cache lines containing the local objects of this method using only a single deactivate instruction. Note that since cache line turn-off using state-destroying leakage mode only evicts the copy in the data cache, even an accidental eviction of a non method local object is not a correctness issue (it may only hurt performance). Note that dirty (dirty indicates that the cache line was modified since it was brought into the cache) cache lines are always written back to the next level when they are turned off to maintain cache coherence. Figure 6 shows the effectiveness of escape analysis in conserving the energy consumption of the data cache when used along with garbage collector based turn off. Comparing Figure 6 with Figure 5, we can observe that the percentage of the leakage energy that is consumed by the cache lines containing garbage objects is reduced. On the average, we achieve 17% overall data cache energy savings in our 10 benchmarks by turning-off method local objects immediately after the corresponding method returns.

6. LAST USE ANALYSIS

Last use analysis is used to turn off the RCLs of objects more aggressively immediately after their last accesses. In the program

```
int f(int n) {
    Object o = new Object();
    int i = 0;
    int s = 0;
    while(++i < N) {
        access(o);
    }
    for(i=0; i<1000; i++) {
        s += g();
    }
    return s;
}
```

Figure 7: An example code fragment.

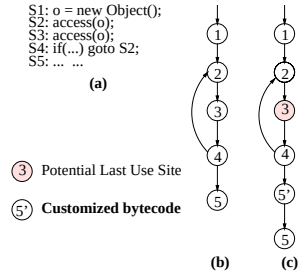

Figure 8: Last use analysis.

shown in Figure 7, we observe that object *o* will never be accessed after the while loop. Consequently, we can turn off the object *o* immediately after the exit of the while loop (i.e., much earlier than the exit of the method or before a garbage collector can collect it). It should be noted that, although the application does not access those last used objects, their headers and reference fields (for last used but still live objects) are still accessed in the garbage collection. This may cause extra cache misses and energy overhead (this overhead is also captured in our results). Fortunately, for most applications, garbage collection is not invoked very frequently.

In order to perform the last use analysis, we utilize a "last-reference register" that indicates the address of the object that has been last used. This can then be used by a deactivate instruction that is inserted into the bytecode sequence to turn-off the RCLs of the object pointed to by the last-reference register. The key here is to identify the last use sites (i.e., the bytecodes that will update the last-reference register) and where the deactivate instruction should be inserted. In some cases, it may not be possible to turn off the cache line at the last use site since the same static site can access the object multiple times. Hence, it is necessary to distinguish between the last use site and where the cache lines that hold the object can be turned-off.

Figure 8 shows an example of how our last use analysis works. Figure 8 (a) is a code fragment and (b) is its Control Flow Graph (CFG). From this CFG, we can identify a loop which consists of nodes 2, 3 and 4. Let us assume that object *o* is not accessed outside the loop. Both S2 and S3 access object *o* many times in the loop. However, none of the accesses made by S2 is a last access. Therefore, S2 is not a potential last use site. On the other hand, at the last iteration of the loop, S3 makes a last access to object *o*. Therefore, we mark S3 as a potential last use site. At S5, we observe that all the objects accessed by the previous potential last use site (S3) have become last used. Therefore, we insert a deactivate instruction (S5') right before S5 to turn off the RCLs of the objects that are last-used by S3.

Our current implementation uses training sets and profiling traces to find the addresses of bytecodes that make last uses and the addresses for inserting the *deactivate* instructions. Specifically, we first run each application with training input sets using our instrumented virtual machine, which keeps track of each bytecode and generates a detailed trace file. After this, we analyze the trace file using the following three steps to identify last use sites and the

	Code Fragment	Step 1	Step 2		Step 3	
			A	B	A	B
#1	i = 0		0	1	0	1
#2	j = 0		99	1	0	100
#3	o = new Object		99	1	0	100
#4	access o		99	401	0	100
#5	access o	mark	99	401	0	100
#6	j = j + 1		100	400	100	400
#7	if(j<5) goto 4		100	400	100	400
#8	i = i + 1		100	0	100	0
#9	if(i<100) goto 2		100	0	0	100
#10	return		1	0	0	1

Figure 9: Example: finding the locations to insert deactivate instructions.

addresses to insert deactivate instructions. An example is shown in Figure 9. The results of this analysis are added to the corresponding class files as annotations using our annotation tool. These annotations are used by the virtual machine at runtime to optimize data cache energy consumption (the details will be explained later).

Step 1: We scan the trace in the reverse order of execution to find: (1) when each object is last used; and (2) which bytecodes have ever made last accesses. The bytecodes that have ever made last accesses are marked as potential last use sites (see the "mark" in the third column of Figure 9).

Step 2: We scan the trace in the order of execution. In this scan, we update the last-reference register right after each execution of the bytecode that has been marked as potential last use site in step 1. Note that this register is shared by all potential last use sites. Further, associated with each static bytecode in the program are two counters: A and B. Before the execution of each bytecode, if the last-reference-register is not null and it refers to an object that has become last used (by comparing the object's death time against the execution time of current bytecode), we increase the counter A of the current bytecode by one. Otherwise, the counter B of this bytecode is increased by one. Obviously, $A + B$ is equal to the number of times that this bytecode has been executed. The fourth column in Figure 9 shows the contents of counters A and B for each bytecode after the code fragment (in the second column) returns. From this column, we observe that bytecode #6 is executed 500 times, for 100 of which the last-reference-register refers to the last used objects. Note that the initial value of the last-reference-register is null, and that the reference that is stored in the last-reference-register by bytecode #5 remains unchanged until next loop iteration. Therefore, we observe that A counters of bytecodes #2, #3, #4 and #5 are all set to 99. After the scan, all the bytecodes with $A/(A+B) > T$ (T is a given threshold, T=95% in this work) are marked as candidates for inserting the deactivate instruction.

Step 3: We scan the trace once more in the order of execution. The scan in this step is similar to that in step 2 except that, after the execution of each bytecode that has been marked as candidates for inserting in step 2, the last-reference-register is set to null. Our purpose here is to filter out the redundant insertion candidates. After this scan, the bytecodes whose A and B counters still satisfy $A/(A + B) > T$ are marked as inserting points. Their addresses, as well as the addresses of the last use sites that have been found in step 1, are fed into a class file annotation tool, which adds a notation to each Java method indicating which bytecodes in the method are inserting points and which are last use sites. The last column of Figure 9 gives the contents of counters A and B for each bytecode after the code fragment returns. In this example, only bytecode #8 is determined to be an inserting point.

During execution, when the notated Java class files are loaded, the virtual machine marks the bytecode at each last use site and inserts a deactivate instruction right before each bytecode that is notated as an inserting point. Right after each marked bytecode is executed, the virtual machine updates the last-reference-register. The inserted deactivate instruction uses the address in the last-reference register to identify the object and relevant RCLs to turn-off. Note that our approach is not always accurate in determining last use and sometimes may cause some of the objects to be turned off prematurely. This premature turn off may incur additional performance penalties for fetching the data again from memory (or L2 cache); however, this does not affect the correctness of execution. Our experiments (see Table 3) demonstrate that the impact of such extra cache misses is marginal. Another problem is that executing the inserted deactivate instructions incurs overhead. Table 4 shows the actual number of deactivate instructions inserted in the selected applications. Compared to the total number of bytecodes of the corresponding benchmarks, the numbers of inserted instructions are very small. The third/fourth column in Table 3 shows the cache miss rate/execution time increase of our ten benchmarks with the last use analysis optimization. Note that in some cases the miss rate is reduced and the performance is improved. This is because our optimization prevents the last-used data from competing for the cache lines with live data. Figure 10 shows the impact of last use analysis on the data cache energy consumption when used in conjunction with the GC-controlled leakage management. Last use analysis can also be combined with escape analysis to further conserve the leakage energy (denoted as "Escape + Last Use" in Figure 12). On the average, we achieved 11% data cache energy savings through last use analysis optimization. When it is combined with escape analysis, it saves 6% (on average) more energy as compared to using escape analysis without last use analysis. It should be noted that, in both cases, we use the GC-controlled leakage management as well.

Benchmark	Baseline	Last Use		Access Gap	
	Cache Miss Rate (%)	Cache Miss Rate (%)	Exec Time Increase (%)	Cache Miss Rate (%)	Exec Time Increase (%)
calculator	1.67	1.99	0.54	3.86	3.69
chess	1.83	1.85	0.04	2.66	1.89
emailviewer	4.50	4.54	0.07	5.65	1.61
manyballs	1.57	1.62	0.09	3.69	3.61
mdoom	1.44	1.41	-0.04	2.96	2.61
mpg	0.88	0.89	0.03	2.67	4.10
pushpuzzle	1.14	1.12	-0.04	1.65	1.01
scheduler	2.45	2.56	0.06	5.03	1.47
sfmap	2.15	2.19	0.07	3.22	1.67
webviewer	2.03	2.05	0.03	3.81	3.06

Table 3: Performance degradation due to last use and access gap analysis optimization

Benchmark	Last Use Inserted	Access Gap Marked
calculator	149	469
chess	130	818
emailviewer	206	1594
manyballs	98	394
mdoom	110	603
mpg	129	866
pushpuzzle	128	718
scheduler	133	566
sfmap	116	578
webviewer	196	1399

Table 4: The number of inserted deactivate instructions in last use analysis and the number of marked instructions in access gap analysis.

7. ACCESS GAP ANALYSIS

The optimization schemes that we have presented so far focus on the the RCLs of the objects that are no longer used by an application. The merit of these schemes is that they do not increase cache miss rate (except the last use analysis). Access gap analysis scheme analyzes the intervals between two consecutive accesses to each objects and turns off RCLs when the intervals are long enough. The

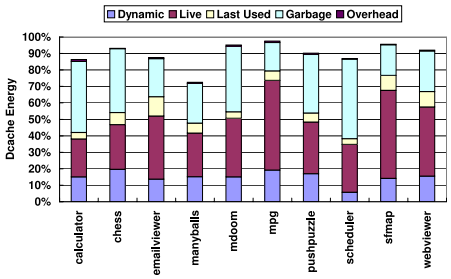

Figure 10: Data cache energy of last use analysis optimization. (The GC controlled scheme is also used.)

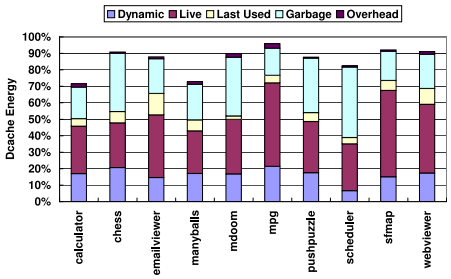

Figure 11: Data Cache energy of access gap analysis optimization. (The GC controlled scheme is also used.)

advantage of this scheme is that it has more chances to turn off cache lines. However, it also introduces extra cache misses since we may turn off the RCLs of some of objects that may be used in the near future. The cache miss rates of our benchmarks with access gap analysis optimization and the increase of execution time due to the extra misses are shown in the fifth and sixth columns of Table 3.

Our access gap analysis is also trace-based. Let us assume that bytecode $b1$ accesses object o at time t_1. Assume further that the next access to o is made by bytecode $b2$ at t_2. We define "the access gap associated with $b1$ " as $\Delta t = t_2 - t_1$. Note that if the access made by $b1$ is the last access to o, then $\Delta t = T - t_1$ where T is the overall execution time of the application. Based on the traces, we calculate the average length of access gaps associated with each bytecode. And then, we use our class file annotation tool to annotate each bytecode with the average length of the access gaps larger than the threshold G. The threshold G we used is based on the minimum duration required for leakage energy savings to amortize the penalties for the deactivate instruction and turn-on when the next access to the same object happens. During the course of execution, KVM marks the annotated bytecodes when the class file is loaded. Right after the execution of each marked bytecode, KVM immediately turns off the RCLs of the recently accessed object. The last column of Figure 4 gives the number of instructions that were marked in different applications.

Figure 11 shows the impact of this access gap analysis optimization when used in conjunction with the GC-control scheme. We achieve an average of 14% saving in the overall data cache energy consumption. Similar to the last use analysis, the access gap analysis may also be combined with the escape analysis and the GC-control scheme (denoted as "Escape + Access Gap" in Figure 12). With such a combination, the average saving of data cache overall energy is 21%.

Figure 12 shows how the different schemes compare with each other and how a combination of schemes can achieve significant energy savings.

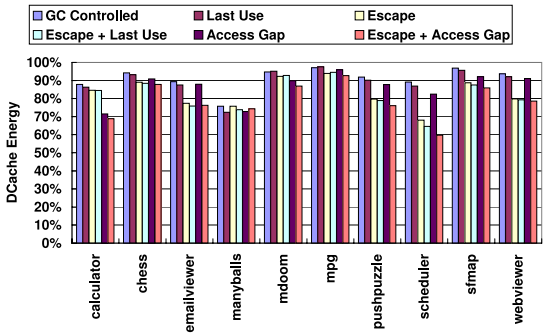

Figure 12: Comparison of optimization schemes. (All schemes use GC controlled approach as well.)

8. CONCLUSION

Leakage energy consumption is becoming an important concern for designers for future processors. This work specifically focuses on reducing the leakage energy in the data caches by exploiting the state of the object during its life span. Our analysis reveals that a major portion of leakage energy is wasted in retaining objects beyond their last use. Based on this observation, we explore the use of three different approaches that use the garbage collector, escape analysis and last-use analysis to reduce leakage energy. Finally, we also try to optimize the leakage energy consumed by the live objects by tracking their access interval. A combination of our schemes can reduce the data cache leakage energy by 21.3% on the average across different applications in our default configuration.

9. REFERENCES

[1] J. A. Butts and G. Sohi, "A static power model for architects," in *the 33th Annual International Symposium on Microarchitecture*, Dec. 2000.

[2] K. Flautner, N. Kim, S. Martin, D. Blaauw, and T. Mudge, "Drowsy caches: Simple techniques for reducing leakage power," in *the 29th International Symposium on Computer Architecture*, (Anchorage, AK, USA), May 2002.

[3] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache decay: Exploiting generational behavior to reduce cache leakage power," in *the 28th International Symposium on Computer Architecture*, (Sweden), June 2001.

[4] 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," in *the 35th Annual International Symposium on Microarchitecture (MICRO-35)*, (Istanbul, Turkey), 2002.

[5] H. Zhou, M. C. Toburen, E. Rotenberg, and T. M. Conte, "Adaptive mode control: A static-power-efficient cache design," in *International Conference on Parallel Architectures and Compilation Techniques (PACT'01)*, (Barcelona, Spain), Sept. 2001.

[6] G. Chen, R. Shetty, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and M. Wolczko, "Tuning garbage collection in an embedded java environment," in *the 8th International Symposium on High-Performance Computer Architecture (HPCA'02)*, (Cambridge, MA, USA), Feb. 2002.

[7] R. Riggs, A. Taivalsaari, and M. VandenBrink, *Programming Wireless Devices with the Java 2 Platform*. Addison Wesley, 2001.

[8] J. Lyman, "Java's surprising comeback." http://www.newsfactor.com/perl/story/18365.html.

[9] B. Cmelik and D. Keppel, "Shade: A fast instruction-set simulator for execution profiling," in *the 1994 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, pp. 128–137, May 1994.

[10] "CLDC and the K Virtual Machine (KVM)." http://java.sun.com/product/cldc.

[11] P. Shivakumar and N. P. Jouppi, "CACTI 3.0: An integrated cache timing, power, and area model," tech. rep., Compaq Computer Corporation Western Research Laboratory, 2001.

[12] J. Whaley and M. Rinard, "Compositional pointer and escape analysis for Java programs," in *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99)*, (Denver, CO, USA), Nov. 1999.

[13] J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff, "Escape analysis for java," in *the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 1–19, 1999.