Using Scratchpad to Exploit Object Locality in Java

Carl S. Lebsack and J. Morris Chang
Department of Electrical and Computer Engineering
Iowa State University
Ames, IA 50011
lebsack, morris@iastate.edu

Abstract

Performance of modern computers is tied closely to the effective use of cache because of the continually increasing speed discrepancy between processors and main memory. We demonstrate that generational garbage collection employed by a system with cache and scratchpad memory can take advantage of the locality of small short-lived objects in Java and reduce memory traffic by as much as 20% when compared to a cache-only configuration. Converting half of the cache to scratchpad can be more effective at reducing memory traffic than doubling or even quadrupling the size of the cache for several of the applications in SPECjvm98.

1. Introduction

The speed gap between processors and main memory will continue to widen. We are already seeing significant impacts of this trend. It was recently reported that Java applications can spend as much as 45% of their execution time waiting for main memory [1]. Although modern cache designs are becoming increasingly large, the costly overhead of miss penalties can still lead to significant performance degradation. Alleviating the memory system bottleneck by reducing memory traffic is the motivation for this research.

Before describing the method of reducing memory traffic we present some important background information. The first key detail we present is that most objects in Java programs are small. Five of the programs in SPECjvm98 have an average object size below 32 bytes. Three of these programs have an average object size below 24 bytes [5]. However cache lines are typically 32 bytes or larger. Therefore most cache lines containing objects will contain more than one object. It is also quite likely that many small objects could be located in two cache lines. Given two contiguous 24 byte objects that start at the beginning of a 32 byte cache line, the second object will reside in two cache lines.

The second key detail we present is that most objects in Java are short-lived. Of the objects allocated in SPECjvm98 applications, 60% or more have a lifetime less than 100KB of allocation [5]. Cache however has no concept of lifetime and considers all modified cache lines to have pertinent data that must be written back to main memory. This means that even dead objects that will never be accessed in the future will consume bandwidth when written back to memory.

The combination of these behaviors creates a system environment with excessive memory traffic. Given a large heap area and aggressive allocation, a churning effect can occur in which continually dying objects are written back to memory with some of the longer-lived objects. Subsequent accesses to the live objects will cause the dead objects to be reread from memory. Depending on the delay between garbage collections, these lines could be read from and written to memory several times. A cache with a write-allocate policy (a write cache miss to a line that will not be completely overwritten will cause the line to first be read from memory) will reread dead objects from memory before allocating new live objects in their place. Even two live objects colocated in a single cache line will not necessarily be accessed together (highly dependent on allocation policy) and thus accesses to one may unnecessarily retrieve the other from memory.

The real problem is that there is no natural mapping between objects and cache lines. Thus there is no obvious correlation between cache locality and object locality. By having multiple objects within the same cache line an artificial interdependence is created among these objects. The same is true of multiple cache lines that are occupied by the same object.

To break the size discrepancy and remove the requirement that all modified contents be written back to memory we add another memory component to the hierarchy that doesn't follow the traditional cache model. Instead we investigate an alternate memory scheme that has no arbitrary subdivisions and generates no traffic to memory on its own. This memory region will be on-chip along with the original

Proceedings of the 2005 International Conference on Computer Design (ICCD'05)
0-7695-2451-6/05 $20.00 © 2005 IEEE

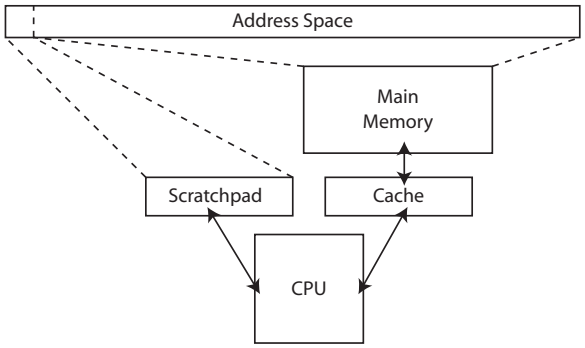

Figure 1. System with Scratchpad

cache. This type of memory region is often referred to as a scratchpad. It is a software managed memory that is itself a subset of the address space distinct and disjoint from that of the rest of the memory system as shown in Figure 1. Anything located in the scratchpad will not be located in main memory, and vice versa, unless the software explicitly makes a copy.

Some commercially available embedded processors contain scratchpad. We will be evaluating the use of scratchpad in a more general case and we are not restricting ourselves to sizes found in commercially available devices. Scratchpad has been shown to be more efficient in terms of area and power and also has a lower access time than a cache organization of equivalent size [2]. These benefits come from the fact that scratchpad does not need extra resources for cache line tags and does not need to first evaluate a tag to ensure the data is valid. In our work we disregard power and latency benefits of scratchpad and focus solely on the ability of scratchpad to reduce memory traffic. We will make our evaluation of efficient scratchpad use by comparing against a cache that has an equivalent data capacity.

Why should scratchpad provide any benefits with regard to memory traffic? First there are no internal boundaries that could interact with object boundaries. Any object located within the scratchpad will be entirely within the scratchpad, and objects next to one another will have no artificial interdependence due to the scratchpad. Second, a dead object within the scratchpad will not be written to main memory without an explicit software command to do so. Thus the goal of employing scratchpad is to ensure that objects allocated within it are created, accessed, allowed to die, and then reclaimed without moving back and forth to main memory. Once objects are reclaimed, the space can be reused for new allocations.

Given this possible solution there is one important research question that must be answered: Can we verify that a system using cache and scratchpad can outperform a standard cache-only system if both systems have equivalent on-chip storage capacity?

In our preliminary work we found that small short-lived objects have the highest locality and are most suited to mapping in scratchpad. The nursery of a generational garbage collector is a natural heap subregion that segregates small short-lived objects. By mapping this heap region to scratchpad we take advantage of the locality of the small short-lived objects. In this work we show that a system with cache and scratchpad can reduce memory traffic by as much as 20% over that of a cache-only system. In fact, for many programs it is more efficient to divide on-chip resources into scratchpad and cache than it is to double or even quadruple the size of the cache.

The rest of this paper will focus on answering the above research question in detail. Section 2 contains a description of the various tools used throughout the experimentation. Section 3 describes the experiments in detail along with their results and an interpretation in relation to our research question. Section 4 provides a discussion of related research in the context of our work. Section 5 concludes this work.

2 EXPERIMENTAL SETUP

A diverse set of tools was needed to perform the experiments in this research. We chose to use the applications from the SPECjvm98 [17] benchmark suite. SPECjvm98 has three different input sizes (s1, s10, and s100) which correlate to three different runtimes for each application. Smaller sized inputs are useful when the applications are run on a full system simulator due to the massive time overhead incurred in the simulation environment.

The virtual machine we use in our experiments is SableVM [11]. For these experiments we wanted to investigate generational garbage collection, which was not available in SableVM. We were able to build a fixed size nursery generational collector based on the original collector. Our collector uses remembered sets and promotes all live objects from the nursery on a local collection. Our implementation works for all of the SPECjvm98 benchmarks as well as all the other applications written to verify garbage collector functionality. The collector implementation was made publicly available in the source distribution of SableVM.

To trace all memory accesses initiated by an application we employ bochs [14], a functional emulator of the x86 architecture. Bochs is an open source project that allows a full operating system to be installed within the simulated environment. By emulating the x86 architecture, bochs is capable of running the same binary executables compiled for our Intel Pentium IV systems. Bochs also provides a basic set of stubs for instrumentation of various simulator behaviors including the stream of instructions executed and the stream of reads and writes initiated.

We chose DineroIV [10] as the cache simulator for our

work. It has a very straightforward interface and allows easy customization of cache configuration.

Bochs was configured to supply memory access traces of SableVM running SPECjvm98 applications. Traces were subsequently run through the DineroIV cache simulator. For all caches in our simulations, regardless of size, we selected 8-way associativity and a 64 byte cache line size.

3 Results

We previously identified that small short-lived objects have the highest locality and employ the nursery of a generational garbage collector to capture these objects. These experiments evaluate the mapping of the nursery of a generational garbage collector to scratchpad.

First, we must ensure that the nursery of a generational collector is not made too small. If objects are not given enough time to die then they will not be reclaimed within the nursery but copied into the mature space. Not only would this prohibit reclamation within the scratchpad, hurting potential memory traffic reduction, but it also requires additional copying which is expensive and leads to excessive collections of the mature space. We also need to make sure that the nursery does not need to be so large that the resulting scratchpad would be unreasonably large.

The following experimental results demonstrate that there are two opposing trends that are important when considering the costs of a copying collector. The tradeoff between these trends is that increased copying can lead to better locality. The opposite is also true. In our experiments we have chosen to stay focused on a fixed-sized nursery generational collector implemented in SableVM.

The first important observation is that the high mortality rate of objects can be taken advantage of with a relatively small nursery. Figure 2 shows the amount of copying from the nursery for several SPECjvm98 benchmarks over nursery sizes ranging in size from 32KB to 2MB in successive powers of two. The copying has been normalized to the minimum, which appears in the nursery size of 2MB. The results of *compress* and *mpegaudio* have been excluded as they allocate so little in the nursery.

As the nursery size is continually increased, we get a diminishing return on the reduction of copying. In fact if we were to ignore the absurdly small nursery sizes (32KB-128KB), the largest variation in copying is only about 25% for a four-fold increase in nursery size. While providing a larger nursery will indeed allow more objects to die, it does so at the expense of using more address space for allocation. Since most objects are short-lived, providing more time than a majority of the objects need will not reduce copying significantly.

A minor observation to be made from Figure 2 is that the behavior for each of the benchmarks is relatively consistent

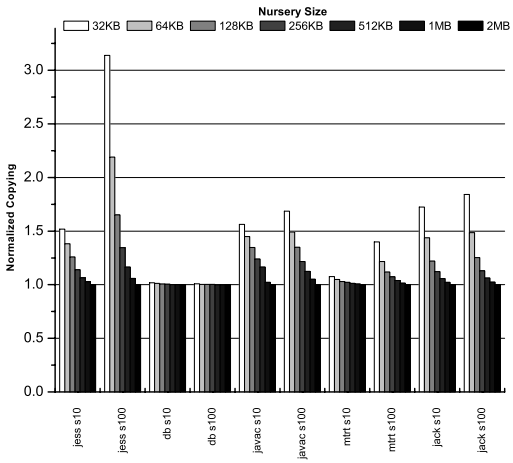

Figure 2. Copying vs. Nursery Size

between problem sizes, s10 and s100. This consistency is important as subsequent results are based on the s10 problem size because of simulation overhead when generating access traces.

The next experiment shows that a larger nursery also puts greater strain on the cache. Figure 3 shows the results in terms of normalized memory traffic caused by cache misses when executing the benchmarks over the nursery sizes from 128KB to 2MB for a cache size of 512KB. As the nursery increases, there is an increase in the traffic to main memory because of increased cache misses.

The results of these experiments show that as long as the nursery is not made too small (less than 256KB) we should not expect to see poor performance from the generational collector, and as the nursery is continually increased, we should not see a drastic change in that performance. As we are working in the range of a reasonable cache size, mapping the nursery of a generational collector to scratchpad could be an efficient method of capturing the locality of the small short-lived objects in Java programs.

Next we map the nursery of GGC directly to the scratchpad. Having the nursery smaller than scratchpad makes little sense. This arrangement would place some other arbitrary memory region in part of the scratchpad which would be an inefficient use of scratchpad. Making the nursery larger than the scratchpad places some of the short-lived objects, identified to have high locality, outside of scratchpad thus reducing potential benefit. For all configurations using scratchpad, we use a nursery of equal size.

The final step in evaluating our proposed memory configuration (Figure 1) is to determine if employing scratchpad can be more effective than simply using a larger cache. Although we mentioned above that scratchpad has other

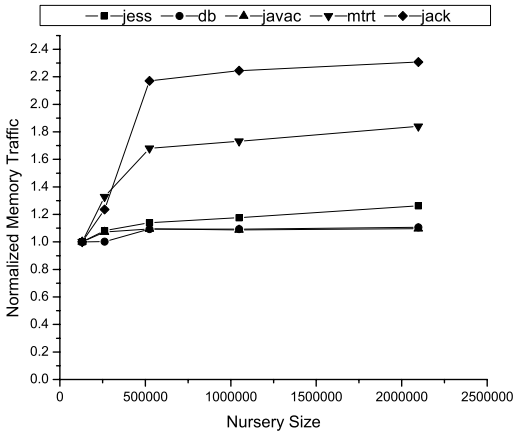

Figure 3. Memory Traffic vs. Nursery Size

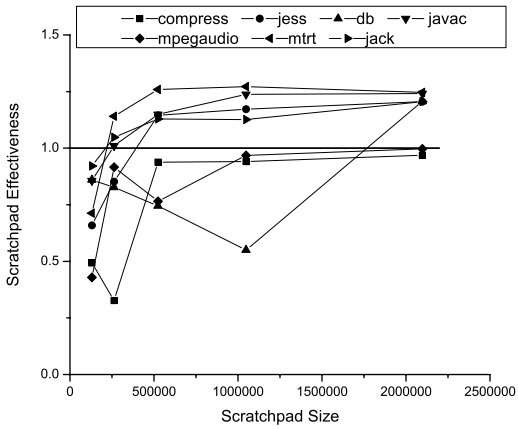

Figure 4. Scratchpad effectiveness

benefits over cache, our main goal is to reduce memory traffic. In order to do so, we have opted for a software managed exploitation of locality based on the findings that small short-lived objects exhibit a high locality. By employing generational garbage collection to confine these objects to the scratchpad through the one-to-one mapping of the nursery, we expect to gain an advantage over a traditional cache.

To make a comparison we tested a series of on-chip memory sizes in which we compare two configurations, a configuration in which the cache and scratchpad are the same size versus a configuration with cache equal in size to the combination of both scratchpad and cache. The plots in Figure 4 show the memory traffic for the scratchpad configuration normalized to that of the cache-only configuration. Therefore the scratchpad configuration is more effective for any point that falls above the unity line, and cache is more effective for any point that falls below the line.

As Figure 4 shows, the scratchpad configuration is more effective than the cache-only configuration for most of the benchmarks for scratchpad sizes greater than 256KB. The three benchmarks that do not perform as well with scratchpad are *compress*, *db* and *mpegaudio*, the three applications we expected to benefit the least. Both *compress* and *mpegaudio* perform almost identically with cache and scratchpad for sizes greater than 512KB.

Both *compress* and *mpegaudio* exercise garbage collection very little by allocating only a few thousand objects as opposed to the millions allocated in the other applications, and therefore we can do little to positively affect their behavior by building a memory management strategy on this algorithm. Specifically, *compress* allocates most of its memory as large objects which never appear in the nursery, and therefore must always appear in the cache.

On the other hand, *mpegaudio* allocates so little overall that it's residency is very small. It does allocate enough for garbage collection to promote some of it's objects to the mature space, leaving very few objects in the nursery to consume accesses. We did test an additional configuration for *mpegaudio* with a 4MB cache and 4MB scratchpad against an 8MB cache. In this configuration nearly all of the objects remain in the nursery allowing *mpegaudio* can perform better on the system with scratchpad. However this size of on-chip resources is unreasonably large for this program as it has so little allocation overall. Both *mpegaudio* and *compress* are often left out of garbage collection research entirely.

The last benchmark, *db*, actually does the worst for any single configuration. This application is extremely sensitive to the size of scratchpad selected as it has a very large percentage of long-lived objects. Our experiments show that even *db* can show an improvement of nearly 17% with a 2MB scratchpad. As the size of on-chip resources is increased the sensitivity of the applications which do not see the benefits of scratchpad are minimized while those that benefit continue to do so.

In addition to showing that a cache and scratchpad system can outperform a cache-only system there is an additional important observation. Not only does the cache scratchpad system outperform a cache-only system of equivalent data capacity, but it can be more effective than doubling or quadrupling the capacity of the cache-only system. Note the total traffic in Table 1 for the 512KB cache and 512KB scratchpad versus the 4MB cache for the benchmarks *jess*, *jack* and *mtrt*. Both *jess* and *mtrt* perform better with 1MB of on-chip cache/scratchpad than 4MB of cache while *jack* performs very similarly for both. Even *javac* performs better with 1MB of cache/scratchpad than it does with 2MB of cache. As cache is becoming one of the largest

Table 1. Bytes transferred between cache and memory in cache-only and cache/scratchpad systems for three on-chip storage capacities.

1MB	Cache	Cache/Scratchpad	
	Total Traffic	Total Traffic	% Improve
compress	116,233,216	123,885,632	-6.58
jess	44,492,608	38,845,824	12.69
db	96,537,472	129,597,056	-34.25
javac	78,061,888	67,900,096	13.02
mpegaudio	37,942,784	49,590,208	-30.70
mtrt	77,810,688	61,785,280	20.60
jack	43,268,224	38,318,656	11.44
2MB	**Cache**	**Cache/Scratchpad**	
	Total Traffic	Total Traffic	% Improve
compress	107,152,320	113,858,432	-6.26
jess	41,975,936	35,814,144	14.68
db	47,247,872	85,878,976	-81.76
javac	64,042,240	51,753,472	19.19
mpegaudio	33,772,608	34,899,072	-3.34
mtrt	67,428,288	53,028,352	21.36
jack	37,552,832	33,352,768	11.18
4MB	**Cache**	**Cache/Scratchpad**	
	Total Traffic	Total Traffic	% Improve
compress	101,638,720	104,939,904	-3.25
jess	41,791,872	34,657,600	17.07
db	46,984,000	38,938,048	17.12
javac	60,309,888	48,575,616	19.46
mpegaudio	29,898,240	29,979,200	-0.27
mtrt	68,842,304	55,233,408	19.77
jack	38,191,872	31,671,680	17.07

consumers of die area in modern processors, this finding that scratchpad can be a more effective addition at reducing memory traffic than a significantly larger cache is very important.

4 RELATED WORK

To the best of our knowledge our work is the first to focus on reducing memory bandwidth consumption for Java applications in a general computing environment. Since this work covers a broad range of topics including Java application behavior, garbage collection, caching strategies and hardware software interaction there is a very large base of related work to consider.

The work most closely related is that of Kim and Tomar et. al. in which they evaluated the use of local memory (scratchpad) to increase the performance of embedded Java applications [13, 18]. They work with memory configuration sizes found in common embedded processors, which is on the order of a few kilobytes. They identify large long-lived objects with a large number of accesses through static profiling and alter the bytecode stream of the application to allocate these objects in the scratchpad for the duration of the program. They also evaluated the use of garbage collection to colocate long-lived objects in the scratchpad. Their goal was to increase the performance of embedded applications running on a system that had scratchpad over the same system if scratchpad were left unused.

Many researchers have noted the relationship between garbage collection and cache performance. Some have studied ways to improve program performance by employing garbage collection to increase locality [16, 4, 9, 15, 20, 19]. Blackburn et. al. performed a comprehensive study of garbage collection techniques in relation to cache performance and found that a generational collector employing a nursery provided the highest locality [3].

Another body of work that is related to ours is that investigating the importance of memory bandwidth. Although we believe we are the first to focus on reducing memory traffic and thus the burden on the available bandwidth in a system for Java applications, there are other works that also address this problem in the more general case [6, 7, 8, 12].

Hiding the latency of the memory system has also been of interest to many researchers and has been most often attempted through prefetching techniques. Prefetching has been investigated specifically for Java [1]. It is important to note that although prefetching is not directly related to our work, it does place a greater burden on the memory bandwidth and thus research in this area could often be complementary to ours.

5 CONCLUSIONS

In this work we demonstrate a comprehensive system design that significantly reduces memory traffic for many Java applications without significantly impacting those applications which do not benefit from our design. We show the process of our work by answering our key research question: Can we verify that a system using cache and scratchpad can outperform a standard cache-only system if both systems have equivalent on-chip storage capacity?

Our memory traffic results confirm that a system with cache and scratchpad can significantly reduce memory traffic (both inbound and outbound) over a cache-only system. For some configurations many programs see a near 20% reduction in total memory traffic. While this alone is significant it is also important to note that for applications that get the greatest benefit, it can be more efficient to divide on-chip resources into scratchpad and cache than to increase the size of the cache 2 to 4 times. The results of this work provide incentive to further investigate hardware modifications to the memory hierarchy to more efficiently support object oriented languages such as Java.

6 ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 0296131 (ITR) 0219870 (ITR) and 0098235. Any opinions, findings, and

conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] A.-R. Adl-Tabatabai, R. L. Hudson, M. J. Serrano, and S. Subramoney. Prefetch inection based on hardware monitoring and object metadata. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 267–276. ACM Press, 2004.

[2] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: a design alternative for cache on-chip memory in embedded systems. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, pages 73–78, 2002.

[3] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: the performance impact of garbage collection. In *Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 25–36. ACM Press, 2004.

[4] T. M. Chilimbi and J. R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *Proceedings of International Symposium on Memory Management (ISMM)*, October 1998.

[5] S. Dieckmann and U. Hölzle. A study of the allocation behavior of the specjvm98 java benchmarks. In *European Conference on Object-Oriented Programming (ECOOP 99)*, August 1999.

[6] C. Ding and K. Kenedy. Bandwidth-based performance tuning and prediction. In *Proceedings of the International Conference on Parallel and Distributed Computing Systems*, 1999.

[7] C. Ding and K. Kenedy. The memory bandwidth bottleneck and its amelioration by a compiler. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2000.

[8] C. Ding and K. Kenedy. Improving effective bandwidth through compiler enhancement of global cache reuse. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2001.

[9] A. Diwan, D. Tarditi, and E. Moss. Memory system performance of programs with intensive heap allocation. *ACM Trans. Comput. Syst.*, 13(3):244–273, 1995.

[10] J. Edler and M. D. Hill. *Dinero IV Trace-Driven Uniprocessor Cache Simulator*. http://www.cs.wisc.edu/ markhill/DineroIV.

[11] E. Gagnon. *SableVM*. http://sablevm.org.

[12] E. G. Hallnor and S. K. Reinhardt. A unified compressed memory hierarchy. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, 2005.

[13] S. Kim, S. Tomar, N. Vijaykrishnan, M. Kandemir, and M. Irwin. Energy-efficient Java execution using local memory and object co-location. In *IEE Proceedings-Computers and Digital Techniques*, 2004.

[14] K. Lawton, B. Denney, G. Alexander, T. Fries, D. Becker, and T. Butler. *bochs: The cross-platform IA-32 emulator*. http://bochs.sourceforge.net.

[15] M. B. Reinhold. Cache performance of garbage-collected programs. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 206–217. ACM Press, 1994.

[16] Y. Shuf, M. Gupta, R. Bordawekar, and J. P. Singh. Exploiting prolific types for memory management and optimizations. In *Proceedings of the Symposium on Principles of Programming Languages*, Portland, Oregon, 2002.

[17] Standard Performance Evaluation Corporation, http://www.spec.org/jvm98. *SPECjvm98 Benchmarks*.

[18] S. Tomar, S. Kim, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. Use of local memory for efficient Java execution. In *Proceedings of IEEE International Conference on Computer Design (ICCD)*, Austin, Texas, 2001.

[19] P. R. Wilson, M. S. Lam, and T. G. Moher. Effective static-graph reorganization to improve locality in garbage-collected systems. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 177–191. ACM Press, 1991.

[20] P. R. Wilson, M. S. Lam, and T. G. Moher. Caching considerations for generational garbage collection. In *Proceedings of the 1992 ACM conference on LISP and functional programming*, pages 32–42. ACM Press, 1992.