Energy Optimization Using Object Co-Location in Java

S. Tomar, N. Vijaykrishnan, M. Kandemir, R. Shetty and M. J. Irwin
Department of Computer Science and Engineering
Pennsylvania State University
University Park, PA 16802

ABSTRACT

With the paradigm shift in computer systems towards ubiquitous computing, energy, together with performance, has become an important parameter to measure efficiency. Java is increasingly becoming the programming language of choice for applications expected to run in embedded and mobile environments. Java's platform independence and security features serve the needs of these environments very well, which expect the same application to run in a variety of environments in a secure manner. The devices used in these environments, for example hand-held computers, have a limited battery life. The needs to increase the period between recharging and decrease the cooling costs provide the incentive to use energy as a performance parameter.

This paper presents object co-location, an optimization method for Java applications. Object co-location exploits the temporal locality in heap references, to achieve better cache performance. This reduces the cache miss rate of programs, and subsequent reduction in memory energy consumption is observed due to fewer main memory accesses.

1. INTRODUCTION

Computing is no longer limited to desktop computers and servers alone, but it is becoming more and more pervasive with the emergence of mobile and embedded systems. Devices are becoming smarter, and it is the result of a well integrated hardware-software interface. The trend of connecting all these myriad devices together calls for an environment which facilitates application development and ensures cross-platform delivery.

Java [1] is widely regarded as a platform for the seamless integration of these myriad devices. It is becoming one of the programming languages of choice for networked and embedded environments. Java's platform independence and security features make it suitable for these environments. Java Virtual Machine (JVM) [2], the cornerstone of Java technology has made it possible. Java programs are translated to a machine independent format called bytecodes, and these bytecodes are subsequently executed by an implementation of the JVM for that device/hardware. These bytecodes can be interpreted, compiled at runtime (called Just In Time (JIT) compilation) or implemented completely in hardware. The JVM specification provides only the semantics of these bytecodes, implementation of the runtime environment is left to the designer of the virtual machine. The JVM is also responsible for the efficient execution of Java programs.

The phrase *"efficient execution"* no longer refers to performance efficiency alone, but energy has also come to be included in it. There are two main reasons why energy consumption has become an important performance parameter. First, the energy dissipated as heat increases the packaging cost required for cooling the device. The increased cost is particularly important in low-cost end products in mobile environments. Saving energy in battery-operated embedded devices is also motivated by need to increase operational periods of devices between battery recharges. Many techniques have been proposed at the circuit and architectural level for energy optimizations. But software, which runs on these hardware also needs to be energy efficient, as it is the primary factor which determines the dynamic switching activity (and hence dynamic power dissipation).

In this paper, we present object co-location, an optimization scheme for Java applications. Using static memory profiles of Java programs, we present a methodology to exploit inherent temporal locality in heap references. Our results show that intelligent placement of objects on the heap can improve cache performance and energy. We also suggest a possible implementation of the scheme using a modified garbage collector component of the JVM.

2. RELATED WORK

Lately, work has been done to characterize and optimize memory system energy of Java applications[24, 6, 7]. A detailed study of memory energy in the JVM was done in [5, 24]. [6] provides an annotation based scheme for allocating arrays on the heap in a way that reduces energy. [7] also provides an energy profile of applications running on a handheld device.

Improving the performance of dynamically allocated memory has been the subject of many research papers. Seidl and Zorn [12, 13] have investigated improving virtual memory performance by segregating objects based on lifetime, frequency of references and call sites. Following the genera-

tional hypothesis (most objects die young), and the fact that short-lived objects constitute a large portion of the objects allocated [12], a large number of objects can be allocated in a relatively small portion of the heap. This improves the spatial locality on the heap and leads to improved virtual memory performance.

Truong etal. [19] have suggested class field reorganization and instance interleaving as two data layout techniques for dynamically allocated data structures. Field reorganization groups more frequently referenced fields together in structure declaration so that they fit in the same cache line. Instance interleaving groups more frequently referenced fields of different instances of a data structure together. The argument behind this scheme is that identical fields in different instances of a data structure are often referenced together, and hence, they should remain in the cache together.

Chilimbi etal. [18] describes a generational garbage collection algorithm, in which objects with temporal locality are placed next to each other, so that they are likely to reside in the same cache block. Every load and store to heap data is profiled and based on this data, temporal relationships between objects are established. Objects with high temporal affinity are then placed next to each other on the heap during the next run of the garbage collector.

For Java, Chilimbi etal. [16] implemented a scheme called structure splitting. In this scheme, Java classes of instance size greater than a cache block are split into hot and cold portions depending upon which portion is used more frequently and vice versa, respectively.

Source code level changes have also been shown to improve performance [14, 15]. Liberal use of Java constructs leads to the creation of many objects and a high frequency of object-to-object copy operation [14]. These costly operations can be avoided by using techniques like object reuse, adequate object initialization and object and thread pooling [14].

3. MOTIVATION

Memory system can consume a large fraction of the total energy in Java applications and other embedded environments [24]. Hence, it is a good candidate for various hardware and software optimizations. For Java, heap is a very good candidate for memory optimizations, as a large fraction of all memory references go to the heap [4]. Further, it has been observed that most of the object references are confined to a small number of objects [17]. As a result, assigning random addresses to these objects can cause two problems in the cache. First, if two highly referenced objects are mapped to two addresses that conflict in the cache, then unless the cache is set associative, a significant number of conflict misses will occur. Second, the entries in a single cache line can be under utilized. Since the object addresses are assigned randomly, on a given cache line, only a small fraction of the line maybe assigned to a highly referenced object, while the rest of the line is essentially wasted because the objects that occupy it are infrequently referenced.

This observation leads us to conclude that assigning consecutive addresses to highly referenced objects can result in improvement in the cache miss rates. This is especially true for Java, because the average object size in Java applications is quite small, about 25-30 bytes [21]. The reason for this improvement is, that when a highly referenced object is fetched from the main memory on a cache miss, another highly referenced object is also fetched (because they are intelligently placed next to each other). Thus, there is good chance that when this object is referenced in future, it will be found in the cache and the penalty of a miss will be saved. The chances of this cache line being replaced are small, as the object adjacent to it is also frequently referenced. This reduction in cache miss rates translates into energy savings as accessing larger off-chip memories is much more expensive (in terms of energy) than accessing on-chip caches. In this paper, we experiment with an object co-location scheme, in which objects displaying temporal locality are co-located.

4. EXPERIMENTAL FRAMEWORK

In this section, we describe the framework we used for our experiments. We first give details of the JVM we used for running our Java applications. We then describe the benchmark applications and the cache simulator used for generating memory system information and profiling different components of the JVM. Our energy model is described after that.

4.1 Java Virtual Machine (JVM)

All our experiments were carried out using the Sun Labs Virtual Machine for Research (ExactVM (EVM)) [10], currently known as ResearchVM. EVM is designed to facilitate experimentation in memory management, especially soft real-time garbage collection. It is a high performance VM, which provides fast memory system, fast synchronization and a fast JIT compiler [10]. The EVM can execute Java programs in two modes: pure interpreter and an adaptive JIT mode. In the adaptive JIT mode, the EVM starts executing the application in the interpreter mode, and gathers *profiling information* regarding the runtime characteristics of the application. It then uses this information to dynamically compile certain methods. Specifically, methods with loops are compiled at the first invocation itself, while methods without loops are compiled when the invocation count reaches 15 [5].

4.2 Benchmarks

The benchmarks we used in our experiments are seven programs from the SPECjvm98 Benchmark Suite [3]. SPECjvm98 is an attempt to define an industry standard benchmark suite for Java programs. These programs are chosen by SPEC based on several criteria including high bytecode content, flat execution profile, repeatability, heap usage, and allocation rate [21]. Of the three input sizes of 1, 10, and 100, we have concentrated on the largest one (s100). Table 1 briefly describes each of the SPECjvm98 benchmarks. While the chosen applications are more likely to be executed in high-end (battery-operated) mobile devices like laptops, we believe the techniques presented in this work are also relevant to other applications typical of low-end mobile and embedded environments. We were constrained in our choice by the lack of a standard for embedded Java applications.

Cachesim5, which is a part of the Shade Tool Set [22], was used to obtain cache behavior statistics for benchmark applications. Shade simulates the execution of an application

Program	Description
db	Small data management program; performs database functions on a memory resident database
compress	Utility to compress/uncompress large files; makes five passes over the input
jack	A Java parser generator with lexical analysis; makes several passes over the same input
javac	The JDK 1.0.2 Java compiler
jess	Java expert system shell; based on NASA's CLIPS expert system
mpeg	MPEG-3 audio stream decoder
mtrt	Multi-threaded raytracer

Table 1: SPECjvm98 programs

and provides a programming interface that allows the user to collect arbitrary data while the application runs. Cachesim5 simulates a cache, traps each memory reference made by the application, and checks whether the reference will hit or miss in the cache. The information gathered at run time was fed to an analytical cache energy model [11] for our energy calculations. The model expresses cache energy in terms of the read and write port numbers, the number of registers, and several other relatively simple system and parameters for 0.35μ CMOS technology.

5. METHODOLOGY

Performance and energy results with objects co-located in memory were obtained in the following three steps:

Generating temporal information graph: In order to obtain information on how objects were temporally related, we generated a temporal information graph. This graph tries to relate all objects that are accessed within a specific window of references. To generate this graph, a program was written, which took object information generated through EVM, and memory reference trace as inputs. The program stepped through the memory trace of a benchmark, `n` references at a time (All experiments use $n = 100$). The objects in two adjacent windows (current and previous) of references are considered to have temporal locality and a node is assigned for each object in these windows and an edge is assigned between each pair of objects. To account for multiple references of an object within these adjacent windows, if an edge already exists the weight of the edge is incremented. This task is performed repeatedly for the subsequent windows until the entire memory trace is scanned. At the end of the entire scan of the memory trace, we had a temporal information graph, which depicted temporal affinity between objects. The higher the weight of the edge between two objects, the more closely (temporally) they were referenced throughout the program. The edges were then written to a file and sorted in decreasing order of weights. The generation of the temporal graph can be improved further by employing techniques such as sliding windows, and varying the size of the window, `n`. We are currently experimenting with alternate techniques for generating the temporal information graph.

Co-locating objects: The graph obtained in the first step was input to a program which co-located objects based on the information on temporal relationship contained in the graph. The program read edges of the graph in the decreasing order of weights.
Following algorithm (scheme `a`) was used by the program to place objects next to each other:

```
while(edges remain in candidate set) do
     select edge in the candidate set with the maximum
weight and remove it from candidate set;
if(neither object incident on the selected edge has
been relocated)
     relocate both objects;
else
     if(only one of the objects incident on the selected
edge has been relocated)
          relocate the other;
end while;
```

After determining the objects to relocate, the references belonging to the relocated objects were changed to reflect their new position. The trace with new references was then used as the input to the cache simulator to obtain cache statistics.

We also experimented with another allocation strategy. In this scheme, we identified the paths in the temporal information graphs that span all the nodes. The paths are selected to maximize the sum of the weights of edges across all the paths spanning the graph. All objects within the same path were then co-located.
The algorithm is presented below (we call it scheme `b`):

```
while(edges remain) do
     select edge in the candidate set with the maximum
weight;
if(neither node incident on the selected edge is found
in any existing paths)
     add this edge to a new path and remove the edge
from the candidate set;
else
     if((adding this edge does not cause a cycle) &&
       (does not cause any vertex to have degree > 2)
          add the edge to the appropriate path and remove
it from the candidate set;
     else
       remove the edge from the candidate set;
end while;
```

The paths selected were then used by our program to co-locate objects. The traces obtained using the modified co-location algorithm were then input to the cache simulator to generate cache statistics. The purpose of experimenting with allocation scheme `b` was to take more temporal relationships into account (and not just between two objects). Since both the schemes `a` and `b` provided comparable results, the results are presented only for scheme `a`.

6. ENERGY SAVINGS WITH CO-LOCATION

Table 2 shows energy savings with co-location for seven SPECjvm98 benchmarks. For a 16K, direct mapped cache, average energy reduction of 21.36% (averaged over seven benchmarks) is observed. Figure 1 shows the absolute energy reduction obtained with co-location for benchmark db. Larger savings are obtained for bigger block sizes. It is expected, as, with a bigger block size, more objects can be brought into the cache at a time. These objects have been clustered together because they exhibit temporal locality, hence, soon other objects will also be referenced, and those references will not cause the additional penalty of a main memory access. These reduced main memory accesses result in energy improvements.

Size	Assoc	javac	comp	jack	db	jess	mpeg	mtrt
16K	1	24.8	24.8	24.6	25.4	24.3	23.9	24.4
	2	21.5	21.4	21.6	21.6	21.5	20.8	21.0
	4	14.3	14.3	14.3	14.4	14.3	13.9	14.0
	8	39.6	39.6	39.6	39.6	39.6	39.5	39.6
32K	1	44.6	44.7	44.5	45.1	44.5	43.9	43.1
	2	30.8	20.8	21.2	20.9	20.7	20.4	19.8
	4	12.8	12.9	13.0	13.0	12.9	12.5	13.2
	8	7.6	6.7	7.6	7.7	7.7	7.4	7.6
64K	1	46.8	46.7	46.6	46.9	46.8	46.3	45.0
	2	48.8	48.9	48.9	48.8	48.9	48.4	48.7
	4	12.0	12.0	12.1	12.0	11.9	11.6	12.1
	8	6.7	6.7	6.8	6.7	6.7	6.5	6.9
16K	1	34.6	35.3	34.5	35.8	34.2	19.2	19.7
	2	15.1	15.2	15.2	15.2	15.1	14.3	15.5
	4	43.2	43.2	43.3	43.2	43.2	43.1	43.0
	8	8.2	8.2	8.3	8.2	8.2	8.4	8.3
32K	1	31.1	31.4	30.8	31.8	31.0	29.1	29.5
	2	14.4	14.6	14.7	14.6	14.3	13.8	14.0
	4	7.6	7.7	7.7	7.7	7.6	8.0	8.0
	8	6.5	6.5	6.5	6.6	6.6	6.7	6.4
64K	1	50.7	50.7	50.5	50.8	50.8	48.2	49.2
	2	16.3	16.4	16.3	16.3	16.3	15.4	16.4
	4	7.0	6.9	7.0	6.9	6.9	6.9	7.0
	8	5.7	5.8	5.8	5.7	5.7	5.5	5.8
16K	1	20.4	21.3	20.4	22.1	20.2	20.4	21.4
	2	40.9	40.9	41.0	41.0	40.9	43.5	43.8
	4	9.9	9.8	10.0	9.9	9.9	9.7	9.3
	8	9.5	9.5	9.5	9.5	9.5	9.2	9.2
32K	1	48.4	48.2	48.1	48.8	48.2	49.0	47.9
	2	9.7	9.8	9.9	9.9	9.7	8.0	8.3
	4	7.7	7.7	7.8	7.7	7.7	7.5	8.1
	8	6.4	6.4	6.4	6.4	6.4	6.3	6.0
64K	1	23.3	23.0	23.0	23.3	23.5	23.6	22.5
	2	10.5	10.5	10.5	10.6	10.5	15.6	10.8
	4	6.5	6.5	6.6	6.5	6.5	6.29	6.5
	8	45.3	45.2	45.2	45.2	45.2	45.3	45.3

Table 2: Percentage energy reduction with the co-location scheme as compared to similar configuration without co-location. For the three cache sizes of 16K, 32K, and 64K, values are shown for block sizes 32, 64, and 128, in that order.

Figure 2 shows the effect of block size on energy savings obtained with co-location scheme. In going from a block size

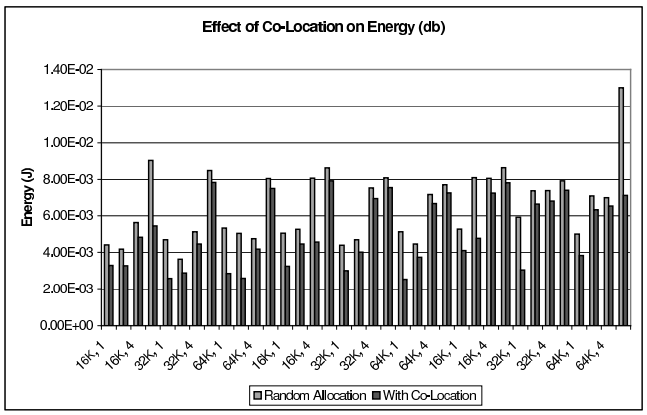

Figure 1: Absolute Energy values for db. Cache configuration are represented by the tuple (size in K, associativity). The results are presented for the different configurations for three block sizes 32, 64, and 128 bytes from left to right.

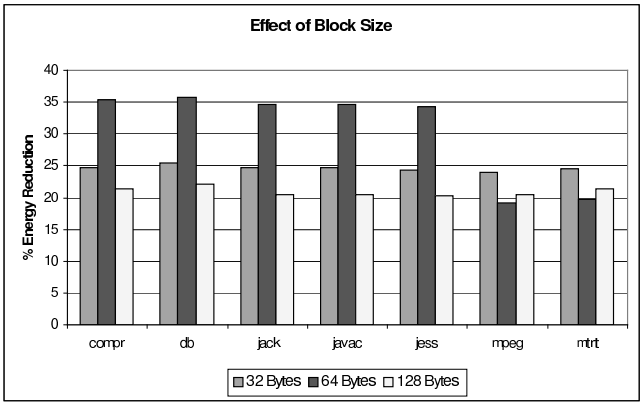

Figure 2: Energy savings for block sizes of 32, 64, and 128 bytes with the co-location scheme as compared to a similar configuration without colocation. Configuration is 16K, Direct-Mapped cache.

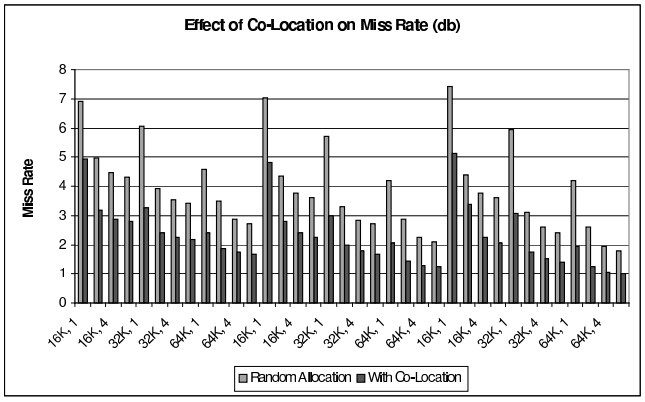

Figure 3: Miss rates with and without co-location for `db`. Cache configuration are represented by the tuple (size in K, associativity). The results are presented for the different configurations for three block sizes 32, 64, and 128 bytes from left to right.

of 64 bytes to 128 bytes, smaller improvements in energy are observed. The reason for this is lower reduction in the corresponding miss rates. Next subsection explains the reason why cache miss rates do not reduce as much as expected with co-location, in case of larger block sizes.

6.1 Cache Miss Rate Reduction

Reductions in cache miss rates obtained with co-location scheme are shown in Table 3. The results shows that significant improvements can be obtained from the co-location scheme in terms of cache performance. For a 16K, 2-way set associative cache, an average reduction of 35.4% is obtained. As can be observed, larger cache sizes of 32K and 64K get more benefits from the scheme.

Effect of block size: Figure 3 shows cache miss rates for the benchmark `db`. As would be expected, miss rates decrease with increasing associativity for a given cache size. Effect of block size can also be observed from this figure. Increasing the block size from 32 bytes to 64 bytes brings about larger benefits with co-location. This is because now, more objects which are temporally related can be fetched in one access. Because these objects are referenced relatively frequently with respect to each other, this results in larger benefits. This can be observed from Table 3 also.

Although the benefits from using larger block sizes can be seen with co-location, a general property of caches can be observed from the results: The reduction in cache miss rates for the random allocation case, and the co-location case individually, is smaller in case of block sizes of 64 and 128 bytes. This happens because the benefits obtained from bigger block size are somewhat offset by the fact that now a block takes up more memory in the cache, and there maybe more frequent replacements. It can be observed from figure 4 that the savings for block size 128 bytes are lower than those for block size 64 bytes.

Effect of associativity: Direct mapped caches gain more from the co-location scheme as compared with 2, 4, or 8-way set associative caches for both 32K and 64K caches

Size	Assoc	javac	comp	jack	db	jess	mpeg	mtrt
16K	1	27.9	27.8	27.7	28.7	27.3	26.8	27.5
	2	35.6	35.5	35.8	35.7	35.7	34.6	34.8
	4	35.1	35.0	35.2	35.4	35.1	34.8	34.7
	8	35.4	35.3	35.4	35.3	35.4	35.4	35.3
32K	1	45.7	45.8	45.4	46.4	45.4	42.7	43.3
	2	38.2	38.2	38.7	38.3	38.0	36.0	36.4
	4	36.4	36.6	36.7	36.7	36.6	37.0	37.3
	8	36.5	36.5	36.6	36.6	36.7	36.7	36.6
64K	1	47.3	47.0	46.8	47.3	47.3	42.3	43.1
	2	46.3	46.4	46.5	46.4	46.4	45.6	46.0
	4	39.1	39.1	39.4	39.1	38.9	39.1	39.5
	8	38.1	38.2	38.3	38.1	38.1	38.4	38.7
16K	1	29.6	30.6	29.4	31.3	29.0	29.7	30.5
	2	35.3	35.5	35.5	35.4	35.3	35.2	35.5
	4	35.9	36.0	36.1	36.0	35.8	35.3	35.1
	8	37.2	37.1	37.2	37.1	37.0	37.5	37.4
32K	1	47.0	47.3	46.7	47.9	46.8	44.6	45.1
	2	39.9	40.3	40.4	40.3	39.6	38.1	38.7
	4	37.8	38.0	38.0	38.0	37.8	39.0	39.3
	8	38.1	38.2	38.1	38.2	38.4	37.9	37.7
64K	1	51.1	51.0	50.5	51.3	51.1	46.7	47.4
	2	49.2	49.3	49.2	49.2	48.9	49.0	49.4
	4	42.0	41.9	42.2	41.9	41.7	41.9	42.3
	8	40.8	41.0	41.1	40.9	40.8	41.2	41.3
16K	1	28.9	29.9	28.8	31.1	28.5	28.7	30.1
	2	22.3	22.4	22.6	22.8	22.3	31.7	32.6
	4	39.8	39.7	40.1	40.0	39.7	37.9	38.0
	8	42.8	42.6	42.5	42.6	42.4	41.2	41.6
32K	1	47.8	47.6	47.3	48.6	47.6	46.6	47.0
	2	42.9	43.1	43.4	43.2	42.4	35.0	35.9
	4	41.3	41.4	41.5	41.5	41.3	42.4	42.7
	8	41.3	41.4	41.3	41.5	41.3	39.6	39.6
64K	1	53.5	52.8	52.8	53.4	53.6	51.3	51.5
	2	52.8	52.7	52.7	52.9	52.6	52.9	53.7
	4	44.9	44.9	45.1	44.9	44.6	44.5	45.0
	8	43.9	44.1	44.0	44.0	43.8	44.3	44.6

Table 3: Percentage reduction in miss rate due to co-location optimization. For the three cache sizes of 16K, 32K, and 64K, values are shown for block sizes 32, 64, and 128, in that order.

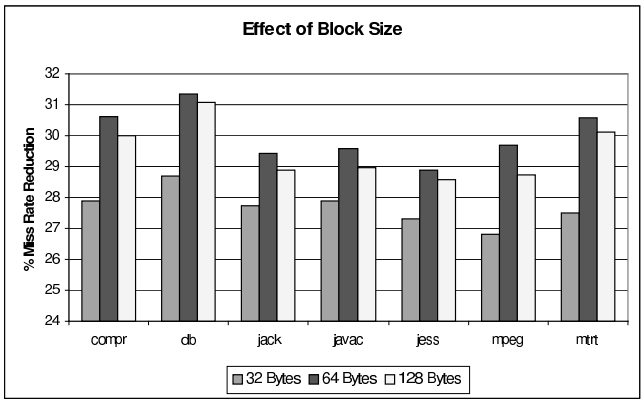

Figure 4: Effect of block size on cache miss rate reduction with co-location scheme. A 16K direct mapped cache is used.

in most cases. Higher associativity caches give lower miss rates even with random allocation. Direct mapped caches perform worse with random allocation because of the one-to-one mapping between main memory and cache addresses. Hence, direct mapped caches benefit more due to the conflict reduction after intelligent co-location of objects. This effect can be observed in Table 3.

7. DISCUSSION

Our results have shown that co-location can improve the performance of Java programs from performance and energy viewpoints alike. Our experiments were based on static memory profiles of SPECjvm98 benchmark applications. Therefore, we generated the temporal information graph based on this perfect knowledge about future. In practice, such information is not available while a program is running. Hence, we need some mechanism to obtain this information at runtime and provide it to the JVM. The JVM then needs to co-locate objects based on this information about memory references. In this section, we discuss how this information can be conveyed to the EVM, and present a possible solution.

Using garbage collector to co-locate objects: The garbage collector can be used to co-locate objects based on the temporal information made available to it. The EVM uses a compacting, generational garbage collector. If temporal information is available at garbage collection time, the collector can place candidate objects for co-location contiguously during the compaction phase.

We simulated the impact of such a runtime optimization by using our co-location method at each run of the garbage collector. A sentinel was inserted into the profiled memory reference trace every time the garbage collector was invoked by the virtual machine. This sentinel is used to create the temporal information graph whenever the garbage collector is invoked. Based on this temporal information, objects are co-located.

Table 5 shows the reduction in cache miss rate when this scheme we used. Corresponding reduction in energy values is shown in table 4. This simulated scheme performed as well as the scheme based on perfect knowledge. The applications in the SPECjvm benchmark suite are long running, and the garbage collector invocations are spaced apart by large number of memory (and heap) references. Hence, at each run of the collector, we have collected enough information about the past references to produce benefits from co-location.

8. CONCLUSION

Our experiments have shown that object co-location can significantly improve the cache miss rate of Java applications. We have gauged benefits from energy viewpoint also. In order to test a possible implementation of the scheme, we simulated co-location using the garbage collector to co-locate objects. Results obtained were comparable to the scheme based on perfect knowledge of all memory references. However, it must be noted that the creation of the temporal information graph and analysis will involve run time overheads. Our current study is on investigating low overhead techniques to translate the improved cache locality observed in this work to performance improvements in JVM implementations.

9. REFERENCES

[1] J. Gosling, B. Joy and G. Steele *The Java Language Specification*, Addison-Wesley, 1996.

[2] T. Lindholm and F. Yellin *The Java Virtual Machine Specification*, Addison-Wesley, 1997.

[3] Standard Performance Evaluation Corporation. SPECjvm98 Documentation, Release 1.0 August 1998.

[4] Jin-Soo Kim, Yarsun Hsu. Analyzing Memory Reference Traces of JAVA Programs. *Workshop on Workload Characterization*, October 1999.

[5] Anupama Murthy. Memory System Characterization of Java Applications. Master's Thesis, Dept. of Computer Science and Engineering, The Pennsylvania State University, May 2000.

[6] R. Athavale, N. Vijaykrishnan, M. Kandemir and M. J. Irwin. Influence of Array Allocation Mechanisms on Memory System Energy. To appear *International Parallel and Distributed Processing Symposium*, April 2001.

[7] J. Flinn, G. Back, J. Anderson, K. Farkas and D. Grunwald. Quantifying the energy consumption of a pocket computer and a Java Virtual Machine. Proceedings of *International Conference on Measurement and Modeling of Computer Systems.*

[8] L. Benini, A. Macii, E. Macii and M. Poncino. Increasing Energy Efficiency of Embedded Systems by Application-Specific Memory Hierarchy Generation. *IEEE Design and Test of Computers*, Volume 17(2), pp. 74-85, April-June 2000.

[9] DineroIII cache simulator. http://www.cs.wisc.edu/~larus/warts.html#Tycho.

[10] The Sun Labs Virtual Machine for Research. http://www.sun.com/research/java-topics/.

Size	Assoc	javac	comp	jack	db	jess	mtrt
16K	1	25.52	12.96	24.93	26.01	25.35	11.03
	2	22.50	21.53	22.80	22.57	22.80	21.96
	4	14.95	14.96	15.02	14.98	15.14	14.52
	8	40.07	40.05	40.09	40.07	40.15	39.90
32K	1	45.69	45.73	44.79	45.36	45.54	44.26
	2	21.71	20.54	21.99	21.62	21.80	20.42
	4	13.48	13.49	13.68	13.53	13.61	13.60
	8	8.03	7.99	8.08	7.99	8.12	7.85
64K	1	47.51	47.00	46.98	47.19	47.40	45.70
	2	49.24	49.09	49.20	49.10	49.29	48.93
	4	12.44	12.35	12.50	12.37	12.40	12.33
	8	6.99	6.93	7.04	6.94	7.00	6.99
16K	1	36.58	17.94	35.38	37.35	36.20	5.95
	2	11.96	11.38	12.25	12.01	12.23	12.43
	4	44.46	44.27	44.48	44.42	44.53	44.19
	8	10.24	10.12	10.26	10.18	10.28	10.22
32K	1	32.70	33.39	31.36	32.62	32.42	31.69
	2	10.76	10.05	10.93	10.74	10.78	10.41
	4	45.25	45.27	45.29	45.23	45.26	45.42
	8	8.55	8.44	8.59	8.53	8.62	8.31
64K	1	64.50	64.35	63.91	64.24	64.30	63.70
	2	16.27	16.31	16.19	16.15	16.31	16.19
	4	44.67	44.63	44.68	44.61	44.65	44.64
	8	4.83	4.78	4.89	4.79	4.84	4.83
16K	1	21.95	6.28	20.78	22.96	21.71	3.23
	2	41.46	44.00	41.64	41.47	41.60	44.42
	4	10.60	10.22	10.68	10.58	10.78	9.91
	8	10.36	10.12	10.41	10.31	10.42	9.98
32K	1	49.62	49.40	48.55	49.32	49.43	49.82
	2	47.51	45.71	47.59	47.40	47.47	45.92
	4	8.28	8.23	8.41	8.31	8.35	8.61
	8	7.25	7.05	7.30	7.27	7.34	6.74
64K	1	21.68	21.00	21.12	21.38	21.62	21.44
	2	10.72	10.68	10.68	10.71	10.72	10.92
	4	6.96	6.97	7.06	6.96	6.98	6.93
	8	45.66	45.63	45.68	45.65	45.66	45.66

Table 4: Percentage energy reduction when co-location was performed at each run of the garbage collector. For the three cache sizes of 16K, 32K, and 64K, values are shown for block sizes 32, 64, and 128, in that order.

Size	Assoc	javac	comp	jack	db	jess	mtrt
16K	1	28.82	12.81	28.07	29.42	28.61	10.33
	2	37.37	35.76	37.74	37.41	37.75	36.34
	4	36.69	36.69	36.78	36.70	37.04	35.88
	8	37.06	36.99	37.11	37.05	37.36	36.40
32K	1	47.32	47.39	45.94	46.80	47.09	45.19
	2	39.79	37.66	40.16	39.55	39.90	37.52
	4	38.15	38.11	38.54	38.20	38.41	38.28
	8	38.18	38.02	38.28	37.98	38.45	37.55
64K	1	48.79	47.64	47.59	48.07	48.54	44.71
	2	47.44	47.02	47.31	47.05	47.58	46.57
	4	40.50	40.24	40.60	40.28	40.43	40.23
	8	39.43	39.13	39.59	39.24	39.50	39.30
16K	1	31.27	31.56	30.15	32.66	30.70	30.21
	2	37.56	36.11	38.01	37.56	37.98	37.85
	4	37.80	37.05	37.91	37.63	38.09	36.72
	8	39.18	38.68	39.23	38.93	39.32	38.93
32K	1	49.39	50.09	47.67	48.93	48.96	48.49
	2	41.84	39.78	42.08	41.62	41.81	40.49
	4	39.92	40.01	40.14	39.83	40.00	40.83
	8	40.03	39.46	40.13	39.86	40.28	38.98
64K	1	52.60	52.28	51.18	51.99	52.20	50.06
	2	50.38	50.42	50.08	49.99	50.42	50.25
	4	43.55	43.32	43.62	43.20	43.45	43.37
	8	42.53	43.25	42.77	42.35	42.62	42.37
16K	1	30.75	8.76	29.13	31.92	30.41	4.57
	2	23.63	32.65	24.38	23.80	24.28	34.27
	4	41.06	39.57	41.25	40.92	41.54	38.68
	8	44.23	43.18	44.37	43.99	44.36	42.77
32K	1	50.28	49.86	48.19	49.67	49.91	50.66
	2	44.78	36.61	45.12	44.29	44.62	37.64
	4	42.86	42.45	43.26	42.80	42.98	43.71
	8	42.55	41.32	42.71	42.58	42.87	40.11
64K	1	54.99	53.35	53.52	54.11	54.45	54.32
	2	53.94	53.66	53.68	53.46	54.06	54.29
	4	45.85	45.81	46.18	45.71	45.91	45.67
	8	45.37	44.97	45.54	45.22	45.28	45.31

Table 5: Percentage reduction in miss rate when co-location was performed at each run of the garbage collector. For the three cache sizes of 16K, 32K, and 64K, values are shown for block sizes 32, 64, and 128, in that order.

[11] V. Zyuban and P.Kogge. The Energy Complexity of Register Files. *International Symposium on Low Power Electronics and Design*, pages 305-310, 1998.

[12] Matthew L. Seidl and Benjamin G. Zorn. Segregating heap objects by reference behavior and lifetime. Proceedings of *8th international conference on Architectural support for programming languages and operating systems*, October 1998.

[13] D. Barrett and B. Zorn. Using lifetime predictors to improve memory allocation performance. Proceedings of *SIGPLAN'93 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices 28(6), Albuquerque, NM, June 1993, ACM Press, pages 187-196.

[14] Reinhard Klemm. Practical Guidelines for Boosting Java Server Performance. Proceedings of *ACM 1999 conference on Java Grande*.

[15] Allan Heydon and Mark Najork. Performance limitations of the Java core libraries. Proceedings of *ACM 1999 conference on Java Grande*.

[16] Trishul M. Chilimbi, Bob Davidson and James R. Larus. Cache-conscious structure definition. Proceedings of *ACM SIGPLAN '99 conference on Programming language design and implementation*, May 1999.

[17] M. Siedl and B. Zorn. Low Cost Methods for Predicting Heap Object Behavior. Technical Report, Department of Computer Science, University of Colorado, Boulder, CO.

[18] Trishul M. Chilimbi and James R. Larus. Using generational garbage collection to implement cache-conscious data placement. Proceedings of *International symposium on Memory management*, October 1998.

[19] D. Truong, F. Bodin and A. Seznec. Improving Cache Behavior of Dynamically Allocated Data Structures. *International Conference on Parallel Architectures and Compilation Techniques*, October 1998.

[20] Bill Venners. Inside the Java Virtual Machine, Second Edition. McGraw-Hill, 1999.

[21] Sylvia Dieckmann and Urs Holzle. A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks. Proceedings of *European Conference on Object-Oriented Programming*, June 1999.

[22] Bob Cmelik and David Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. Proceedings of *ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, pages 128-137, May 1994.

[23] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, W. Ye and H. Kim. Energy-driven integrated hardware-software optimizations using SimplePower. Proceedings of *International Symposium on Computer Architecture*, June 2000.

[24] N. Vijaykrishnan et. al. Energy Behavior of Java Applications from the Memory Perspective. (To appear) Proceedings of *USENIX JVM Research and Technology Symposium*, April 2001.