2008

Performance analysis and optimization of the Java memory system

Carl Stephen Lebsack
Iowa State University

Performance analysis and optimization of the Java memory system

by

Carl Stephen Lebsack

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Program of Study Committee:
J. Morris Chang, Major Professor
Arun K. Somani
Akhilesh Tyagi
Zhao Zhang
Donna S. Kienzler

Iowa State University

Ames, Iowa

2008

DEDICATION

I would first like to dedicate this thesis to God by whom all things were made and in whom they are held together. Only by His grace and mercy through the gift of His Son, Jesus Christ, was I afforded the opportunity to live and work and play.

I would also like to dedicate this thesis to my parents, Stephen and Debra Lebsack, whose support was pivotal in the completion of this work. I want to thank my siblings, Mary and her husband, Aaron Hudlemeyer, David and Laurel, and the rest of my family for their unwavering belief in my ability to complete PhD study and their continual love and support during my unbelief. I also thank Mary for her help in proofing and editing the thesis text. I, am, terrible, with, commas. Laurel deserves an extra thanks for enduring the front lines of battle while living with me during her own transition into college. I would also like to thank my fiancee, Tegwin Taylor, who, although she came along near the end of my study, was extremely uplifting and provided significant motivation to finish at long last.

Thanks also goes out to the rest of my friends and family for their continued encouragement and prayers. I could not have done this alone.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1 Introduction

Java is one of the most dominant languages used in development today (75). It can be found in desktop, server, embedded and portable computer systems. The language has many desirable characteristics that have made it popular including object-oriented design, automatic dynamic memory management (garbage collection), platform independence, security and a rich library of components available for rapidly constructing complex software. Product developers are drawn to the robustness afforded by the various language features and they appreciate a quick time-to-market achieved with a lower incidence of software defects than previous language systems offered.

Before narrowing the discussion to the work presented in this thesis, this chapter will provide a high level context for the research by describing the Java system. The benefits that Java provides come at a cost. To provide the rich level of support to the language that makes it robust and portable an additional layer of software is added between the host operating system and the applications. This layer is the Java Virtual Machine or JVM. Figure 1.1 shows the layout of a computer system with respect to the layers of hardware and software to support Java applications and applications written in traditional languages such as C. This additional layer provides not only a new layer of abstraction, but also constitutes software that introduces performance overhead in the system.

The JVM is only one component of the Java language system as a whole. Programs written in the Java language are first converted by a Java compiler from their human readable source representation into an intermediate form referred to as Java bytecode. It is the intermediate bytecode form that comprises a platform-independent Java application which is executed by a JVM. Portability, whereby any JVM implementation is capable of running the bytecode, is

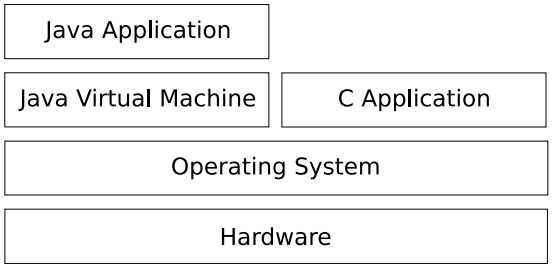

Figure 1.1 Java System

provided at the cost of the code not being optimized for any specific architecture.

Figure 1.2 shows the relationship between the Java compiler, often a Java application itself, and the source code which is converted to application code. The class library is a repository of pre-compiled Java classes that can be linked dynamically by an application at runtime by the JVM and need not be distributed with the application. The Java system also provides an interface to execute native code (code written in other languages and compiled to a specific architecture) through the Java Native Interface.

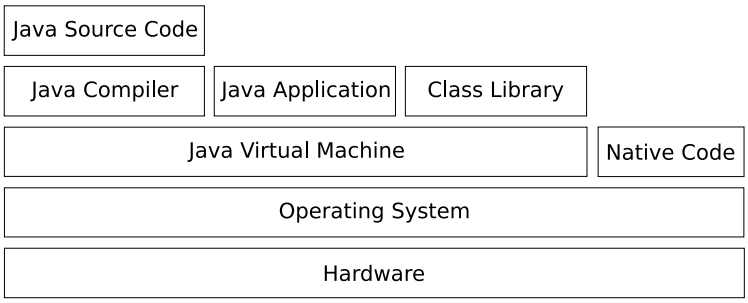

Figure 1.2 Full Java System

The JVM is the component of the Java system that is the focus of this thesis. It is a large complex piece of software that is platform specific. The binary for a particular JVM will run only on the system and architecture for which it was compiled. The complexity arises from many subsystems that work together to provide the runtime features of the Java language. The JVM is much like an operating system in that it manages resources and provides applications an interface to many commonly used features. Some of the features include thread scheduling,

memory management, dynamic code compilation and synchronization. It is the JVM that participates directly in the runtime performance of Java applications.

The two most vital components with respect to the performance of Java applications include Just-in-Time compilation (JIT) and Garbage Collection. The purpose of JIT is to convert the intermediate bytecode presented by the compiler to native code for host architecture. This process maintains the portability of the bytecode for distribution across platforms while allowing the applications to run at speeds near that of code compiled specifically for a particular architecture. The main difference between direct compilation and the dynamic bytecode compilation of JIT is that JIT occurs at runtime and the compilation is itself a runtime overhead. The speed increase generally substantially outweighs the overhead. While JIT is an active area of research, it is quite mature. Nearly every commercially available JVM as well as many research JVM implementations incorporate the technology for its performance benefits. JIT is not the focus of this research, but it does play an active role in performance.

Garbage collection is the other major subsystem that affects the performance of Java applications. The purpose of garbage collection is to avoid explicit memory management by the programmer. Memory management is one of the most difficult tasks to complete when writing software, and is a source for many bugs. The most notable bug is the memory leak which occurs when a program continually requests memory without ever returning it to the system. Eventually, memory resources are exhausted and the program or system crashes. Garbage collection helps to avoid such a situation by identifying which memory is no longer accessible to the program and automatically reclaiming it for reuse. Garbage collection also introduces a runtime overhead on a Java application as processor and memory resources are used to periodically scan the heap for memory that can be reclaimed.

There are many different garbage collection algorithms to choose from and a good overview is provided by Jones and Lins (40). However, research and practice have settled on the generational algorithm or one of its variants as the algorithm of choice. Sun's JDK (71), BEA's JRockit (10) and Jikes RVM (39) all rely on a version of generational garbage collection as the default collection algorithm. Matthew Hertz et al. showed that given on average five and

a half times the minimum memory resources needed to run, generational garbage collection can yield performance near that of explicit memory management (without the risk for memory leaks) (35). Stephen Blackburn et al. showed that selecting a collector with a generational style nursery was the most important factor with respect to performance because of the locality impact on cache performance (11).

This thesis focuses on memory system performance of Java applications. Specifically we analyze the behavior of the applications in terms of their dynamic memory allocation and access patterns. Additionally, we analyze properties of the whole system performance including runtime and cache misses. Experiments are conducted under various simulation environments as well as on real hardware.

During the course of this work, several different JVM implementations were studied, both open and closed source as well as research platforms and commercial products. Despite the wide variety of platforms and evaluation techniques, the core objective of this work is to evaluate the behavior and performance of the memory subsystem with respect to the entire system from the Java application to the hardware. An important point regarding the work in this thesis is that the state of the art has continued to progress. While this could go without saying, it is the reason that so many evaluation techniques have been employed. It is also the reason that the benchmarks used between studies are not consistent. Prior to the release of the DaCapo suite in October of 2006 (12), the most standard and commonly used benchmark suite was SPECjvm98 (69), which is still in use. Chapters in this work which exclude the DaCapo suite were completed before its release.

The remainder of the thesis is organized as follows: Chapter 2 discusses the creation of a full system simulation environment used to gather data on the behavior of the cache system while running Java applications. The primary contribution highlighted in this chapter is the creation of a full system simulator from readily available open-source tools. Several modifications were needed to make the desired measurements and the details are presented. We also show some practical deployment suggestions using a cluster that drastically increases the speed at which simulations can be run in bulk. This environment was used to gather the data for the

experiments discussed in Chapter 3 and Chapter 4. The data collected in these experiments would require multiple years of simulation time on a single CPU machine and the experiments would not have been feasible without the ability to run large numbers of simulations in parallel.

Chapter 3 includes a preliminary study of Java application behavior with respect to dynamic memory usage. It defines a new locality metric, *access density*, which allows for comparison among objects to determine locality patterns within an application. This metric is used to classify applications from the SPECjvm98 (69) benchmark suite and to show differences across applications. The primary contribution of this chapter is to provide a detailed analysis of program locality with respect to the dynamically allocated objects. This analysis helps to explain the behavior of programs in terms of their interaction with the memory management policy of a Java Virtual Machine.

Chapter 4 is a further study that leverages locality information gleaned by the access density categorization to propose the segregation of on-chip memory resources into two distinct regions, one for cache and one for scratchpad. We show that this hardware segregation can be matched to software segregation of the generational heap nursery from the rest of the application's memory resources. The main contribution of this chapter is the significant reduction in memory traffic that can be achieved by the collaborative hardware and software partitioning of the allocation space from the rest of the application memory.

Chapter 5 is a study that investigates hardware and software prefetching mechanisms with respect to Java object allocation as well as cache replacement biasing. This work aims to develop an alternate set of hardware mechanisms to support the locality of Java applications without the need for dedicated scratchpad resources. The main contributions of this chapter include the detailed cache analysis and the results that show prefetching can yield a significant amount of performance improvement when used for allocation while cache replacement biasing only shows benefits when bandwidth limits prefetching effectiveness.

Chapter 6 is the final study in the thesis and includes an in-depth performance analysis of Java applications with respect to various memory system parameters including heap size, nursery size and cache size. This work is unique in the thesis in that it utilizes real hardware

as opposed to simulation environments. It relies on a novel use of page coloring within the FreeBSD (28) operating system to partition the cache of a real system in several configurations. This real system environment is orders of magnitude faster than the simulation environments used in earlier chapters and affords a more exhaustive analysis of Java application performance and sensitivity to memory system parameters. The main contribution of this chapter is the novel use of page coloring to perform an in-depth cache analysis of Java applications on real system hardware. This evaluation, while confirming some general conclusions from other work, also includes some refining observations about memory system performance that are very practical for application developers, virtual machine developers, operating system developers, hardware developers and system administrators.

Finally, Chapter 7 provides some general conclusions from this thesis work. This chapter wraps up the discussion by providing a final, high-level discussion about the range of evaluations detailed in the previous chapters.

CHAPTER 2 Constructing a Memory Hierarchy Simulation Environment

2.1 Abstract

Modern computer systems are so complex that much of their inner workings are hidden and difficult to measure. Simulation is the only feasible approach to glean certain information. In this paper we describe a simulation environment constructed to gather information on the memory hierarchy behavior for a virtual machine running Java applications. We describe, in detail, the creation of the environment and the methods used to gather two different kinds of measurements: tabulated memory accesses for individual Java objects, and memory traffic for a proposed hierarchy, including a scratchpad. We also present a unique approach to quickly gather large amounts data through deployment of the simulation environment on a parallel cluster machine.

2.2 Introduction

The growing complexity of modern computers presents an ever increasing problem in evaluating efficiency and performance. The memory hierarchy is a good example of a complex system that is difficult to evaluate because of the multiple layers of cache. Each cache can have a different behavior based on associativity, line size and replacement policy. Couple a complex hardware with software that is also increasing in complexity and it becomes extremely difficult to evaluate a particular application in its interaction with the memory hierarchy.

There are two major approaches to expose architectural behaviors: performance counters, and simulation. While performance counters can provide useful information, the granularity is coarse. Simulation can provide a wealth of information to researchers about complicated behaviors that cannot be measured by other means, but it can be several orders of magnitude

slower than real execution. In some cases, the data gathered can be overwhelming to store and analyze. In this paper, we describe the construction of a simulation environment we use to gather two different measurements related to the memory hierarchy.

The first measurement is a tabulation of all memory accesses, reads and writes, to every individual object in a Java application. This measurement cannot be done with performance counters as the hardware has no concept of software objects. Although there is a possibility this measurement could be implemented in software without simulation, the instrumentation required would be impractical. Accesses to objects can originate from many different software levels including the Java application, the virtual machine and support software from the operating system and external libraries. Each software layer is largely independent. Each would have to be instrumented for accurate data collection. Ideally, we would like an environment that would run the entire software system, including applications, support software and host operating system, all unmodified. We want the environment to introduce as little instrumentation as possible to avoid interfering with the measurements we wish to make. Simulation is the natural solution.

The second measurement is traffic between cache and main memory, with the incorporation of a scratchpad in the memory hierarchy while running Java applications. Performance counters can provide memory traffic information for the system in which they are built. In the context of a proposed hypothetical architecture however, there is no general way to use performance counters for one memory hierarchy to predict behavior on a completely different hierarchy arrangement. There is also no feasible way to accurately predict memory hierarchy behavior through software instrumentation. Again, simulation is the natural solution.

In both of the measurements, we need to capture individual memory accesses. In the first, we want to correlate memory accesses to software Java objects. In the second, we want to correlate memory accesses to resulting traffic between memory hierarchy levels. This paper describes a single environment that we customize to make both measurements. In this work, we are not proposing a silver bullet for the measurement of the memory hierarchy. What we hope to provide is a comprehensive discussion on our simulation environment so that other

researchers might be able to leverage our experience when faced with similar problems.

The rest of the paper is divided as follows. Section 2.3 describes the construction of the simulation environment. Section 2.4 discusses catering the environment to collect the desired measurements. Section 2.5 describes related research work using other simulation environments and section 2.6 provides some concluding remarks.

2.3 Environment Construction

2.3.1 Components

The simulation environment consists not only of the simulator itself, but of all the components under simulation as well as the host system running the simulator. This section provides a description of all the components in the simulation environment. The layered view of the environment can be seen in Figure 2.1.

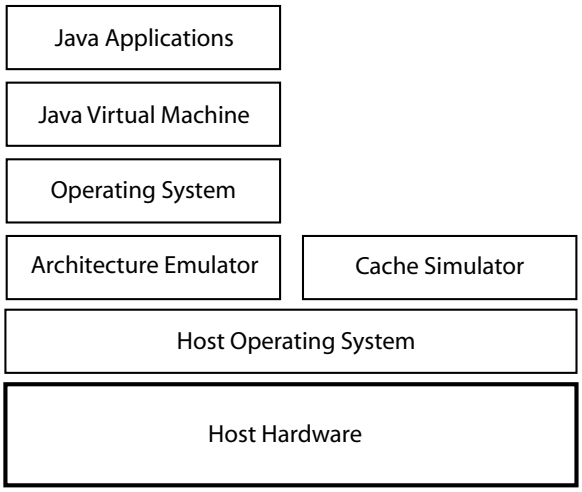

Figure 2.1 Simulation environment components

2.3.1.1 Java Applications

The top layer in the environment, for both the desired measurements, includes the Java applications under investigation. In both measurements, we use the applications from SPECjvm98(69). This set of seven applications is used extensively in the Java research community. The ap-

plications are complicated enough to significantly exercise the memory subsystem, while also providing multiple data set sizes that are suitable for running under simulation in a reasonable amount of time.

2.3.1.2 Java Virtual Machines

The next layer in the environment is the Java Virtual Machine. For each of the two measurements, we choose a different Java Virtual Machine. For the first measurement, accesses to individual objects, we opt to use Kaffe(42) because its memory manager does not relocate objects during their lifetime. For the second measurement, we choose SableVM(29). We modify SableVM to include a generational copying garbage collector and want to determine whether a new memory hierarchy will provide better locality support for such a collector.

2.3.1.3 Operating System

To support the Java Virtual Machine an operating system is needed to provide certain features such as threading and dynamic library loading. We use a custom install of Linux based on the Slackware distribution to keep the simulated filesystem small.

2.3.1.4 Architecture Emulator

The architecture emulator we use is bochs(48). Bochs emulates the x86 instruction set and supports the installation of a full operating system. It emulates the architecture as well as several hardware components including a graphical display. Bochs has built-in function stubs for instrumentation. Capturing accesses to memory from all instructions is straightforward.

2.3.1.5 Cache Simulator

The cache simulator we employ is DineroIV(26). This simulator is extremely flexible and can be used in a two distinct ways. It can be run as a stand-alone executable and process cache traces from files, file redirects or pipes. It can also be integrated directly into another application, by means of library calls, to process traces without redirection to a file or pipe.

2.3.1.6 Host System

The host system includes both the hardware and the operating system layers from Figure 2.1. The machines we use are all Pentium IV based x86 machines running Red Hat Linux. We use both stand-alone single CPU systems as well as a cluster with dual CPU nodes. The Linux versions vary among the systems we use to gather measurements. The environment is largely self-contained and does not need recompilation between the systems we use.

2.3.2 Integration

All of the components in the simulation environment are open-source, except for the benchmark applications from SPECjvm98. The flexibility offered by open-source components simplifies integration of the environment. All of the functionality of the various components is exposed and can be readily modified. In this section we describe the modifications made to link the various components together into a complete functioning environment.

The Java applications from SPECjvm98 are the ultimate target of the measurements and are left unmodified. The fact that the source is not available for the suite does not inhibit their integration into the environment. They act as the static inputs to the system.

The Linux operating system installed within the emulator is also an unaltered component. It is used simply as a support layer for functions needed by the virtual machines.

The two virtual machines, Kaffe and SableVM, need minor modifications in order to alert the environment with regard to desired events that are to be recorded. These modifications need be minimal so as not to interfere with the desired measurements. The method we choose to allow for communication of information from the virtual machine layer to the architecture emulator layer is referred to as the Magic Instruction. We borrow this concept from SimICS (79).

A Magic Instruction is a NOP instruction that does not appear in a standard executable. For x86, the instruction we use is *xchg %bx, %bx*. By placing this NOP explicitly in the source code of an executable via inline assembly the application can communicate directly with the emulator. We modify bochs to look for this specific NOP instruction.

Since the simulation environment is running a full multi-tasking operating system, it becomes important to be able to filter out the execution of a single application. Filtering can be handled by using the Magic Instruction in conjunction with the CR3 register of the x86 architecture. The CR3 register points to the page table for the currently running process, and will be unique to each process. By recording the CR3 register at the first appearance of a Magic Instruction, all other events, such as memory accesses, can be filtered to include only those of the desired process by first checking the value of the CR3 register. The Magic Instruction only appears in the executable under investigation.

The Magic Instruction can also be used to communicate data to the emulator. By explicitly setting individual register values via inline assembly prior to invoking the Magic Instruction, data can be passed from the application to the emulator. This process provides a means for the Java Virtual Machines to communicate the addresses of memory segments, such as the range of the heap, and even events such as object creation, along with the object sizes and addresses.

There is a significant advantage in configuring the simulation environment to use the Magic Instruction in this fashion. Since the Magic Instruction is a NOP, and the architecture emulator supports the same instruction set as that of the real hardware we use, the very same Java Virtual Machine executables built for simulation can be run directly on the real system. The executables can be verified on the real system and tested with all of the applications prior to deployment in the simulated environment. Verification on the real hardware is faster and saves lots of debugging time. The environment as a whole is simplified as there is no need for a cross-compiler.

The final component we integrate is the cache simulator. We considered the option of merging DineroIV directly into bochs to eliminate overhead of interprocess communication. However, DineroIV is not thread-safe and is designed to manage a single cache trace at a time. In order to speed collection of the entire desired data set at the minimal expense of individual runs, we opt to use the standalone executable version of DineroIV, unmodified. Using the executable version of DineroIV allows us the flexibility of configuring bochs to

generate multiple independent trace streams simultaneously and redirect each stream to a separate instance of DineroIV. These separate traces can be unique traces, such as instruction traces versus cache traces or filtered traces, or they can be duplicate identical traces that can be directed to DineroIV instances with different cache configurations. The interface between bochs and DineroIV was configured to use named pipes. Named pipes provide a means to create an arbitrary number of independent streams, each of which can be handled by a separate instance of DineroIV where each instance can be configured with separate parameters.

With the massive amount of data generated for an address trace during the execution of a Java application, it is infeasible to store the trace to disk. The traces, even compressed, quickly fill the largest available disks. Additionally, the time required to store and retrieve the traces to and from a disk is significantly longer than that required to simply rerun the simulation. The unmanageability of the large amount of data provides another strong incentive to optimize the simulation environment for simultaneous collection using multiple instances of DineroIV.

2.3.3 Cluster Deployment

Simulation is extremely time intensive. On a single processor standalone Pentium-IV 2.8GHz system, some simulations in this work can take more than 12 hours. When potentially hundreds of simulations are needed, it is obviously undesirable to run all of the jobs sequentially on a single machine. While procuring access to additional machines can help to reduce the time needed, resources are often limited and managing data spread across a multitude of machines can be difficult. The desired solution is a single machine that is capable of running a large number of simulations in parallel. In this section we describe the unique configuration options to the simulation environment that enable deployment on a cluster.

We want to run the jobs on a remote cluster in batch fashion. It would be undesirable to launch each simulation manually and monitor its progress from its graphical window. The first step we make is to build the bochs executable without an external display. None of the applications in SPECjvm98 are graphical, but we remove even the console display. This is done after verification of the environment with a display. Bochs already has support for building a

"headless" executable. This version of the emulator can be run remotely and launched from a batch scheduler (such as PBS) where no X session is available to the application.

The next major step is to allow for multiple instances of the simulation environment to be run from the same filesystem and to write results to a common location. The biggest obstacle is sharing the simulated filesystem within the emulator. A standard configuration of bochs uses a file configured as a hard disk image for the root file system. The problem arises when two independent emulators attempt to share the same hard disk image. Because the hard disk image is modified during the execution of the emulator, two instances of the emulator trample over one another's file system data and corrupt the hard disk image.

One solution would be to replicate the hard disk image for each instance of the emulator. Even with a minimal install of Linux, replicating the hard disk image is a significant waste of disk space. This is especially true in the clustered environment available to us where user quotas are relatively small. Even with temp space available, a large amount of time can be spent simply copying the disk image across the network.

To solve the problem, we use an alternate approach. The root file system must be readable and writable, but a large portion of the disk image does not ever need to be modified. Bochs also supports mounting ISO images as read-only file systems. We construct a bootable ISO image that creates the root file system as a ramdisk, and then mounts the ISO to that file system. The ISO contains all of the necessary OS resources as well as the Java Virtual Machines and Java applications. Since the whole simulation environment is now running on a read-only file system, it can also be terminated at any time without the danger of corrupting the file system. With a hard disk image, if the emulator crashes, there is the potential that the disk image file could be corrupted.

Booting from the read-only ISO image allows for multiple instances of bochs to be run simultaneously from the same disk image. The remaining problem is that we must have a way to externally trigger different applications to be run within the simulation environment. The solution is to dynamically create a second ISO image that contains only a small script with the desired commands to be run. The root file system is modified to run the script from the

second ISO when the operating system boots. (This is set in /etc/rc.local within the simulated root file system). The second ISO is extremely small and each instance of bochs is given a different image for its second ISO.

A single batch script can then generate an ISO to run a particular Java application within the environment, launch bochs and any number of instances of DineroIV and direct all of the file outputs to a common directory organized by Java application name, application parameters and cache parameters. The clustered environment provides a means of running large numbers of simulations in parallel as well as maintaining all of the collected information. With this arrangement we are able to complete a set of simulations in under a week that, on a single machine, would take over two months.

2.4 Experimental Measurement

In this section, we describe how the simulation environment described in the previous section is configured to perform two different measurements. The first measurement is the tabulation of all memory accesses to individual objects. The second is the traffic between main memory and cache for a variety of memory hierarchy configurations.

2.4.1 Object Accesses

The first measurement we collect with the simulation environment is the tabulation of all accesses, reads and writes to each individual object in a Java application. As described above, there is no feasible way to collect this information without simulation. Even under simulation, this type of measurement requires a sophisticated mechanism to associate memory accesses to individual objects.

The first important choice in designing the mechanism to record accesses for individual objects is the Java Virtual Machine itself. We choose Kaffe because it does not relocate objects during their lifetimes. A virtual machine that relocates objects could be used for this type of measurement, but the environment would have to track the object across relocation events. This is entirely feasible, but we opt for a simpler design.

The method we employ is to use the Magic Instruction to convey the heap boundaries when the heap is allocated by the virtual machine. A set of pseudo-heaps are mirrored within the simulation environment as shown in Figure 2.2. One of the pseudo-heaps is used to store data about where the objects are located, one is used to tabulate read accesses and the last is used to tabulate write accesses.

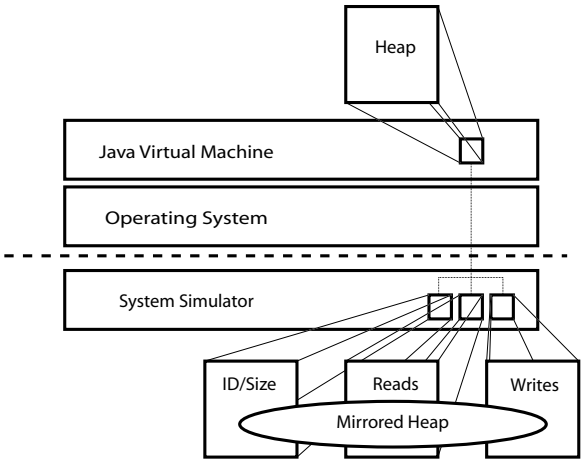

Figure 2.2 Mirrored Java heap in simulator

When an object is allocated the Java Virtual Machine signals bochs via the Magic Instruction that a new object has been created, including the object address and size. From within bochs, the object location heap is updated. All read and write accesses that fall within the bounds of the heap, whether caused directly by the Java application or the result of virtual machine behaviors such as garbage collection, are simply added to the corresponding counters in the respective read and write pseudo-heaps. When an object is reclaimed by the garbage collector, the Java Virtual Machine signals bochs with the address of the reclaimed object. This signal triggers the simulation environment to tabulate all of the reads and writes that fall within the object's memory region which will be written to a log file along with additional object information (size, etc.). The object regions in all three pseudo-heaps are reset to accommodate a new object allocation.

The above process will provide the data for most objects within a Java application. However, not all objects will be reclaimed by the garbage collector during the execution of the

application. Some objects live for most of the application's execution and some objects allocated near the end of the program simply will not have been scavenged by a garbage collection invocation. These objects are handled when the application terminates. The virtual machine signals bochs that the application has terminated. All remaining objects in the pseudo-heaps can be tabulated and recorded at this time.

2.4.2 Scratchpad Modeling

The second measurement we make is that of traffic between cache and main memory for two memory hierarchy configurations. This measurement requires the integration of DineroIV into the environment.

The first memory hierarchy configuration is a standard layout with cache and memory. We simulate only one level of cache as we are interested only in the traffic between the last layer of cache and main memory. In most cache configurations, the contents of the last layer of cache is a superset of the contents of the lower layers. Thus the lower layers can be ignored entirely.

In this measurement, we include only the memory accesses from the Java Virtual Machine process, and filter all other accesses from other system processes based on the value of the CR3 register as described above. We write all accesses from this process to a named pipe to which an instance of DineroIV is attached. DineroIV simulates cache behavior based on specified parameters and reports traffic to and from memory as one of its calculated statistics.

The second memory hierarchy configuration includes a scratchpad at the same level as the cache, as shown in Figure 2.3. Scratchpad is simply a fully addressable memory space that is software managed. The address range of the scratchpad is disjoint from that of main memory. Any data located in the scratchpad is not located in the cache or main memory unless an explicit copy is made. Scratchpad is found in commercially available embedded devices. In our evaluation, we want to simulate scratchpad sizes and configurations that are not commercially available.

For our evaluation, we want to determine if using a hierarchy with a scratchpad can be more efficient, in terms of traffic between main memory and cache, when compared to a cache-

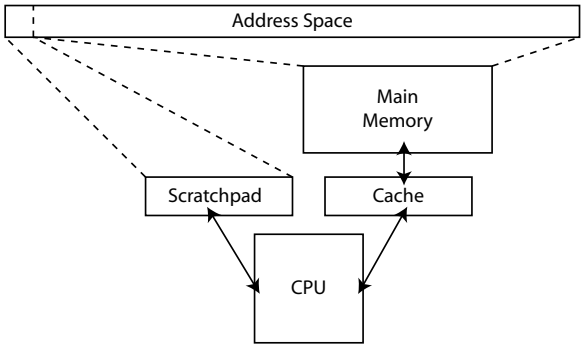

Figure 2.3 Memory hierarchy with cache and scratchpad

only hierarchy with an equivalent storage capacity. For this evaluation, we map the nursery of a generational garbage collector, which we developed for SableVM, to the scratchpad. This is simulated by recording the address boundaries of the nursery using the Magic Instruction as before. To simulate the addition of the scratchpad to the hierarchy, we use bochs to filter out all accesses that fall within the address range of the nursery, and hence, the scratchpad. These accesses can simply be discarded in the stream sent to DineroIV as all accesses to scratchpad can be considered hits.

One major benefit of integrating DineroIV through named pipes is that multiple instances can be run simultaneously. In this example, two separate streams can be generated for the same Java application execution; one unfiltered full trace can be sent to one instance of DineroIV simulating the cache-only hierarchy, and a filtered trace can be sent to a separate instance to simulate the scratchpad/cache hierarchy. This organization allows for the collection of multiple traces without having to rerun the simulation. Either stream could be replicated any number of times to separate instances of DineroIV for simulations of different cache parameters.

2.5 Related Work

This section describes some select simulation environments that may be of interest to someone investigating potential tools for their own work. We briefly describe some other environments that have recently been used in related research.

2.5.1 bochs

The only other published work we are aware of that uses the bochs emulator is that of Scott Kaplan (44). He uses bochs in conjunction with other tools to gather information about virtual memory management. Much of the fundamental elements of collection are similar, but the granularity of the measurements are much finer in our case, as we are dealing with individual objects and cache behavior. We are aware of no work that describes the ability to cater such an environment as we have developed as far as deployment in a cluster for parallel simulation to reduce data gathering time.

2.5.2 Dynamic Simple Scalar

One of the most comprehensive environments we are aware of that has been used in experiments most closely related to our research area is Dynamic Simple Scalar(5). This project was developed by the Architecture & Language Implementation (ALI) research group at the University of Massachusetts Amherst and the University of Texas at Austin. Dynamic Simple Scalar is an upgrade to Simple Scalar(68), the de-facto standard in architecture research, that includes support for running a Java Virtual Machine. As mentioned previously, a Java Virtual Machine may rely on the operating system or external libraries to provides certain functionality. Dynamic Simple Scalar bridges the gap to enable execution on the architecture simulator. A recent work by Hertz used Dynamic Simple Scalar to gather memory statistics on a per object basis in order to compare garbage collection to explicit memory management(35).

2.5.3 Simics

Simics is a simulation environment that has found widespread use in the research community in a large number of areas (79). Simics is capable of emulating several architectures, including x86, and provides a host of features not supported by bochs. We initially investigated Simics very seriously. Our final decision to move to bochs was based on licensing issues. Bochs is open source and requires no license. While Virtutech will supply a single user license free of charge for academic purposes, the license is restricted to one hardware profile. This type

of licensing prohibits deployment on a cluster or even on multiple standalone machines. The measurements we make do not require the advanced features of Simics and can be gathered with bochs. For other types of measurements, it may be necessary to use a more sophisticated tool such as Simics.

2.5.4 Vertical Profiling

Although quite a different approach, Hauswirth, et. al., created an environment, titled Vertical Profiling, that aims to leverage the hardware performance counters to correlate behaviors from multiple software layers to behaviors at the hardware level. The first work demonstrated the technique and was largely a manual data collection process, but continued work is attempting to move the process towards automation(34; 33). This approach is significantly different from our simulation environment but may be an alternative option for certain types of research.

2.6 Conclusion

In this work, we present a detailed description of a simulation environment useful in the collection of memory hierarchy measurements. We describe the use of the environment for two distinct types of measurements of the memory hierarchy: accesses to all individual objects in Java applications, and traffic between cache and main memory for memory hierarchy configurations with and without scratchpad. We describe in detail the integration of the environment that enables the collection of these measurements. We also describe the unique considerations we incorporated to deploy the simulation environment on a cluster which enables parallel execution of multiple simulations which, in turn, speeds the experimentation process. We hope this work is presented in a fashion that enables others to leverage our experience when investigating new research problems.

CHAPTER 3 Access Density: A Locality Metric for Objects

3.1 Abstract

With the increasing speed gap between processors and main memory, the performance of computers is becoming largely dependent on efficient use of the memory hierarchy. In this work we introduce access density, a new locality metric suitable for identifying locality in object-oriented systems. We present a detailed theoretical background for access density and show how the metric captures the locality of objects. We then demonstrate how locality trends uncovered by the application of access density can be exploited in a set of Java applications by modifying the memory hierarchy to support object locality. The experimental results from the modified architecture validate the access density metric through both positive and negative examples. The applications that exhibit the targeted locality trend generate 11% to 21% less memory traffic when using the modified memory hierarchy. The applications that do not exhibit the trend generate more memory traffic under the modified hierarchy.

3.2 Introduction

Performance of modern computer systems is largely dependent on the use of the memory hierarchy. Caches are becoming increasingly large but miss penalties are significant due to main memory latency. It was recently reported that Java applications can spend as much as 45% of their execution time waiting for main memory (1).

Object-oriented programs, in general, are memory intensive. They allocate large numbers of small objects that tend to die very quickly. While objects might be related by references, often their relative positions in the heap (and thus the virtual address space) are unknown to the compiler, especially in a virtual machine environment that employs garbage collection. In

languages that run in a managed environment, the compiler can make no assumptions about memory locations of objects as the allocator can vary from one virtual machine to another and garbage collection can relocate objects during execution. Compiler optimizations for the memory hierarchy are thus extremely limited for object-oriented programs. Since many object-oriented programs are written to be portable across platforms, whether by recompilation or distribution in byte-code form, tuning at the source code level for a particular cache architecture is undesirable.

The object-oriented paradigm is the dominant paradigm for new software written today. Most programmers today use object-oriented languages such as Java, C#, C++ or Visual Basic in the development of their applications. In the TIOBE Programming Community Index listing for October, 2005, nine of the top ten most popular programming languages are object-oriented(75). Sun's Java and Microsoft's .Net initiative are making managed code running on a virtual machine the mainstream for software development. Robustness, portability, quick time to market and ease of design are but a few of the benefits of developing software in this type of framework. The sheer prevalence of these technologies warrants serious evaluation of their performance.

The key problem for object-oriented systems in terms of memory performance is that objects don't map nicely to hardware cache lines. Cache lines are a fixed size from 32 to 128 bytes in modern processors, and the trend is to continue to increase that size. Some researchers are even proposing 512 byte cache lines (32). Objects, however, tend to be small. In the case of Java, most objects are smaller than 32 bytes(21). Since objects can also vary in size (generally aligned to word boundaries), it is also quite likely that objects will not align with cache lines. Two resulting side effects are that individual cache lines are likely to contain multiple objects and some objects will cross boundaries and lie in multiple cache lines.

These side effects alone are not problematic, but when coupled with another discrepancy between cache behavior and object behavior, a serious problem arises. The discrepancy is that objects tend to have a short lifetime. They are allocated, used briefly and then no longer needed. Cache hardware, however, has no concept of liveness. It only knows *modified*. Since

every object will have been modified during it's lifetime (zeroing, initialization, actual data usage), the cache will assume the data is useful and write the contents to main memory regardless of whether those objects will ever be used again. Since dead objects and live objects will be intermixed within the cache lines, dead objects can potentially consume memory bandwidth in both directions (write traffic and read traffic). The traffic consumed by dead objects is entirely wasted and contributes to inefficiency by occupying space in cache and on the memory bus that could potentially be used by useful data.

In this work we focus on object-oriented programs, with specific examples from Java, the language currently ranked number one by the TIOBE Programming Community Index. There is room for significant performance improvement if these applications can be tuned to more efficiently use the memory hierarchy.

There has been substantial research work conducted in the area of cache performance of object-oriented systems employing garbage collection. Many of these works evaluate the use of various garbage collection schemes and their impact on cache performance. Others attempt to modify the memory management system to provide greater cache performance. In this work we have a more generalized approach and aim to provide a more comprehensive understanding of locality in object-oriented systems. Through our work we reconfirm findings from previous work while providing significant additional insight into those findings. Our new evaluation also leads to the development of a technique to substantially reduce the traffic between cache and main memory for many of the applications under investigation.

In this work we:

1. Define *object locality* as a locality concept at the granularity of individual objects.

2. Introduce *access density*, a new metric that can be used to provide a measure of object locality.

3. Calculate access density for objects within the applications of SPECjvm98.

4. Evaluate the object locality patterns in SPECjvm98 as given by the access density calculations.

5. Develop a technique for exploiting a specific object locality pattern.

6. Simulate the impact of the new technique on traffic between the cache and main memory.

7. Validate the access density metric with the simulated memory traffic results.

The remainder of this paper is divided as follows: In Section 3.3 we provide the theory and discussion of object locality and the introduction of the access density metric. In Section 3.4 we describe the experimental setup for calculating access density values for objects in SPECjvm98, as well as the simulation framework for measuring traffic between the cache and main memory. In Section 3.5 we provide a detailed discussion and evaluation of the experimental results. In Section 3.6 we discuss related work and in Section 3.7 we conclude the discussion.

3.3 Theory

Work in the area of memory system performance of garbage collected systems often relies on experimentation. While many interesting results have been uncovered through experimentation with varying garbage collection schemes' interactions with cache, the potential discoveries are limited by the experimental permutations evaluated and the information available for collection. Modern computer processors provide mechanisms for monitoring overall cache performance, but the data is not fine grained enough to permit detailed analysis of why one algorithm might outperform another.

In this work we start at a higher level and attempt to redefine the factor we expect to underlie performance of the memory hierarchy. That factor is locality. Qualitatively, locality is described in two ways. Spatial locality refers to the likelihood that an access to memory will be followed by subsequent accesses to nearby addresses. Temporal locality refers to the likelihood that an access to memory will be repeated in the near future.

In object-oriented systems, objects are the dominant memory consumers and are thus the focus of our new definition of locality. We define *object locality* to be a conceptual hybrid of spatial and temporal locality at the granularity of individual objects. Without the consider-

ation of a particular cache organization, our simple definition of *object locality* is this: Given the choice between two objects, which would we prefer to cache?

The reason we take such a high level view of locality is because once a specific cache organization is assumed, cache behavior is fixed. We want to abstract out the locality of the program without being confined to a particular cache architecture. The most compelling reason to separate evaluation of object locality from cache locality is that, as mentioned previously, objects don't map nicely to cache lines.

Rather than focus on the interplay between objects and cache organization, we focus on object locality. To fully define object locality, we first need to define an object. An object is a memory region that is persistent in both time and space. Although an object may be relocated in the physical and even virtual address space during its lifetime, from the application point of view it is the same object. Locality is a concept that deals with multiple memory accesses and the likelihood that they are related in either time or space. Thus, any measurement of object locality should include values for accesses, time and space.

And now we begin the introduction of *access density*. Access density is the metric we have created to quantify object locality. The formula for calculating access density is given in Equation 3.1, where A is the number of accesses an object receives, S is the size of the object in bytes, and t is the lifetime of the object in bytes allocated.

$$AccessDensity = \frac{A}{S \cdot t} \tag{3.1}$$

The meaning of the value of access density may not be immediately intuitive. First consider that we want to incorporate both spatial and temporal locality within the same metric. Dividing the number of accesses by the size of the object gives a simple unit in accesses per byte. However, dividing the number of accesses to the object by the lifetime of the object is a little less obvious. We have chosen to use the lifetime metric of bytes allocated. This metric is commonly used when measuring the lifetimes of objects in garbage collected systems as the timeframe in which garbage collection is generally triggered depends on exhaustion of a region of the heap and not with the progression of real time, CPU time or another time metric(40).

Choosing bytes allocated as our time metric, we have both space and time defined in terms of bytes. Cache has no concept of real time and deals in terms of spatial distance, in bytes, between accesses and temporal distance, in the number of bytes accessed, between repeated accesses. Access density provides us with a similar expression for object locality in terms of accesses per byte, per byte allocated. We assert this metric is sufficient for comparing the object locality among objects in a system. Objects with a higher access density value have greater object locality and are preferable when choosing between objects.

To provide a simple example, consider a cache with fixed size that can contain a set of objects. Multiple objects can reside in the cache at the same time if the sum of their sizes is less than or equal to the size of the cache. Multiple objects can occupy the same space in the cache if they do so at different times. Given these two boundaries of size and time, we need to identify which objects, when located in the cache, will give the highest performance.

Objects themselves have the same two dimensions of size and time as cache. An object's size is determined when it is allocated. An object's lifetime is determined by the time that passes from its allocation to the time it is reclaimed and its space can be reused for other objects. Therefore, we can view a set of objects as a set of rectangular areas that we are trying to fit within the rectangular area of cache. Figure 3.1 shows two possible object arrangements for the same cache size.

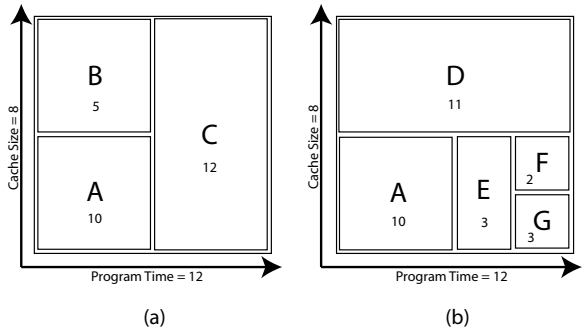

Figure 3.1 Object arrangements in scratchpad

Assume we know that the number of accesses to the objects A through G are those given in Table 3.1. With this information we could calculate that configuration a has 27 accesses to

cache ($A + B + C = 10 + 5 + 12 = 27$) while configuration b has 29 ($A + D + E + F + G = 10 + 11 + 3 + 2 + 3 = 29$). Based on our previous objective, we prefer configuration b because it has more accesses. Accesses to objects not found in the cache would be misses. However, we don't want to generate all possible configurations and compare them. What we want is a way to identify which objects are more likely to participate in arrangements that lead to a high number of accesses to cache, and therefore have the best locality.

Access density allows us to compare regions that have differing sizes and lifetimes to evaluate which has a higher locality. When applied to the whole cache in Figure 3.1, taking S and t to be 8 and 12 respectively, we see that the access density of configuration b is higher than that of a (see Equation 3.2). The real advantage, however, is that we can apply access density to the individual objects in our example. The access density values for each object have also been calculated and are included in Table 3.1.

$$\frac{A_a}{S_a \cdot t_a} = \frac{27}{8 \cdot 12} = 0.28 < \frac{A_b}{S_b \cdot t_b} = \frac{29}{8 \cdot 12} = 0.30 \tag{3.2}$$

Object C may have the highest absolute number of accesses (12), but it does not have the highest locality. Object G has the highest locality (0.50) because although it has only a few accesses (3), it also takes up very little space (2) and lives only a short time (3). Thus, more objects can be collocated with object G to achieve the greatest locality overall within the cache.

After identifying the objects with the highest access density, we still have to determine which objects to assign to the cache. In our example, objects G and A and F are the three with the highest access density and can all be located in the cache together. Now we are left to fill in the remaining available space. Objects C and E both have the same access density

Table 3.1 Object parameters and access density

Object	**A**	**B**	**C**	**D**	**E**	**F**	**G**
Accesses	10	5	12	11	3	2	3
Size	4	4	8	4	4	2	2
Lifetime	6	6	6	12	3	3	3
Access density	.42	.21	.25	.23	.25	.33	.50

value, but C cannot be placed in the cache because it conflicts with objects F and G which are live at the same time as C is live in the program.

This example is grossly oversimplified and not intended to show how to assign objects to the cache. In reality, objects are not allocated simultaneously and will not necessarily completely fill a cache region. Choosing the absolute optimal solution would require an oracle to know how many accesses objects will receive and how long they will live. Access density is a profile based metric that is designed to capture locality at the object level to help uncover locality trends. The example is meant only as a visual presentation of the meaning of access density and how it quantifies the locality of individual objects to provide a means of comparison. Access density helps to identify which objects have the highest locality. Selecting high locality objects will lead to the best overall locality.

While access density will allow us to separate objects by their locality, there are many factors which we could attempt to correlate to object locality. Our goal is not to show all possible locality trends of object-oriented applications, but rather demonstrate a method of identifying locality trends. In the rest of this work we identify one locality trend through the application of access density. We then devise a technique to exploit this trend and finally verify the identified locality through simulation of the devised technique.

The trend we uncover is that in many Java applications small, short-lived objects have the highest locality. The technique we then employ to exploit this trend is to map the nursery of a generational garbage collector to a software-managed scratchpad. This mapping helps to eliminate the unnatural mapping between objects and cache lines for the most volatile of objects, those that are small and have short life-spans.

3.4 Experimental Setup

3.4.1 Calculating Access Density

The first set of experiments conducted in this work is the calculation of the access density metric for all objects in the applications from SPECjvm98(69). The information necessary for these calculations includes the number of accesses to, the size of and the lifetime of each

object. Gathering this information is complicated. The size of each object can be gathered easily as the virtual machine knows the size of an object at the time it is allocated. The other two values, however, are not explicitly known at any level in the system at any point in time.

The accesses to the object can originate from the application directly at the bytecode level, through interaction with virtual machine functions or even through memory management functions of the garbage collector. The total number of accesses to each object is a summation of individual accesses to fields within that object over the lifetime of that object. Modern computer systems obscure the individual accesses to memory locations by nature of the integration of cache on-chip with the processor, completely prohibiting physical recording. To gather complete address traces for all accesses, we choose to use simulation.

There are many simulation tools available that offer varying sets of features. For this wor,k we wanted full address traces for a Java Virtual Machine running the SPECjvm98 benchmarks. Many Java Virtual Machines rely heavily on operating system services, such as thread scheduling and dynamic library loading, to provide their functionality. These requirements are fulfilled easily through the use of a full system simulator. We chose bochs(48), an x86 architecture emulator on which we installed a small footprint of Linux. We chose to use Kaffe (42) as the Java Virtual Machine because it's allocator never moves objects during their lifetimes, which makes it simple to track accesses to individual objects.

Object lifetime information is difficult to obtain. An object in a garbage collected environment is *dead* at the earliest point in which it can be reclaimed. In normal execution, garbage collection is run only periodically, and objects are reclaimed en masse. While an advanced technique(36) has been developed to significantly reduce the overhead associated with collecting lifetime traces, we used the brute force method of invoking garbage collection at every object allocation within Kaffe to determine object lifetimes.

The access information and lifetime are generated in two separate traces. The two traces are then aligned based on object size, and the information is combined. Alignment and combination of trace information can be done because the benchmarks are repeatable and run deterministically. Some minor variations occur in objects allocated by the virtual machine due

to the environment it runs in (real machine vs. emulated machine), but those variations are easily accounted for. After the two traces are combined, access density is calculated for each individual object.

One non-deterministic behavior we did not attempt to rectify is thread scheduling. All of the SPECjvm98 benchmarks except *mtrt* are single threaded. In the case of *mtrt*, we opted to substitute *raytrace*, the single threaded version of the benchmark, for the access density calculations. The multithreaded version simply runs two instances of *raytrace*, and thus the object locality patterns should be nearly identical. The validation we perform in the second phase of experimentation uses *mtrt* and the expected behavior is observed.

3.4.2 Measuring Memory Traffic

To validate the access density metric, we exploit a pattern uncovered by the metric, namely, that small, short-lived objects have the highest access density in many applications. We take advantage of this behavior by isolating the nursery of a generational garbage collector from the rest of the memory subsystem in a separate software managed cache, often referred to as a scratchpad. This isolation naturally segregates a majority of the small, short-lived objects from the rest of the objects. To perform a comparison between the unmodified system employing a single cache and the new system with cache and scratchpad, we need to measure the resulting changes in memory traffic.

Measuring traffic between the cache and main memory requires much the same environment needed to calculate access density. A full address trace is needed. We again use bochs. Rather than recording accesses to individual objects, the addresses are sent directly to DineroIV(26), a cache simulator. This time we used SableVM(29) as the Java Virtual Machine. We chose SableVM because of its copying garbage collection algorithm. We were able to modify the algorithm to support generational garbage collection with a fixed-sized nursery that can easily be configured to various sizes. We have contributed the collector back to the SableVM project. The code for the generational collector is included in the source code distribution of SableVM.

To make a fair comparison, we compare the two systems in which the total storage ca-

pacity of the cache and scratchpad equal that of the cache in the cache-only system. In both configurations, we configure the cache to be 8-way set associative with 64 byte line sizes. The metric we use for comparison between the systems is total traffic in bytes between the cache and main memory.

3.5 Results

The results are broken into two subsections. In the first subsection we describe the access density calculations and the uncovered trend in object locality. In the second subsection we describe the memory traffic measurements of a system designed to take advantage of a specific object locality trend as compared to the traffic of an unmodified system. We first identify object locality, and then we verify the accuracy of the identification by correlating the memory traffic results with access density calculations.

3.5.1 Access Density Calculations

Before discussing the particular results we describe the methodology for presenting the data. First, we calculate access density for all the objects in the applications of SPECjvm98. However, each application has a large number of objects, in most cases there are several million. We need some way to view the data. We create a dot plot on a log scale for the access density versus both the object size and the object lifetime as shown for *javac* in Figure 3.2.

An obvious trend is demonstrated in this plot, but with so many overlapping data points, it's difficult to see how the objects are distributed along the trend. To provide better visualization, we break down the data into separate plots for size and lifetime. We then calculate average values for access density at every discrete interval (size in bytes, or lifetime in bytes allocated). The average value curve is plotted in one graph (top graph) on a log scale for each benchmark as shown in the two sets of graphs for *javac* in Figure 3.3. The average value curve shows the overall trend for access density verses size or lifetime, but the concentration of that trend still cannot be seen.

A second graph is plotted with the same x axis for each benchmark (e.g. middle graphs for

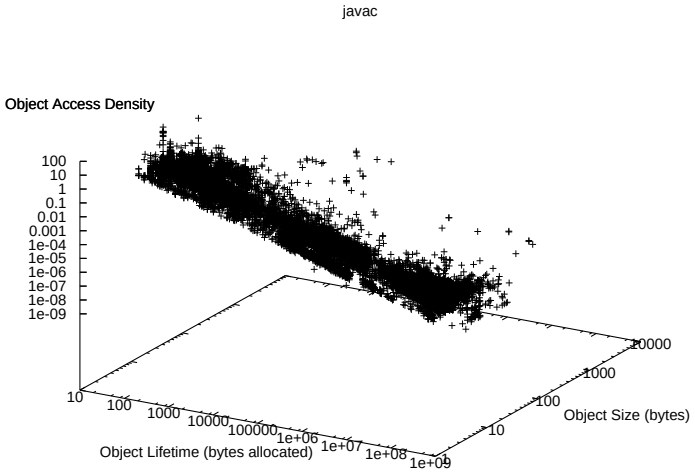

Figure 3.2 Access density for javac

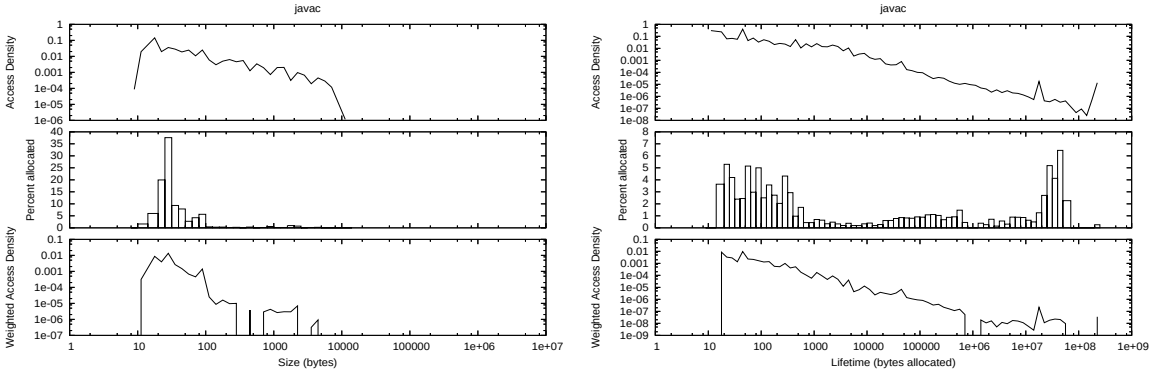

Figure 3.3 Average and weighted access density vs. object size and vs. object lifetime for javac

javac in Figure 3.3). This graph is a distribution of objects within each interval plotted as a percentage of the total bytes allocated. This graph is plotted on a non-log scale. Any interval with less than a threshold of 0.01% of the total space allocated was excluded from further consideration. The space distribution graph helps to see where the objects are concentrated along the access density curve.

To further enhance the view of the data, a third graph is plotted for each benchmark, again, with the same x axis(bottom graph). This graph is a multiplicative combination of the previous two graphs, again plotted on a log scale, which we refer to as *weighted access density*. While simplistic in construction, the weighted access density graph provides some very useful information.

Access density, as found in the first graph, is an accurate average of the object locality for objects of varying sizes and lifetimes. Weighted access density on the other hand, gives us a look at the object locality for all objects in the system as a whole. A very small number of objects of a particular size may have a very high access density and appear as a peak for object locality. If, however, they account for an extremely small portion of the object memory space in a system, they are not significant contributors to the overall object locality of the system. Weighted access density helps show where the object locality resides in terms of both access density and volume of data.

Now we discuss a major trend in object locality in terms of access density that is exhibited in several of the benchmarks. That trend is that the object locality for many programs is the highest for small, short-lived objects. Along this trend, we break the benchmarks of SPECjvm98 into three distinct categories. The first is the set of applications that follow the trend very closely. The second consists of the applications that partially exhibit the trend but also have significant locality counter to the trend. The third is the set of applications that do not follow the trend.

The first part of the trend is that small objects have the highest access density. We define small objects to be less than 1000 bytes. This size is on the same order of magnitude as the 2KB large object threshold for the generational garbage collector in SableVM. Any object smaller

than this threshold will be allocated in the nursery, while objects larger than this threshold will be allocated directly into the mature space. The trend is clearly visible without specifying a threshold, but the threshold helps us to more explicitly define our categories. Since we use the generational collector in our technique to take advantage of the access density trend, it makes sense to choose our threshold based on a parameter of the collector.

The weighted access density vs. size graph for *javac* in Figure 3.3 (third graph in left set) clearly shows the trend as the access density is highly concentrated for objects smaller than 100 bytes. The additional benchmarks from SPECjvm98 that exhibit the highest access density for small objects are *jess*, *raytrace* (single threaded substitute for *mtrt*) and *jack*. Their graphs are found in Figure 3.4. Both *jess* and *raytrace* have access density almost exclusively concentrated in objects less than 100 bytes. In *jack*, the object locality is less concentrated but still well concentrated for objects smaller than 1000 bytes.

For the size component of the trend, two applications fall into the second category and partially follow the trend. The applications are *db* and *mpegaudio*. Their graphs can be found in Figure 3.5. While both applications exhibit high concentration of access density for objects below 100 bytes, they also both have significant spikes beyond our 1000 byte threshold. In the application *db*, there is a significant spike for objects near 100,000 bytes. In the application *mpegaudio*, there are significant spikes just beyond the 1000 byte threshold.

The last category contains only a single application that does not follow the trend that small objects have the highest locality. That application is *compress*. The graph for compress can be found in Figure 3.6. It is quite obvious that this application does not follow the trend as access density is concentrated in objects that are larger than 10,000 bytes.

The second part of the trend we show is that short-lived objects have the highest access density. We define short-lived objects to have a life-span less than 100,000 bytes allocated. It has been reported that of the objects allocated in SPECjvm98 applications, 60% or more have a life-span less than 100KB of allocation(21). This statistic has been cited to justify the use of generational garbage collection for these applications.

Again, *javac* (Figure 3.3) is a representative of the applications that follow this trend and

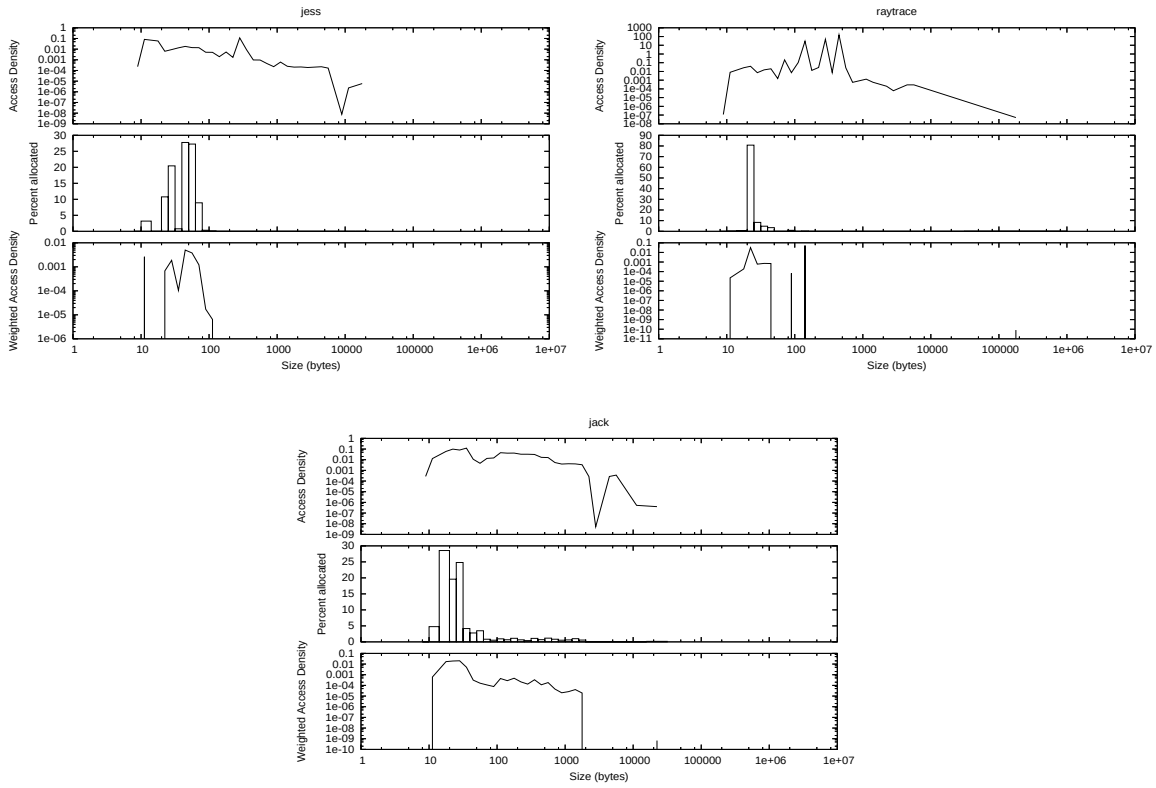

Figure 3.4 Follow size trend - Average and weighted access density vs. object size - Small objects have best locality

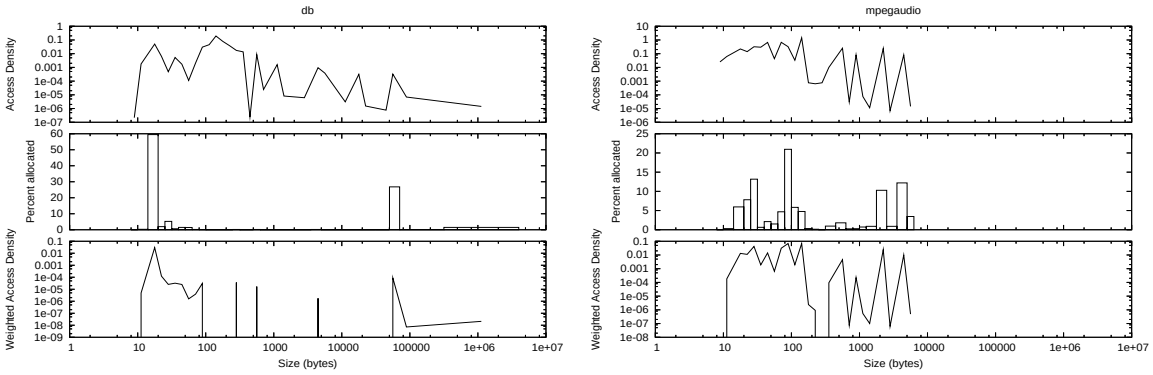

Figure 3.5 Partially follow size trend - Average and weighted access density vs. object size - Small objects have best locality - also significant locality in large objects

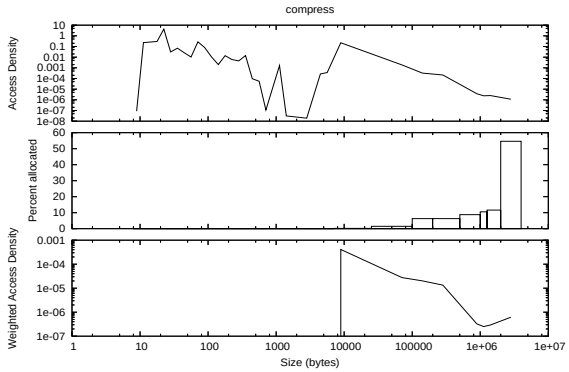

Figure 3.6 Do not follow size trend - Average and weighted access density vs. object size - Large objects have best locality

falls into the first category. While *javac* has a very nice continuous distribution of access density across a range of object life-spans it is still quite clear that the concentration is significantly higher for objects with a life-span less than 100,000 bytes allocated. The other benchmarks that follow this trend are, again, *jess*, *raytrace* and *jack*. Their graphs are found in Figure 3.7. All three applications show concentration of access density almost exclusively for objects with a life-span less than 100,000 bytes allocated, and even largely below 10,000 bytes allocated. The application *jess* has access density concentrated in very short-lived objects with nearly all of its objects' life-spans well below 10,000 bytes allocated.

The second category of applications that partially match the trend that short-lived objects have the highest access density includes *db*. Its graph can be found in Figure 3.8. This application has access density distributed in discrete regions for a wide range of object life-spans. While the overall trend still matches, that short-lived objects have the highest access density, there is a significant concentration of access density in objects with life-spans between 100,000 bytes and 1,000,000 bytes.

The third and final category of applications that do not match the trend that short-lived objects have the highest access density includes *compress* and *mpegaudio*. Their graphs can be found in Figure 3.9. Both of these applications exhibit some of the highest values of access density for objects with life-spans greater than 100,000 bytes, as well as high concentrations

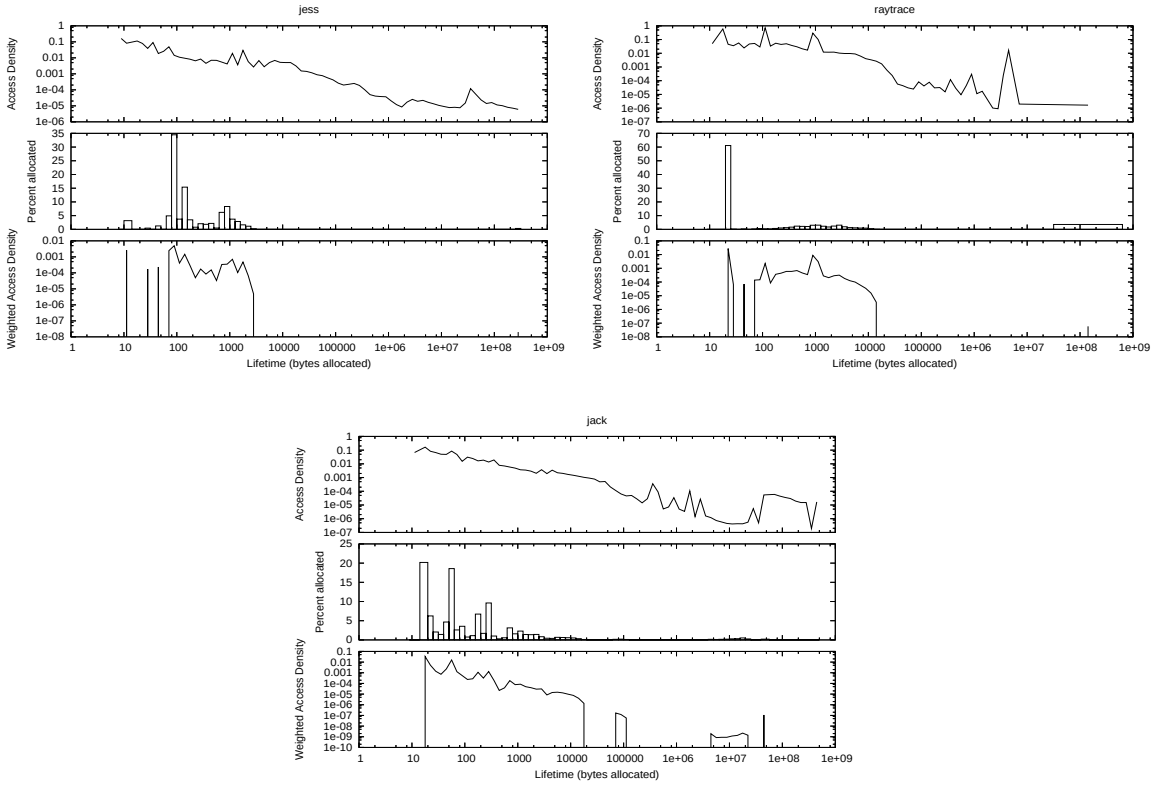

Figure 3.7 Follow lifetime trend - Average and weighted access density vs. object lifetime - Short-lived objects have best locality

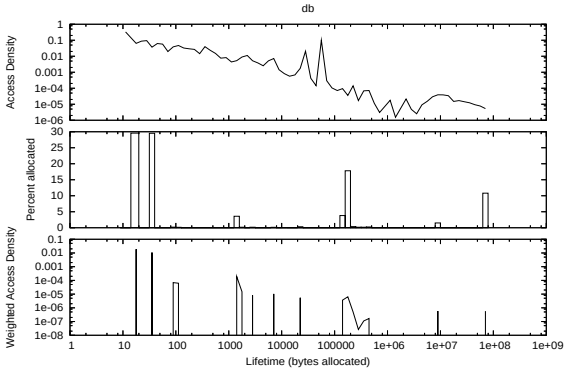

Figure 3.8 Partially follow lifetime trend - Average and weighted access density vs. object lifetime - Short-lived objects have best locality - also significant locality in long-lived objects

of access density in the longer life-spans.

3.5.2 Access Density Analysis

There are a few key observations to be made from the access density results for the applications from SPECjvm98. First, the identified trend that small, short-lived objects have the highest object locality reconfirms previous results that indicate that generational collector variants have the best cache performance(11). Since locality is highly concentrated in small, short-lived objects, and since these objects are allocated contiguously within the nursery of the collector, the nursery becomes a very high locality region that can be exploited by the cache. We further exploit this locality by isolating the nursery in a software-managed scratchpad to segregate this high locality region from interference with accesses to the rest of the memory which reduces memory traffic by filtering out dead objects.

The second observation is that not all of the applications in SPECjvm98 follow this trend. Based on the access density graphs for *compress* we expect that no improvement can be made by employing a scratchpad because nearly all of its data is allocated in objects that are too large to be allocated within the nursery. When running *mpegaudio*, we also do not expect to see the benefits of scratchpad because object locality is concentrated in longer-lived objects. It is important to note that both of these applications are almost always excluded from research in garbage collection. Part of the reason is because both only allocate a few thousand objects, while the remaining applications allocate millions, and thus they do not significantly exercise a garbage collector. We include them as they provide good negative examples and support the access density metric.

The application *db* is another interesting case study. Because it has such a unique distribution of object locality, both in short-lived and long-lived objects as well as in small and large objects, we expect it to see both positive and negative results in terms of memory traffic, depending on the size chosen for the nursery (and consequently, scratchpad). As *db* is a synthetic benchmark with very distinct phases (database creation and querying), it is not a good representation of applications in general, but does provide an interesting example for access

density as it can be used for both positive and negative verification.

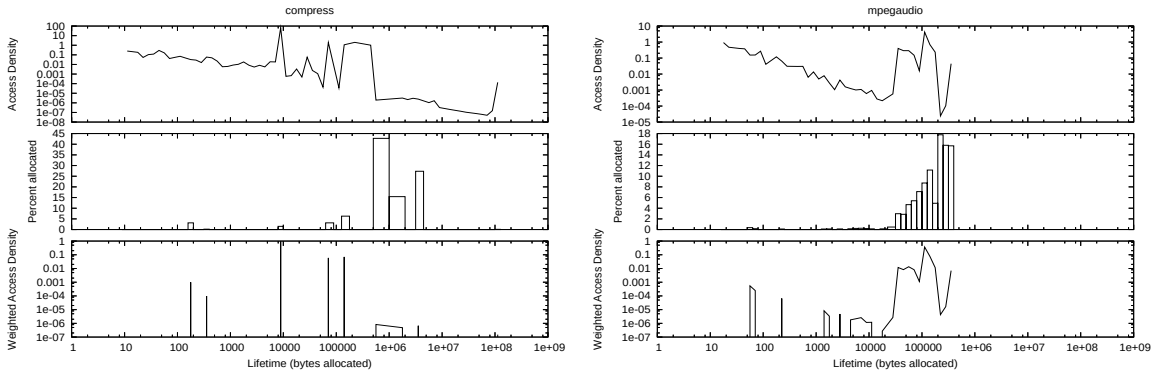

Figure 3.9 Do not follow lifetime trend - Average and weighted access density vs. object lifetime - Long-lived objects have best locality

3.5.3 Memory Traffic Measurements

This final section of results provides validation of the access density metric through the correlation of memory traffic performance when the nursery of a generational garbage collector is mapped to a software-managed scratchpad. Since through the application of access density to the applications of SPECjvm98 we uncovered the trend that small, short-lived objects have the highest locality in many applications, we expect that the nursery of a generational collector would be the most suitable memory region to map to scratchpad.

In our experiments we compare a system with a scratchpad and cache to a system with only cache. In order to make the comparisons as fair as possible we compare systems in which on-chip storage capacity is equivalent. For example, one of the configurations we compare is a system with 512KB cache and 512KB scratchpad to a system with 1MB cache. In both configurations, the caches are configured to have the same associativity and line size. The resulting measurements are for total traffic between cache and main memory.

For this strategy to be effective, we must properly identify and map a region of high locality to the scratchpad or we will significantly hinder overall performance. Consider the worst case in which we reserve half of our on-chip storage space for scratchpad, and then assign

nothing to it. This is equivalent to chopping the cache in half. In order for the scratchpad to provide benefit, we must identify locality well enough to at least match the performance of an equivalently sized cache. This is a demanding goal as cache is capable of representing a very large address space and dynamically adjusting its contents on a line-by-line basis to match access patterns. Scratchpad, on the other hand, is a fixed-sized region with no automatic adjustment of content. All content is managed only by the generational garbage collector and manipulated only periodically, when the nursery is exhausted. If access density had misidentified the locality of the applications in SPECjvm98, we would expect to see a significant increase in memory traffic.

The results, however, do in fact validate access density, and remarkably so. Figure 3.10 contains a graph of the reduction in memory traffic for all of the applications of SPECjvm98 for three different nursery/scratchpad sizes. As the graph shows, all of the applications we identified to closely match the trend that small, short-lived objects have the highest locality (*javac*, *jess*, *mtrt*, and *jack*) uniformly see significant improvement in reduced memory traffic, ranging from 11 to 21%. It is important to note that our access density results from *raytrace* do, in fact, accurately represent the locality found in *mtrt*.

On the right side of the graphs for *mpegaudio* and *compress*, the applications for which we predicted no improvement due to locality existing in large long-lived objects, do in fact generate more memory traffic under the cache/scratchpad system. We would never expect *compress* to see improvement in such a system because its large objects will never be found in the nursery. On the other hand, *mpegaudio* does seem to show continuing improvement as the scratchpad size is increased. Returning to Figure 3.9 and taking a closer look at *mpegaudio*, it's clear that the reason we see improvement is that as the nursery size increases, more of the longer-lived objects will have a longer residency in the nursery and thus reap the benefits of the scratchpad. Although not shown in the graph, we ran another test for *mpegaudio* using a 4MB cache and 4MB scratchpad compared to an 8MB cache and found that a 17% reduction in memory traffic was achievable . While this size is ridiculously large for the application as it allocates only a few megabytes of data total, the experiment does show that if the parameters

of the system are configured to capture the locality of an application, memory traffic can be reduced.

The last benchmark to be discussed is *db*. This application has a behavior that is similar to that just discussed for *mpegaudio*. The graph for *db* in Figure 3.8 shows that locality is distributed in discrete chunks along the life-span of objects. When the nursery is small, not enough of the longer-lived objects are retained in the region to achieve benefits. Even as the nursery size is increased, the marginal benefits achieved are not sufficient to outweigh the benefits the cache receives by being able to handle a larger working set. However, once the nursery is made large enough to begin capturing the locality of a significant number of the objects in the middle of the object life-span, *db* also sees significant improvement using the scratchpad.

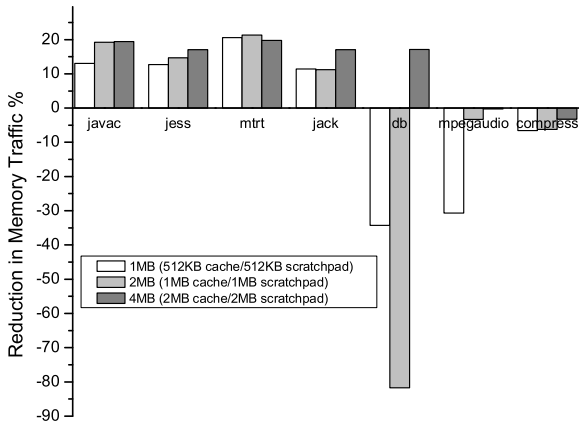

Figure 3.10 Memory traffic reduction vs. equivalent cache size for SPECjvm98 when nursery is mapped to scratchpad of varying sizes.

Through three sets of case studies, applications that match the trend, applications that partially match the trend, and applications that do not match the trend, we provide significant validation of the access density metric. We have demonstrated through both positive and negative examples that access density can identify the locality in object-oriented systems. The identified locality can be exploited and traffic between cache and main memory can be significantly reduced.

3.6 Related Work

We believe we are the first to evaluate locality quantitatively on the individual object level. Since this work covers a broad range of topics including cache metrics, Java application behavior, garbage collection, caching strategies and hardware/software interaction, there is a very large base of related work to consider.

The work that we think is the most closely related overall 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 (47; 76). 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 byte-code 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 collocate 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.

Our work differs significantly in several ways. First, we do not focus solely on embedded applications on existing hardware, but evaluate the use of scratchpad in more general sense as an alternative to cache to reduce memory traffic. Second, we provide a very thorough investigation of the locality that objects exhibit to determine which objects are most suited to allocation in the scratchpad. They focus on the large, long-lived objects based on their decision to statically allocate these objects to scratchpad or relocate objects at runtime using garbage collection. Although we also use static profiling in our work, we do so only to demonstrate the trend we intended to capture. All of our tuning of the system behavior comes from a modification of the virtual machine to map the nursery of the generational garbage collector to scratchpad. None of applications need to be modified in order to work under our scheme. We also incur no additional software overhead above that already inherent to generational garbage collection.

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 (67; 16; 25; 64; 84; 83). 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 (11). We have confirmed these results in our application of access density to individual objects by identifying small, short-lived objects as having the highest locality in many Java applications.

A wide variety of research has been conducted in the area of cache metrics. These range from proposing analytical models for predicting cache behavior (86) to estimating cache performance based on locality measurement (4) and even viewing cache behavior in terms of signal filters (82). Some researchers have used cache metrics either from profiling or from static compiler analysis to assist in program translation to achieve better cache performance (62; 63; 66; 43; 57; 41). Some have analyzed the locality of post cache references (3). We found one work that analyzed the instruction locality of object-oriented programs (7). Another work demonstrates that locality of different data representations can vary significantly and basic locality metrics such as stride have limited applicability (20). One paper discusses analyzing locality of different data types and clustering these types together to get better performance of the memory system (31). Another work suggests an algorithm for predicting the upper performance bound in terms of spatial and temporal locality for a caching system (74). Several works also suggest that locality can vary in different sections of an application and should be handled by separate types of cache (51; 52; 61; 65).

John argues that while a host of locality models have been developed over the years, working set models, LRU stack models, Independent Reference Models, and others, most recent research still quantifies locality in terms of miss ratios (56). She argues that the inter-reference temporal and spatial density functions presented by Conte and Hwu (19) are useful metrics to measure the locality of applications. While these metrics might be useful to measure the locality of a whole program in order to predict its behavior on a real cache, or even provide feedback on how to vary cache parameters to better support a particular program, the metric does not help to identify where the locality in a program exists. This is the fundamental difference between their metrics and our access density metric. Access density allows us to isolate the locality

exhibited by individual objects within the system. Once calculated, objects can be grouped by other characteristics so that exploitable trends can be identified. The metrics proposed by Conte and Hwu and our access density metric are orthogonal, and both types of metrics could be applied to object oriented programs. They simply measure locality in different ways allowing for analysis at different levels.

To the best of our knowledge, we believe our work is unique in its attempt to identify the locality of individual objects in object-oriented systems. We argue that the fundamental paradigm of object orientation naturally lends itself to locality analysis at the object level. Rather than investigate the interplay between a cache organization and the objects in an application, we suggest analyzing the inherent locality in object oriented programs and designing a memory system that is compatible with the locality trends present in those applications.

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 (22; 23; 24; 32).

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.

One final work we'd like to mention is that of Scott Kaplan regarding his use of the bochs emulator for experimentation (44). He used the emulator to capture the behavior of virtual memory for the operation of a full system. Although our work focused on the behavior of the memory system at the cache-level and our environment was developed independently from a later version of bochs (after it had been translated from C to C++), his publicly available modifications offered us guidance in fixing one of the instrumentation features of bochs.

3.7 Conclusion

In this work, we define object locality and present access density as a metric to quantify that locality. We show how access density can be applied and uncover the trend that in many applications, small, short-lived objects have the highest locality. This trend reconfirms previous work that identifies that generational collectors have the best cache performance. We also show how this trend can be further exploited through a direct mapping of the nursery of a generational collector to a software managed scratchpad. This strategy disconnects objects from the standard cache for which there is no good natural mapping. Our simulated memory traffic results confirm the access density metric's ability to measure the locality of objects through both positive and negative examples. The applications that follow the trend see a significant reduction in memory traffic, on the order of 11-21%, while applications that do not follow the trend see an increase in memory traffic. Through this work we show that access density is a useful metric for identifying locality in object-oriented systems that can be subsequently exploited to more efficiently use the memory hierarchy.

CHAPTER 4 Using Scratchpad to Exploit Object Locality in Java

Reprinted from a paper published in
The Proceedings of the IEEE International Conference on Computer Design[1]

Carl S. Lebsack and J. Morris Chang

4.1 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.

4.2 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 (21). 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 find itself in two separate 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 (21). 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 collocated 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 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 4.1. Anything located in the scratchpad will not be located in main memory, and vice versa, unless the software explicitly makes a copy.

Today, processors that contain scratchpad are most often embedded processors. 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 (9). 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.

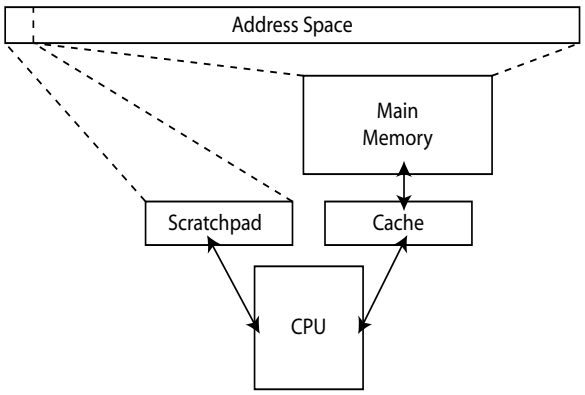

Figure 4.1 System with scratchpad

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 4.3 contains a description of the various tools used throughout the experimentation. Section 4.4 describes the experiments in detail along with their results and an interpretation in relation to our research question. Section 4.5 provides a discussion of related research in the context of our work. Section 4.6 concludes this work.

4.3 Experimental Setup

A diverse set of tools was needed to perform the experiments in this research. This section provides a description of the tools employed as well as a discussion on the construction of the experiments.

We chose to use the applications from the SPECjvm98 (69) benchmark suite in the ex-

periments in this work. 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 (29). For these experiments, we wanted to investigate generational garbage collection, which is 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. Our collector implementation is publicly available in the standard distribution of SableVM.

To gain access to all memory accesses initiated by a CPU, we employ bochs (48), a functional emulator of the x86 architecture. Bochs is an open source project written in C++ 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 (26) as the cache simulator for our work. It has a very straightforward interface and allows easy customization of cache configuration.

4.3.1 Experiments

This section contains a description of the experimentation that was performed to answer the previously stated research question. In order to evaluate generational garbage collection, we needed a JVM which incorporated the algorithm. Although GGC is employed in various forms in several available virtual machines including Sun's JDK and IBM's Jikes RVM, we opted to develop our own implementation to allow for fine-grained control over nursery size and location in memory.

As described in the tools section, we opted to create our own implementation of generational garbage collection in SableVM. SableVM provides instrumentation for reporting statistics such as copying and heap consumption as well as time spent in garbage collection. All timing experiments were run ten times on a Pentium IV machine running Red Hat Linux 9.0 in single user mode. The data reported in most cases is normalized to the fastest run in a series.

The experiments measuring the access patterns to the memory hierarchy are the core support of this work. They validate and provide support of our proposed memory hierarchy configuration (Figure 4.1). We use the bochs emulator to provide memory access information. We developed instrumentation to evaluate the proposed memory hierarchy configuration. Bochs was configured to supply memory access traces which 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.

4.4 Results

4.4.1 Generational Garbage Collection Results

We have previously identified that small, short-lived objects have the highest locality and employ the nursery of a generational garbage collector to capture these objects. What remains is to determine if this method is efficient. In this section, we simply discuss the feasibility of fixing the size of the nursery of a generational collector to that of a reasonably sized scratchpad.

First, we must ensure that the nursery of a generational collector is not 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.

Traditionally, those who research copying collection strategies have focused almost solely on the amount of copying. The assumption is that by reducing the amount of copying garbage collection performs, the overall performance of the application will improve. This is likely based

on the report by Jones and Lins that copying is the largest overhead in a copying collector (40). This has been the basis for the Appel variation to the generational collector as well as Older First collectors (8; 70). While these collectors have been shown to be effective, the issue is not so simple.

The following experimental results demonstrate that there are two opposing trends that are important when considering the costs of a copying collector. The trade-off 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 4.2 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. If copying were the only concern, it would seem that one would always prefer a larger nursery. However, if there were another concern, such as cache locality, we might be better off selecting a smaller size. Proponents of Older First collectors stress the need to give objects plenty of time to die. While providing more time 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 4.2 is that the behavior for each of the benchmarks is relatively consistent 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.

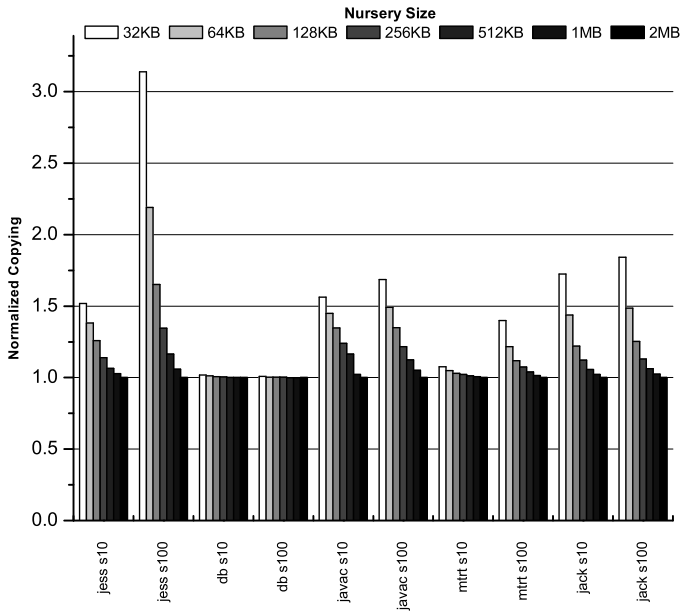

Figure 4.2 Copying vs. nursery size

The graph in Figure 4.3 shows the GC time (normalized to the minimum) over the range from 64KB to 1MB. The normalized GC time is actually larger than the normalized copying found in Figure 4.2 as there is also time spent tracing the root set. A smaller nursery causes a greater number of GC invocations and therefore more time is spent tracing the root set to determine which objects are live. Copying is not the only consideration; the overhead of tracing the root set can also be reduced by using a larger nursery.

The next experiment shows that a larger nursery also puts greater strain on the cache. Figure 4.4 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. We also measured the full execution time for the SPECjvm98 benchmarks on SableVM for a range of nursery sizes. However, as SableVM uses an interpreter which dominates the runtime, the total runtime varied by less than 1% for all of the nursery sizes

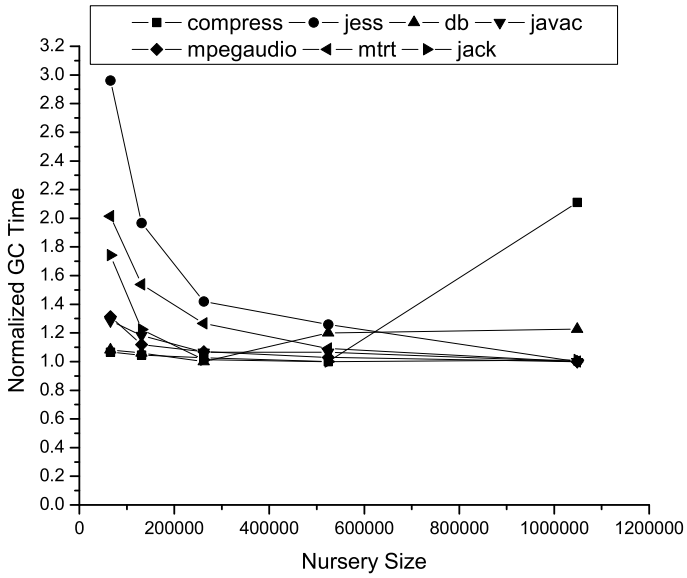

Figure 4.3 GC time vs. nursery size

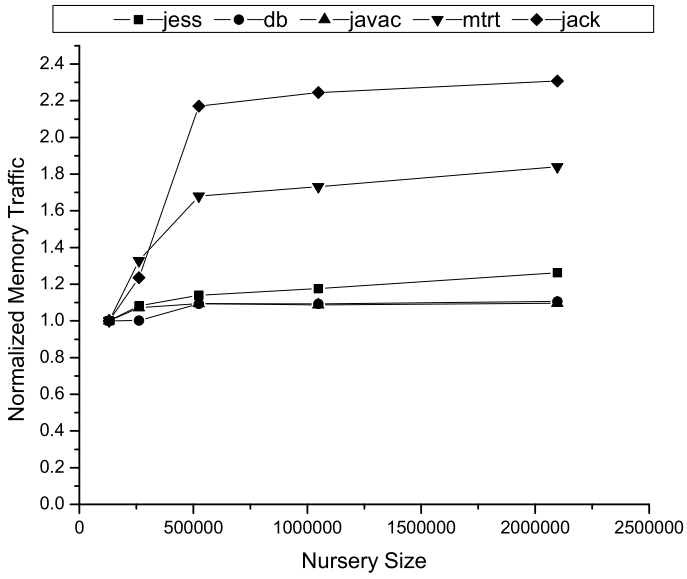

Figure 4.4 Memory traffic vs. nursery size

we tested. A virtual machine employing just-in-time compilation (JIT) would significantly reduce the overhead of bytecode interpretation. However, the accesses to heap objects would be unaffected by JIT. Therefore, we continued our investigation based on the measurements indicating increased cache misses when the nursery size is increased.

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. Therefore, as we are working in the range of a reasonable cache size, we have shown that mapping the nursery of a generational collector to scratchpad can be an efficient method of capturing the locality of the small, short-lived objects in Java programs.

4.4.2 Memory Traffic Results

In this final results section, we evaluate a system with scratchpad and cache versus a cache-only system with regard to memory traffic. This section provides the answer to our research question but also goes further to attempt to identify optimum sizes of both cache and scratchpad.

First, as mentioned earlier, we decided to 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 scratchpad is a more reasonable option. Essentially, we would still see the benefit of having a relatively high number of accesses in the scratchpad, but part of the nursery would also be competing with other memory regions for residency in the cache. Therefore, we expect that having a nursery larger than the scratchpad would provide performance for memory traffic somewhere in between sizing the nursery identically to scratchpad and having just a cache. For all configurations using scratchpad, we use a nursery of equal size.

The first step in evaluating our proposed memory configuration (Figure 4.1) is to determine if employing scratchpad can be more effective than simply using a larger cache. Although we

mentioned above that scratchpad has other 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 local memory are the same size versus a configuration with cache equal in size to the combination of both scratchpad and cache. The plots in Figure 4.5 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.

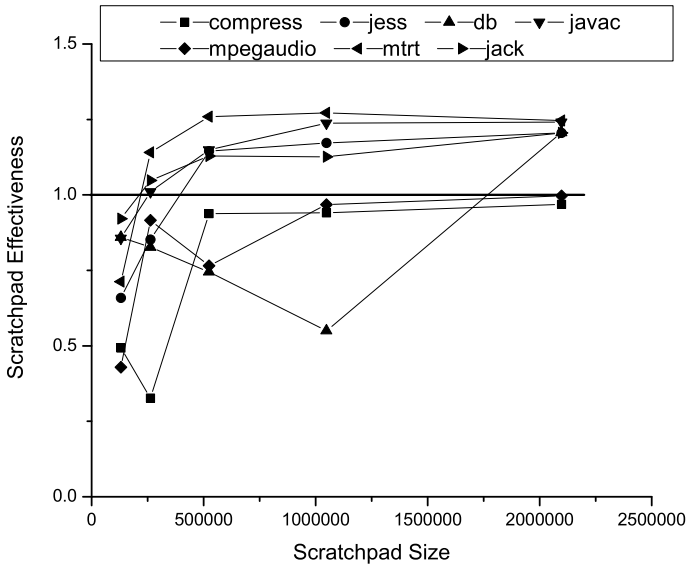

Figure 4.5 Scratchpad effectiveness

As Figure 4.5 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 local memory for sizes greater than 512KB. It's *db* that shows the worst performance overall with local memory, until local memory reaches 2MB. This is because *db* has such a large percentage of long-lived objects that consume a large portion of the accesses to the heap. However, once the nursery reaches 2MB, enough of the accesses to long-lived objects occur while those objects are still in the nursery, and even *db* shows a 17% improvement using local memory over a cache only configuration.

We further investigate the best size for the scratchpad. To make this identification, we fixed the cache size to 512KB and then ran a series of tests that included a local memory which varied in size from 128KB to 1MB. Now that we are investigating the use of a hardware component in conjunction with GGC, we have the additional reason to keep the nursery small, as silicon area is expensive. Figure 4.6 shows the memory traffic normalized to the minimum for the range of local memory sizes. In this experiment, we leave out *compress* and *mpegaudio* because they generally don't see benefits from the scratchpad and don't aid in determining the optimum scratchpad size.

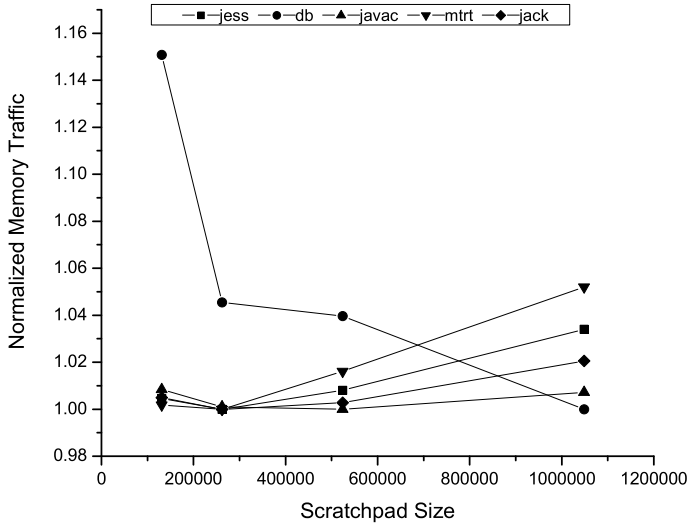

Figure 4.6 Normalized memory traffic for fixed cache size

As the graph shows, the memory traffic varies much less now that we are employing scratch-

pad instead of cache only as shown in Figure 4.4. This is because scratchpad eliminates traffic created by writing dead objects back to memory, as well as traffic generated by allocating objects in regions not accessed recently. Based on this graph, we see that our selection of the scratchpad size will not affect the memory traffic drastically within the tested range. Therefore, in order to decide which local memory size would be best, we must also consider the amount of copying generated within the same size nursery (Figure 4.2). When comparing both of these factors, memory traffic and copying costs, we suggest using a nursery size of at least 512KB.

The final stage of our experiments was to determine which cache size best complements our choice of scratchpad. In this experiment the scratchpad size, and thus the nursery size, was fixed to 512KB. We then tested the benchmarks over the cache sizes ranging from 32KB to 1MB in successive powers of 2. Each test consisted of a comparison between a configuration with and without the local memory. We then plotted the improvement the local memory provided as a percentage in reduced memory traffic. The results can be seen in Figure 4.7. Again we left out *compress* and *mpegaudio* as their interaction with scratchpad does not aid in the selection of the best cache size.

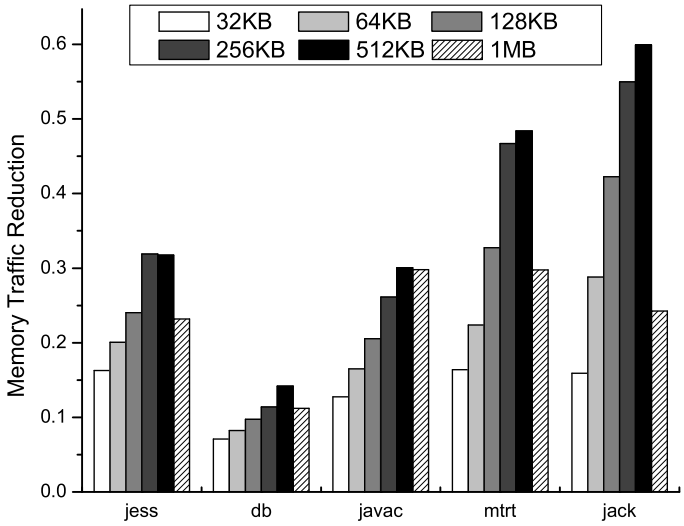

Figure 4.7 Memory traffic reduction using 512KB scratchpad

The best improvement overall when comparing all of the 5 benchmarks appears at the

512KB size. Thus, the best overall configuration for our system, as shown in Figure 4.1, contains a cache and local memory both of size 512KB. The final values for reduced traffic for employing this configuration instead of a system including a single 1MB cache are shown in table 4.1. Also included in the table are comparisons for on-chip resources of 2MB and 4MB. 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 do, continue to see the benefits.

Table 4.1 Reduced memory traffic using scratchpad (Bytes)

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

As table 4.1 shows, not all applications have less memory traffic by using scratchpad instead of having only a cache. The applications that do worse for some configurations are *compress*, *db* and *mpegaudio*. Both *compress* and *mpegaudio* exercise GGC 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 its residency is very small, but it does allocate enough for GGC 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, and thus the scratchpad and *mpegaudio* can perform better on a 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.

In addition to showing that a cache and scratchpad system can outperform a cache-only system there are a few additional important observations. First, note that in Table 4.1 applications that show improvement have a reduction in memory traffic both to and from main memory (write traffic and read traffic). Modern computer systems employ buffers that attempt to hide the latency of main memory, and write traffic can be hidden more easily than read traffic. However, our results show that there is a significant reduction of both kinds of traffic.

Second, 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 4.1 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 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.

4.5 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 (47; 76). 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 collocate 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 (67; 16; 25; 64; 84; 83). 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 (11).

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 (22; 23; 24; 32).

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.

4.6 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 two to four 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.

CHAPTER 5 Cache Prefetching and Replacement for Java Allocation

A paper submitted to
The Journal of Systems and Software[1]

Carl Lebsack[2,3], Quinn Jacobson[4], Mingqiu Sun[5], Suresh Srinivas[5], and Morris Chang[2]

5.1 Abstract

Java programs are memory intensive and exhibit two well known behaviors associated with allocation that affect cache behavior. These include a high allocation rate which we find causes on average 10% to 15% of cache misses and some displacement of other cache contents, and a high mortality rate, which can leave the cache littered with dead objects. We evaluate a combination of hardware and software techniques to improve cache behavior in response to these behaviors. The first set of techniques we evaluate includes hardware and software prefetching. We propose the use of software prefetching to ensure that cache misses are not incurred during allocation. The second technique we propose is a modification to cache replacement to limit the displacement of other data by allowing objects that die to be evicted early through biasing. We evaluate the cache performance as well as the contributions of these techniques on a set of nineteen standard Java workloads. Based on the results of this work, we recommend that software prefetching on allocation be the default Virtual Machine configuration on all platforms. We also confirm the practice of disabling hardware prefetching in server environments where sufficient bandwidth is not available to issue prefetches in a timely fashion amidst demand misses. We also recommend the use of replacement biasing

[1]Submitted for review, January 2008.
[2]Iowa State University, Ames, IA 50010
[3]Work conducted during a graduate research internship with Intel Corporation in 2006.
[4]Nokia Research Center, Palo Alto, CA 94304
[5]Intel Corporation, Hillsboro, OR 97124

in server environments to help improve cache performance when bandwidth constraints limit prefetching effectiveness.

5.2 Introduction

A key limiting factor in performance scaling in the microprocessor industry is the memory bottleneck. The growing speed gap between main memory and processors is a heavily-investigated trend. There are also two recent trends applying additional pressure to the memory subsystem within modern computers: Chip-Level Multiprocessing and the ubiquity of managed software. Chip-Level Multiprocessing is the de-facto standard among the processor manufacturers today. The most important impact of this trend to the memory subsystem is an expected constant cache size per core for the foreseeable future. Software applications are becoming more complex and demanding of system resources. New applications are being developed based on a Managed Runtime Environment (MRTE), such as Java or .NET. The Tiobe Programming Index from November 2006 reports that Java is the most popular language used in development of open-source projects and nine of the top ten languages are object-oriented (75). Many of these languages provide advanced software engineering features, such as garbage collection, that require a managed runtime system (e.g. the Java Virtual Machine (JVM) for Java and .NET for C#). In these runtimes, memory is allocated by the programmer without being explicitly freed. Once the heap is exhausted, the garbage collector is invoked to reclaim unused memory. An automatic memory management framework requires four to five times the memory resources to hide garbage collection costs and match the performance of applications with explicit memory management (35). Even with sufficient main memory, managed applications can spend more than 40% of their time waiting for main memory (1). Software applications written today are, simply put, memory-intensive.

The primary contribution of this work is to provide a much needed analysis of cache performance of Java workloads on today's systems. These systems are extremely complex, comprised of many layers of hardware and software, and make performance of applications difficult to characterize. We attempt to provide key insight into the performance of Java applications

by matching behaviors at the virtual machine level to the behavior of the hardware memory subsystem. In addition to analysis, we propose two techniques – an application of software prefetching for allocation and a modification to hardware cache replacement – to enhance the performance of Java applications.

Object-oriented programs, such as those written in Java, allocate prolifically and most objects die after a very short time (12; 21). Thus, allocation in Java exhibits two behaviors that we attempt to analyze in the context of cache performance. The first is the rapid allocation rate. Research has shown that the most important choice for cache performance in selecting a memory management system is the use of a bump-pointer nursery-style allocator (11). This type of allocation management is found in nearly every performance oriented JVM, both commercial and research. Bump-pointer allocation means that a contiguous region of memory is used for the allocation of new objects. A single pointer is used to maintain the location where the next object is to be allocated. When an object is allocated, the pointer is simply incremented, or bumped, by the size of the object to the address for the next allocation. Variations on the technique only affect the size of such regions and whether the regions are shared or thread local.

In this research, we find that Java applications using bump-pointer allocation in the absence of both software and hardware prefetching can incur on average 21% of their cache misses upon allocation. Because bump-pointer allocation is such a regular pattern of incremental access, we propose to analyze the effects of both hardware and software prefetching techniques. The advantage of evaluating the use of prefetching on Java applications, and specifically allocation, is that allocation is a behavior of all Java applications managed by the JVM. We target the JVM for hand-tuned prefetching enhancements because the benefits are propagated in various degrees to all Java applications run in the managed environment.

The second characteristic Java applications exhibit is the high mortality rate of objects. In addition to the misses incurred upon allocation, the rapid allocation rate also has the potential to displace useful data from the cache. The objects that die quickly can occupy the cache even after they die, consuming resources that could be utilized for active data. At the L2 cache level,

the rapid allocation and short-term reuse is very similar to a streaming pattern. In this work we find that the L1 cache filters the short-term temporal locality and, on average, 19% of the lines in an L2 cache are only ever accessed once. In some Java applications the percentage can be as high as 45%. Techniques for handling poor temporal locality data have been proposed. Some modern processors have software prefetch instructions that allow the specification of poor locality, such as the *prefetchnta* instruction from the x86 ISA. In the research community it has been proposed to leave prefetched data in the LRU position in the cache to prevent aggressive prefetching from displacing cache contents (55). There have also been suggested modifications to the cache replacement policy to account for temporal accesses (85). We propose a new replacement policy that targets newly allocated objects for early eviction, which both offers greater flexibility than leaving data in the LRU position and is also easy to implement within existing cache architectures. We subsequently evaluate the effectiveness of this technique in increasing cache performance for Java applications.

After evaluating these three techniques, hardware prefetching, software prefetching and replacement biasing, we make the following recommendations for the following situations: We confirm that a server application is capable of saturating available bandwidth on a four core shared cache architecture and that hardware prefetching can actually degrade performance. In this situation we recommend the common practice of disabling hardware prefetchers. We find that, in client environments, hardware prefetchers provide substantial performance improvements, from 25% to 55%, depending on cache size and available bandwidth. We recommend that software prefetching on allocation always be employed in the Java Virtual Machine. When hardware prefetching is unavailable, software prefetching can decrease miss rates by an average of 10% to 15% which equates to an average of 20% to 25% performance improvement, depending on cache size and available bandwidth. In some cases, software prefetching may provide negligible additional benefit over hardware prefetching, but no degradation occurs. Finally, we recommend that cache replacement biasing be used in server environments where there is potential for bandwidth saturation. Cache replacement biasing can offer an additional 5% improvement over software prefetching alone.

The rest of this paper describes the prefetching and replacement techniques and the experimental evaluation in the context of Java workloads. Section 5.3 describes in detail the techniques investigated as well as related work. Section 5.4 discusses the experimental methodology and framework used in the evaluation, while Section 5.5 provides the results of that evaluation. Section 5.6 further discusses related work and Section 5.7 concludes the discussion.

5.3 Cache Management Techniques

Cache is the hardware component most specifically targeted to memory system performance. Application performance is closely tied to the ability of cache to predict program locality through prefetching and capture program locality through reuse. Java applications rely on dynamic allocation to a managed heap for all created objects. This dynamic allocation constitutes the vast majority of memory usage in Java applications. In this work we evaluate the cache performance of Java applications specifically with respect to allocation, both in terms of hardware and software prefetching and in cache replacement and reuse.

5.3.1 Hardware Prefetching

The first set of techniques we investigate is prefetching mechanisms. There is a long evaluation history of both software and hardware prefetching which includes a good survey by Todd Mowry in his PhD thesis (59). More recent work includes prefetching specific to Java for both arrays and linked data structures (1; 14; 15). In this work we focus specifically on allocation in Java. Because of the expected regular incremental pattern of accesses associated with allocation, we would expect that hardware prefetching based on a unit-stride prefetcher should provide a substantial reduction in misses for Java applications. Bump-pointer allocation induces a linear progression through a portion of the address space. A hardware unit-stride prefetcher detects misses in access patterns that sequentially span a region of the address space. We expect that such a hardware prefetcher can adequately capture the allocation behavior in Java. In this work we evaluate a hardware prefetching scheme, named RPT, capable of detecting unit stride patterns as well as single and multi-stride patterns (38). The addition of single

stride and multi-stride pattern detection adds enhanced capability in identifying more complex access behaviors of applications. We choose RPT because it is purported to be representative of state of the art hardware prefetchers found in practice. We extend the work by evaluating the performance of Java applications with regard to hardware prefetching.

5.3.2 Software Prefetching

Hardware prefetching will not isolate allocation patterns from the rest of the accesses within a Java application. To determine the contribution of misses from allocation we also include software prefetching. We propose a software prefetching scheme that prefetches ahead of allocation to specifically capture all potential misses incurred on allocation. For our software prefetching scheme we initiate prefetches for all the lines within a block, one full block ahead of allocation. This procedure can be implemented with current prefetch instructions issued in a loop, assuming the next block is in physical memory as prefetches will not induce page faults. When allocation crosses a block boundary, all of the lines in the next block will be prefetched as the current block will already have been prefetched by the previous boundary trigger.

The prefetch offset and prefetch size are tunable, but in this work we focus on a 4KB page size offset and block size. Our work has shown that the ideal size of the block and offset is 256 bytes as sizes smaller do not capture all allocation misses, and larger sizes do not offer significant further performance benefit. However, we propose that the technique can be expanded to the 4KB page size with hardware support for a block prefetch. The first reason is that this arrangement still yields greater than 99% prefetching accuracy which indicates that nearly all lines are used after they are prefetched and before they are evicted by other accesses. Also, no degradation in cache performance occurs, meaning that the prefetched data is not displacing other useful data. We propose, based on these two observations, that a block prefetch instruction added to the ISA could be used effectively for Java allocation. Mowry (59) mentions the approach of bringing in multiple lines with the issue of a single prefetch instruction. At the time of his investigation he cautioned against cache pollution, which, as the size of L2 caches has become large, is less of a concern. Block prefetching has also been

promoted in the networking area where large regular-sized data structures are used in the buffering and processing of data packets (87).

Block prefetching, as we propose, would prefetch whole pages of data into the L2 cache, one line at a time, with a single software instruction. We argue that the implementation of such an instruction is trivial in the presence of a hardware prefetcher. Current hardware prefetchers maintain a table of streams for which they are predicting and issuing prefetches. A software initiated block prefetch would need a similar small hardware table such as the 32-entry table we evaluate. The table would contain the base address of the page and a counter to track which line is next to be prefetched. Hardware prefetchers wait for idle periods on the memory bus and initiate any available pending prefetches. Supplying an instruction to initiate a block prefetch can induce prefetching for a large number of lines without the need to loop over individual line prefetch instructions, which consume slots in the instruction pipeline as well as entries in the load queue.

Our proposed scheme of software prefetching on allocation was first implemented in BEA JRockit and has subsequently been implemented in IBM's J9 as well as Sun's JDK. In this work we evaluate both hardware prefetching and our proposed software block prefetching as well as the combination with respect to allocation in Java applications.

5.3.3 Cache Replacement Policy

In addition to prefetching, we also investigate modification to the cache replacement policy. Because objects in Java have a high mortality rate, the same address space we attempt to prefetch for allocation is likely to have a very short usefulness. We investigate the possibility of treating the cache lines prefetched for allocation in a different manner than the rest of the cache. Cache lines containing newly allocated objects are likely to have a short life-span, but we do not want them to be replaced immediately. We also would prefer that they not be promoted all the way to the top of the LRU stack to the MRU position. Our approach is to place data in the middle of the stack above the LRU position, but not in the MRU position. The prefetched allocation region should not be replaced before it is accessed, but once used,

should not be retained much longer.

Our proposed cache replacement modification is based on an existing common hardware implementation of pseudo-LRU replacement. We chose a pseudo-LRU replacement policy as it can be efficiently implemented in real hardware, as opposed to the full LRU algorithm which has no efficient real implementation. This pLRU cache replacement policy is the Tree pLRU algorithm which tracks Most Recently Used (MRU) status by performing updates to the corresponding nodes in a tree structure.

The tree algorithm is not a strict LRU ordering and hence only an approximation. Each access will cause an update to all nodes in the path starting at the root and terminating at the leaf node corresponding to the line accessed. **Figure 5.1** shows an example where line 3 is accessed and causes the nodes in the path to be updated with the values (0,1,1) where 0 denotes the access occurs in the left branch and 1 the right. The state of the tree after the access reflects that 3 is indeed the most recently used line. As nodes are shared between lines, there is no way to tell what line is the second most recently accessed and could be either line 0, 2 or 6. This hardware policy treats all accesses equivalently and updates all levels in the tree for every access.

Since the LRU ordering in the tree algorithm is not strict and only the MRU status is encoded completely, the key becomes identifying what line should be evicted when replacement is needed. Upon the need for eviction, the tree is traversed, starting at the root, using the inverse of the node values to select a victim that was not recently used. In the example in **Figure 5.1**, line 4 would be selected for eviction based on the tree status shown. Line 4 may in fact be the least recently used, but as the tree is an incomplete encoding, the least recently used could also be 1, 2 or 7. While the absolute position is not precise, this encoding does guarantee that line 4 is in the older half of the LRU stack.

Our proposed alteration is based on the assumption that not all accesses are equivalent. In the presence of priority among lines we want the hardware to bias towards victimization of lower priority lines. We assume priority is an attribute of the individual access. A highest priority access performs a full update to the tree structure, as in **Figure 5.1**. A full update

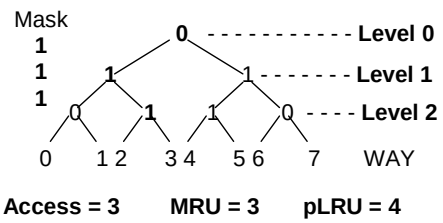

Figure 5.1 Tree pLRU

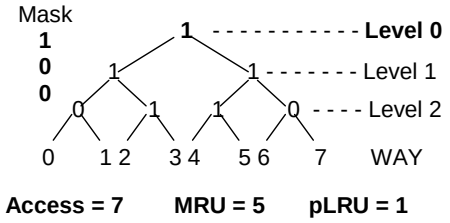

Figure 5.2 Partial update tree pLRU

does not mask out any of the bits of the tree pLRU update (the mask in **Figure 5.1** is all ones). A lower priority access will be restricted to a partial update to the tree as in **Figure 5.2**. The partial update consists of a mask where at least one of the bits in the mask is set to zero, preventing an update of the corresponding node level in the tree. In hardware, the mask can be combined, via a logical and operation, with the write-enable bits corresponding to individual node levels within the tree. A partial update will ensure that a recently accessed low priority line is not victimized immediately because it will update at least one node in the tree, but the line is not promoted all the way to MRU status.

We consider a partial update to be a masked version of a full update. Each bit is allowed to perform an update when set to one, or prevented from performing an update when set to zero. For a sixteen-way set-associative cache, there are four levels to the tree corresponding to a four bit mask. A full update would be an update with a mask of four ones (binary 1111, or decimal 15). A partial update of only the most significant bit would use a mask of a single one and three zeros (binary 1000, or decimal 8) and an update that was not allowed to update any levels would use a mask of four zeros (binary 0000, or decimal 0). We assume the default priority is a full updat,e and all biasing is negative biasing increasing the likelihood of earlier eviction. The difference between a mask of four zeros and leaving a line in the LRU position is that the mask can be used at any time. When a line is first prefetched it can receive a mask of eight (binary 1000), placing the line above the LRU position. A subsequent access with a mask of zero (binary 0000) does not place the already present line in the LRU position but rather retains the present ordering.

We evaluate this modified replacement policy by utilizing the bias masks on data that is

prefetched for allocation because we expect the data is likely to have poor temporal locality and thus should be evicted shortly after being accessed. We show results from biasing all prefetches in both software only and hardware only prefetching schemes. We also show biasing all prefetches when both hardware and software prefetching are employed. While the description above offers a degree of flexibility in the manner in which bias is introduced, we select one standard scheme for all replacement biasing in this paper, which could be directly implemented in hardware for hardware and software prefetches. We use a partial update of eight (binary 1000) when a prefetch is first brought into the cache. This mask essentially reserves one eighth of the cache for prefetching. If a continuous stream of prefetches were issued, since only the root node in the tree is ever updated for prefetches, only two lines would ever be used, one on each half of the tree. We use a second update mask of zero (binary 0000) when a prefetched line is first used. No update is equivalent to leaving a line where it resides in the LRU stack. This policy maintains that prefetched data will only be able to displace one eighth of the cache even when the data is accessed once. Subsequent accesses to the prefetched data will promote the line to MRU, thus cache lines with long-lived data will not be forcibly evicted from the cache if they continue to receive accesses.

One eighth of the cache for the cache sizes present today is a substantial size capable of capturing the life-span of a significant number of objects in Java workloads, as it has been reported that 80% or more of objects in Java are no longer live after 100KB of further allocation (21). The technique allows for partitioning of one sixteenth (no updates, leaving lines in LRU), one eighth, one fourth and one half of the cache. Although it has also been reported that converting half of the cache resources to scratchpad for allocation can reduce memory traffic (49), our technique offers the flexibility of using a smaller percentage of the cache resources for allocation. One eighth most closely matches the life-span of Java objects and this is the size for which we report results.

The amount of hardware needed to introduce the biasing on the basis of masks is negligible with regard to the size of the cache. In fact, modern caches already employ a bit per line to denote whether the line was prefetched. This bit is used to reinforce the pattern detection

for hardware prefetchers with feedback of used prefetches. The bit is also used on processors that include performance counters to report cache statistics. In such a system, the only new hardware introduced is the mask logic itself, which is part of the logic used to update the tree status. For a 16-way associative cache, the two four-bit masks will require only the addition of fifteen four-to-one multiplexors for the write-enable bits for the nodes in the tree structure. A separate tree is maintained for each set, but the update logic can be shared.

5.4 Experimental Framework

In this section, we explain the tools and methods used in the evaluation of prefetching and cache replacement mechanisms. The software stack consists of several components. The highest level consists of the Java benchmarks which include SPECjvm98 (69), DaCapo(beta-2006-10) (12) and SPECjbb2005 (69). SPECjvm98 and DaCapo are both client application suites and in our experiments we use them with default workloads. SPECjbb2005 is a server based middleware workload. For SPECjbb2005, we modify the configuration from the standard reportable setup to force separate workloads of one and four warehouses which are referred to as *jbb1* and *jbb4*, respectively, in the results section.

We select BEA JRockit (10), a publicly available commercial grade Java Virtual Machine, for our work. JRockit contains a sophisticated JIT engine, a high throughput parallel garbage collector and full class libraries. The memory manager allocates new objects contiguously with bump-pointer allocation in per-thread allocation regions. Software block prefetching is simulated as an additional instruction during the allocation process. Each time object allocation crosses a page boundary, a block prefetch is issued one page ahead of the current allocation. We run the JVM on Windows Server 2003 supporting configurations from one to four processors. Windows is allowed to initialize and run for a warm-up period before applications are launched.

We use SoftSDV (78), an environment that will run a full operating system and associated applications and emulate the necessary hardware. SoftSDV supports simulation of multiple processor cores and can produce full memory access traces. We use SoftSDV to record memory

access traces of benchmark applications for subsequent processing by a cache simulator. In all of our simulations, SoftSDV is configured to emulate an Intel Xeon style IA32 architecture with 2GB of main memory. We perform simulations of one, two and four processors.

We developed a trace-based cache simulator for this work. The simulator supports complex multilevel cache hierarchies for multiprocessor shared cache environments. Caches can vary in size, line size and associativity. Caches can either be private with coherency maintained by the MESI protocol, or shared. The cache simulator supports replacement policies including the basic tree pLRU replacement policy and our proposed modified version to handle temporal locality. The simulator also supports hardware prefetching based on the RPT prefetcher (38) and software block prefetching. RPT is purported to be representative of the state of the art, practical and low cost. It detects unit stride, single stride and multi-stride patterns based on cache misses and accessed prefetches. It requires only a small table and a simple state machine to detect patterns with very high accuracy.

In our experiments we model a two-level cache hierarchy. The L1 cache parameters are set to the following: 32KB, 64 byte lines, 8 way associativity, and tree pLRU replacement. In a multiprocessor environment, each processor has its own private L1 cache and coherency is maintained through the MESI protocol. The L2 cache parameters in this work are subject to several variations. The parameters that do not change are 64 byte lines and 16-way associativity. We also performed evaluation of 8-way associativity but found no substantial differences in results. The L2 cache ranges in size from 512KB to 8MB and is shared by up to 4 processors. In our work we report data with and without hardware prefetching. We report numbers for both the original tree pLRU replacement algorithm and our proposed biased tree pLRU algorithm.

For all cache simulation experiments, a trace of three billion memory references is recorded for the program and subsequently fed to the cache simulator. The first one billion accesses are used to warm the cache hierarchy and the next two billion are used in the collection of statistics. This trace size roughly corresponds to three seconds of runtime on real hardware. For most of our applications, this constitutes 10% or more of the total execution. We restrict the size of the

traces only to conserve storage space. We ran some traces to completion for comparison and found no significant difference in results. SPECjbb2005 is the notable exception with respect to runtime. As mentioned previously, we have modified the properties file to select a single warehouse configuration (either one or four warehouses) at a time and record the trace starting from the timed portion of the throughput measurement.

To provide performance projections, the cache simulator supports a performance model based on the Epoch model of memory level parallelism (17; 18). The Epoch model assumes that all non-dependent loads can be issued together and complete simultaneously. This assumption simulates the memory read overlap achievable on an Out-of-Order architecture. The Epoch model ignores non-memory instructions and memory instructions that have hits in on-chip caches, assuming that these latencies will be hidden within the latencies associated with main memory loads. We expand the Epoch model to also incorporate bandwidth limitations. Bandwidth is consumed by both misses to main memory and prefetches. We model the bandwidth limitation as an additional dependency within the load stream. When a particular window has been filled to capacity with non-dependent loads the loads are issued and a new window is created regardless of the presence of a dependent load. The bandwidth value (window size) is in terms of the number of memory reads that can be overlapped within the latency of a single miss. Table **5.1** shows the bandwidth available on two state of the art processors and is calculated from benchmark data published on Tom's Hardware (77). We model four different bandwidth values (4, 8, 10, 16) to evaluate performance sensitivity to bandwidth. Our parameters are chosen to be close to those available on current hardware as well as to include values significantly above and below. We assume that bandwidth is constant regardless of the number of cores as bandwidth in SMP systems is a function of the memory bus and does not necessarily scale with the number of cores.

Table 5.1 Memory parameters for select processors

Processor	Frequency	Latency	Bandwidth	MLP Bandwidth
AMD Opteron 285	2590.3 MHz	60.8 ns 160 cyc	9498 MB/s	10 reads/miss
Intel Xeon 5160	3000.0 MHz	95.5 ns 285 cyc	5909 MB/s	9 reads/miss

For memory intensive workloads, a memory performance model such as our modified Epoch model provides a reasonable estimate of execution time. This model breaks down as an application becomes compute bound and memory latency can be completely hidden. We assume the workloads in our investigation are reasonably memory bound and thus an MLP performance model is applicable. This assumption is based on work showing that Java application performance is in fact sensitive to cache misses and, thus, memory bound(1). The Epoch model requires a full dependence chain to ensure proper ordering of dependent loads. This dependence chain is produced from a performance model plug-in to SoftSDV that tracks memory load dependencies through all register operations. The trace is recorded as a stream of addresses with the addition of dependence information.

5.5 Experimental Evaluation

In this section we provide an evaluation of the three techniques – hardware prefetching, software prefetching and biased cache replacement – and their relative impacts on cache misses. We also provide an evaluation for the techniques as they are combined in several variations and include other cache metrics including accuracy, coverage and read traffic. We also include analysis of a performance model that provides insight into memory performance when bandwidth is taken into consideration. Based on the analysis provided, we offer several recommendations.

5.5.1 Hardware Prefetching for Java Applications

The first evaluation we perform is a measurement of the base miss rates of our Java applications as well as the miss rates when hardware prefetching is enabled. The results from this experiment are found in **Figure 5.3**. These miss rates are absolute and can be used as a reference for comparison with other works. The remainder of the results in this section will be presented as miss rates relative to the base miss rate of the application. Miss rates are presented for L2 caches of 512KB, 1MB and 2MB and are calculated as misses per memory operation. While the miss rates are fairly low, they can contribute significantly to runtime performance because of the latency of main memory. Nearly all applications see significant

miss rate reductions with hardware prefetching enabled. On average, the reduction is 30% to 50% with an increased effectiveness as cache size increases. This can be explained by the reduced conflict/capacity misses achieved by the increased cache size, as the absolute contribution of prefetching is fairly constant. In nearly all cases, hardware prefetching can yield a greater reduction in misses than simply increasing the cache size, a case not as common in other workload types. Hardware prefetching should be employed for Java client workloads.

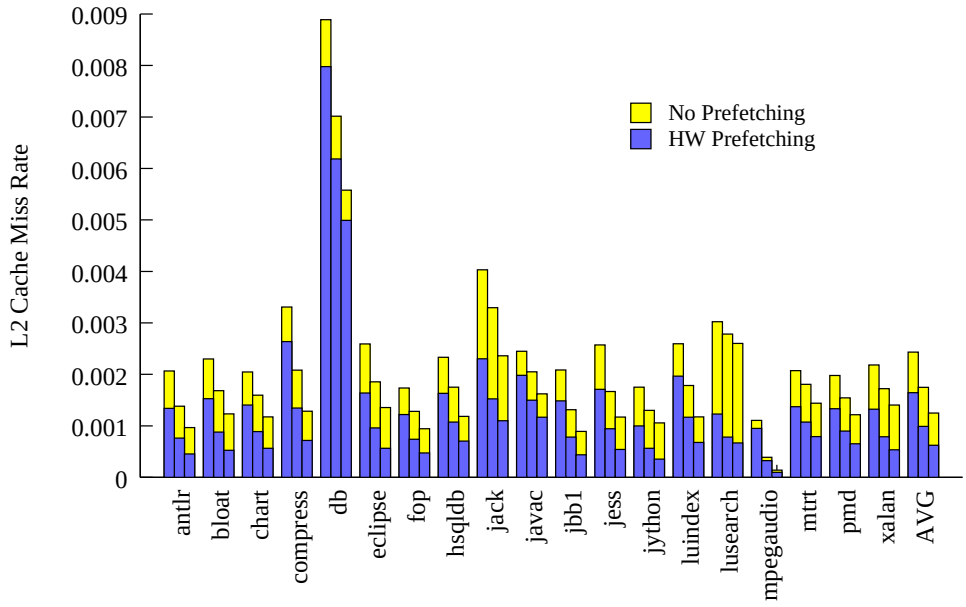

Figure 5.3 Absolute miss rates for HW prefetching in Java applications with 512KB, 1MB and 2MB L2 caches

5.5.2 Software Prefetching on Allocation for Java Applications

The next evaluation incorporates software prefetching in comparison to hardware prefetching. The graph in **Figure 5.4** shows the miss rates for applications with software prefetching, hardware prefetching and the combination of both relative to the miss rates with no prefetching. The three bars for each benchmark represent the cache sizes 512KB, 1MB and 2MB respectively.

Several key observations can be made from **Figure 5.4**. Allocation can contribute significantly to cache misses in the absence of prefetching. With the addition of software prefetching

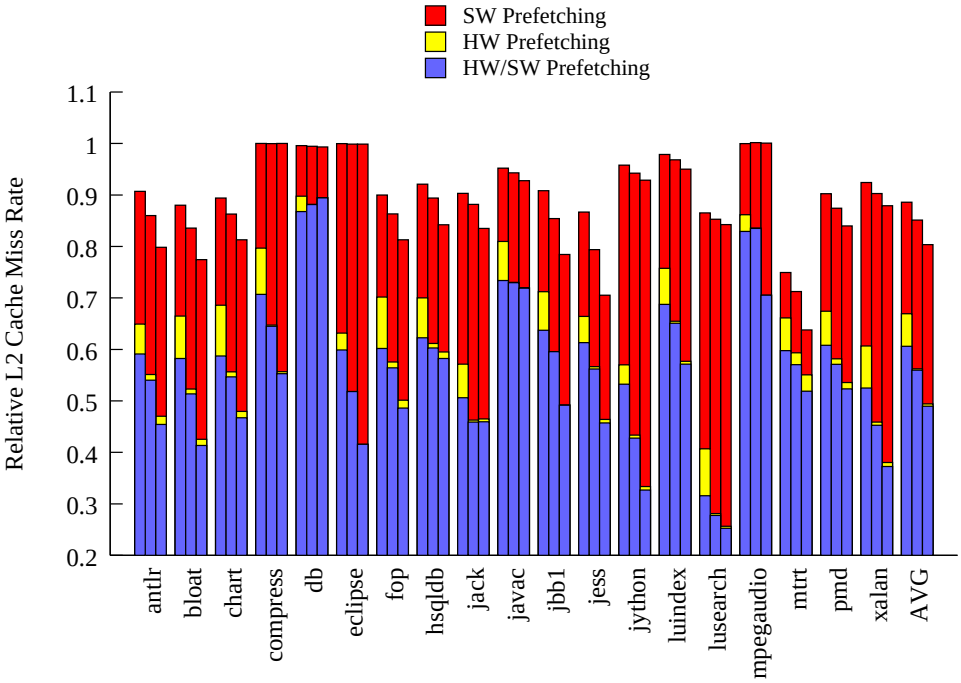

Figure 5.4 Relative miss rates for HW and SW prefetching in Java applications with 512KB, 1MB and 2MB L2 caches (baseline is without prefetching)

only on allocation, cache misses can be reduced by an average of 13% to 21% depending on cache size. Hardware prefetching is capturing a substantial number of misses that are not caused by allocation with an average of 34% to 50% reduction. While hardware prefetching does not catch all of the allocation misses, as evidenced by the 0.5% to 6% additional improvement offered by the addition of software prefetching for allocation, it does capture a significant portion of them, and is more successful as cache size is increased. In a system where hardware prefetching is not available or disabled (server environments), software prefetching can offer substantial benefits. When hardware prefetching is available, software prefetching can offer marginal improvements.

It is also apparent that allocation does not incur a substantial number of misses in all Java applications (*compress*, *db*, *eclipse* and *mpegaudio*). Even though software prefetching on allocation does not always substantially reduce cache misses, it never incurs a penalty and in many cases it does yield benefits. We recommend that software prefetching always be enabled

as the default configuration within a Java Virtual Machine.

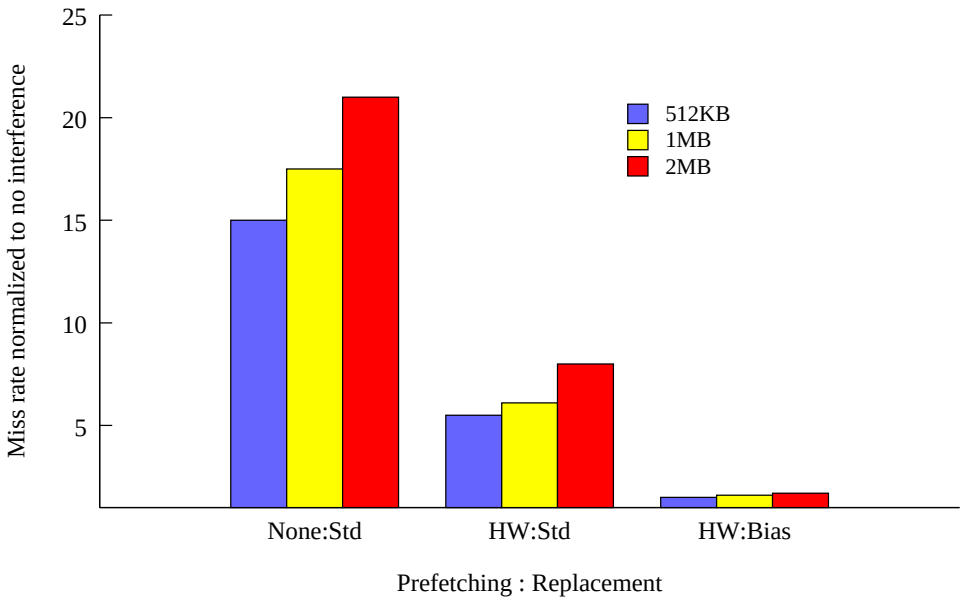

Figure 5.5 Biased replacement effect on disruptive interference

5.5.3 Biased Cache Replacement for Java Applications

Before showing the addition of our cache replacement biasing technique to the Java workloads directly, we first illustrate an example of how the technique works. The goal is to reduce the disruption of data within the cache that will likely be used again by restricting data that is not likely to be reused to a subsection of the cache. To show that our implementation is an effective means of isolating an application from being displaced by a large number of cache lines, we wrote an example program designed to continuously access a pattern of an address space considerably larger than the L2 caches. We ran this program in an environment with two processor cores sharing an L2 cache. On the other core, we ran benchmarks from our suite. We used biased software prefetching for the disruptive application. **Figure 5.5** shows that our application is able to significantly interfere with our original applications in the absence of prefetching by causing miss rates to increase fifteen to twenty times. Even in the presence of hardware prefetching, our benchmark applications incurred more than five times as many misses because of displacement. However, when the disruptive cache loads are biased for ear-

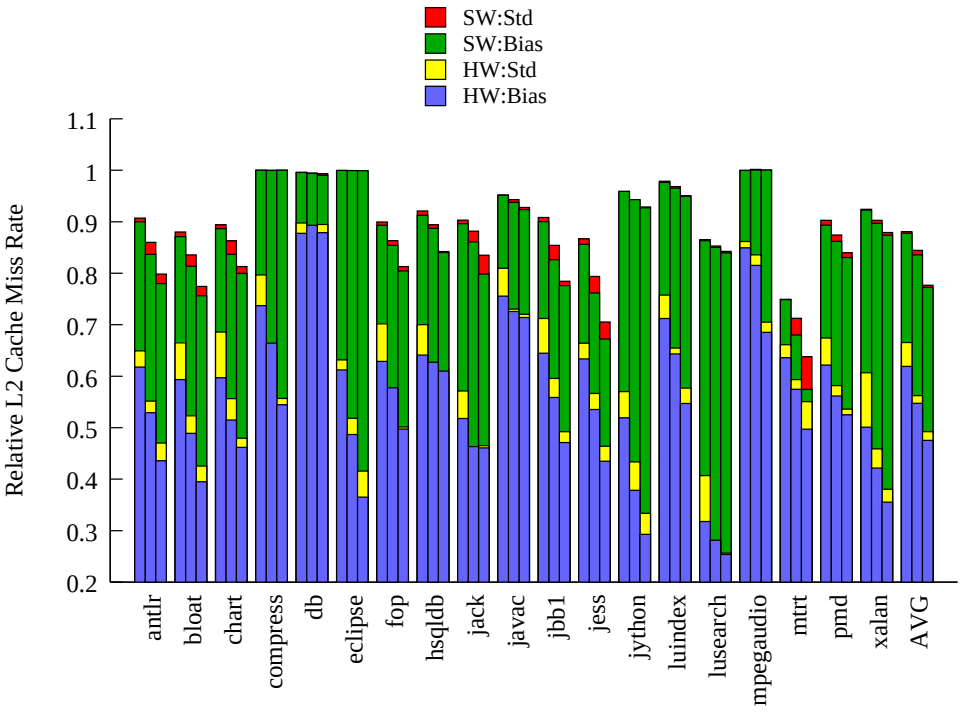

Figure 5.6 Relative miss rates for replacement biasing of HW and SW prefetching in Java applications with 512KB, 1MB and 2MB L2 caches

lier replacement by being restricted to one eighth of the cache (performing a partial update on only the root node in the tree pLRU structure), the disruptive interference is reduced and benchmark miss rates were less than twice that of no interference.

We suspected that the allocation in Java applications might be a source of significant data displacement from the L2 cache, but biasing results yield only marginal improvement. The graph in **Figure 5.6** shows the effect of biasing the replacement of either all software or all hardware prefetches. Biasing in this regard shows an additional average reduction of 0.3% to 0.9% for software prefetches and 1.5% to 4.5% for hardware prefetches. While the average improvement is small, some applications (*bloat, chart, lusearch, mtrt, xalan*) are improved by more than an additional 5% in cache sizes likely near the respective working set sizes. The graph in **Figure 5.7** shows an additional scenario when both hardware and software prefetching are combined with the addition of replacement biasing. The combination of both hardware and software prefetching has the lowest miss rate of the prefetching configurations. The average miss rate can be reduced by an additional 0.7% to 1.8% by biasing all prefetches. When biasing is used exclusively for software prefetches, the cache lines expected to be most suitable for biasing, the average miss rate can be reduced by an additional 2.0% to 2.5% depending on cache size. Overall, cache size seems to be less important with respect to biasing. These results also indicate that the aggressive allocation in Java applications does not displace a significant amount of other useful data from the cache when run in a single core environment.

5.5.4 Memory Read Traffic

One additional key observation that should be noted is the impact of prefetching schemes on memory read traffic. Prefetching offers opportunity for increased performance by consuming additional available bandwidth on the memory bus in order to hide latency. **Figure 5.8** shows the increase in read traffic for the prefetching and replacement biasing schemes. Schemes that include hardware prefetching yield the largest increase in memory traffic. Software prefetching generates almost no additional traffic. These results help to confirm the practice of disabling hardware prefetching in server environments where bandwidth conservation is an objective.

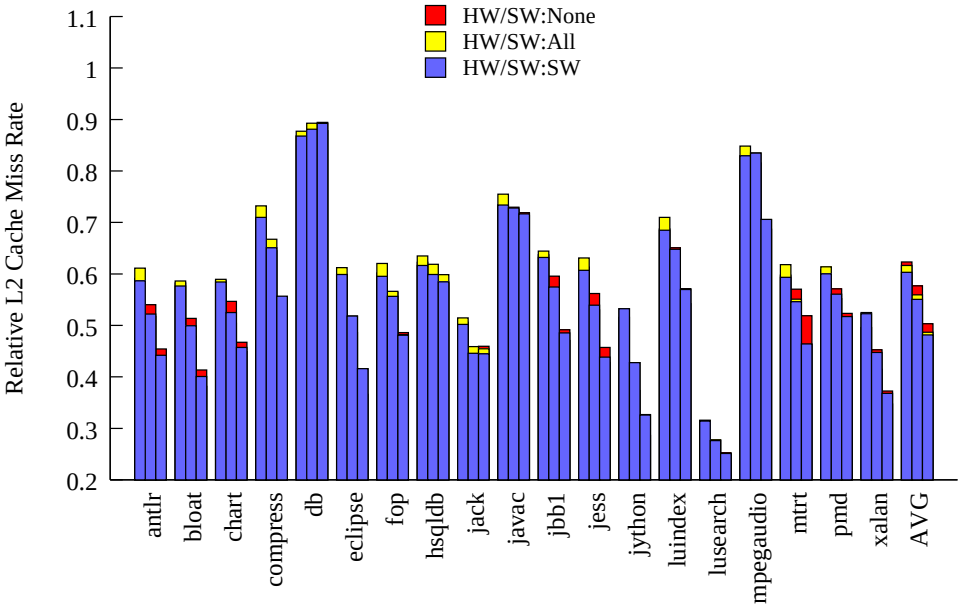

Figure 5.7 Relative miss rates for replacement biasing of combined HW and SW prefetching in Java applications with 512KB, 1MB and 2MB L2 caches

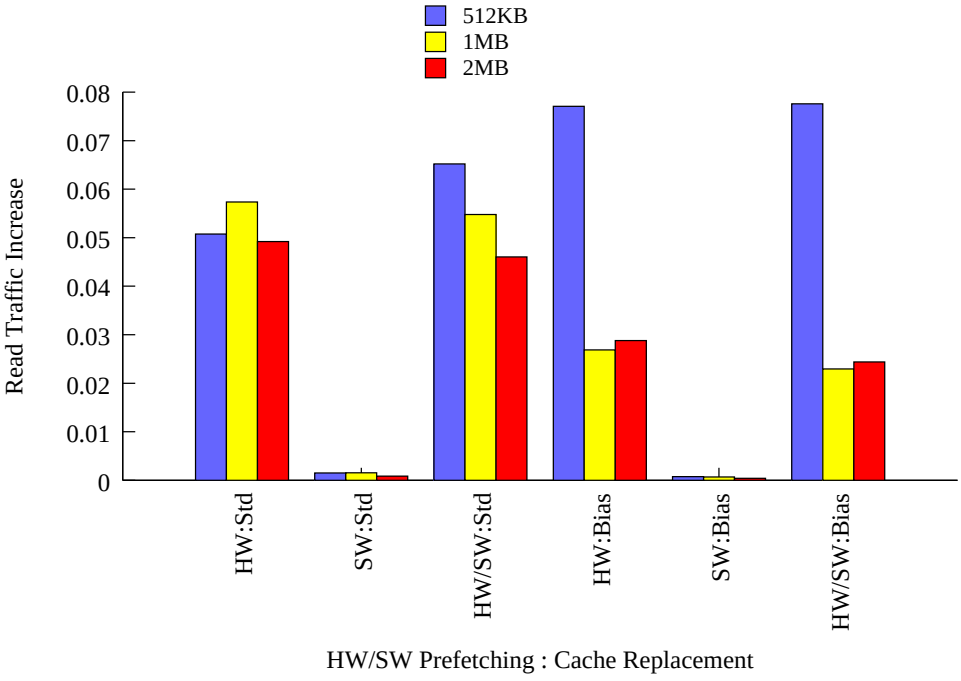

Figure 5.8 Increase in memory read traffic of various techniques for Java applications with 512KB, 1MB and 2MB L2 caches

Biasing limits the increase in read traffic for cache sizes 1MB and 2MB, while further increasing the traffic for 512KB cache. This indicates that the smaller cache is more likely to victimize biased prefetches before they are used, requiring some useful prefetches to be read from memory twice in close succession. Based on these observations, biasing would appear to offer potential benefits in server environments.

5.5.5 Prefetch Accuracy

The effect of biasing can also be seen with regard to prefetch accuracy, the ratio of useful prefetches to issued prefetches. **Figure 5.9** shows the prefetch accuracy for the various techniques presented. Software prefetching has a very high accuracy when compared to hardware prefetching because of the predictability of allocation. Also, the addition of biasing does slightly decrease the accuracy as prefetched data are more likely to be evicted before use. Accuracy is affected by replacement biasing to a greater degree in smaller caches. Replacement biasing does not seem to greatly affect the accuracy of software prefetching or the read traffic needed to issue software prefetches. Accuracy increases as cache size increases because a larger cache reduces the conflict misses that occur. Fewer conflicts reduces the likelihood that a useful prefetch will be evicted before it can be used. The extremely high accuracy of software prefetching for allocation encourages its use in any environment. With near 100% accuracy, the technique has almost no overhead in wasted bandwidth even when prefetching 4KB pages.

5.5.6 Prefetch Coverage

In addition to accuracy, prefetch coverage (the ratio of misses covered by prefetches) is another common metric used to evaluate prefetching. **Figure 5.10** shows the prefetch coverage of the prefetching and cache replacement schemes. Software prefetching, while having a very high accuracy, has very low coverage. Allocation contributes only partially to the overall cache misses in Java applications, as shown by the miss rate reduction comparison in **Figure 5.4**. Coverage of hardware prefetching is much larger, but has lower accuracy and leads to the expense of increased memory read traffic. Replacement biasing has little impact on coverage

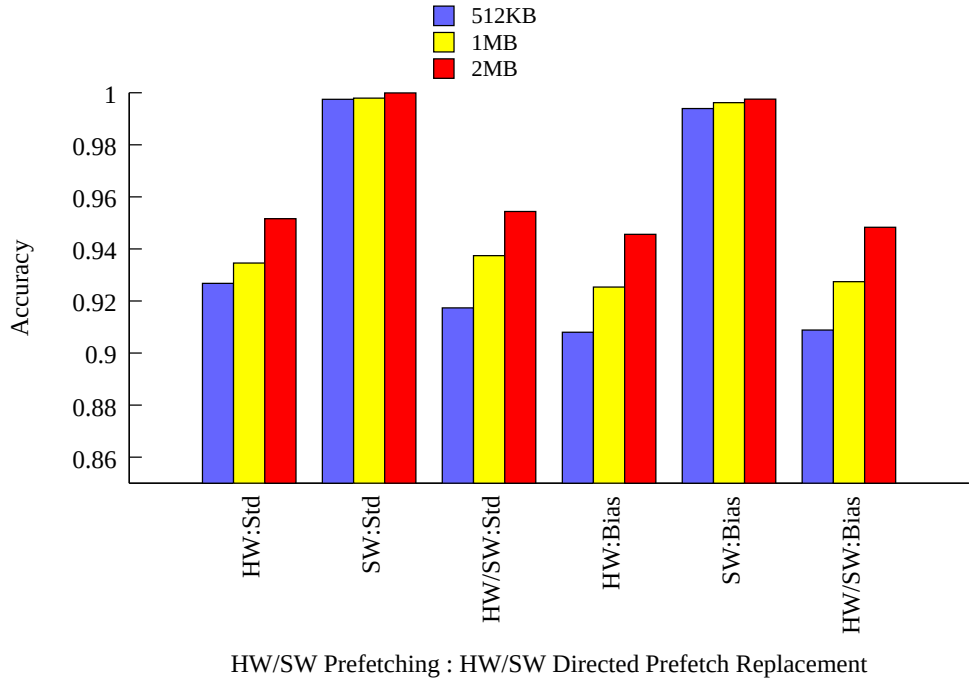

Figure 5.9 Prefetch accuracy of various techniques for Java applications with 512KB, 1MB and 2MB L2 caches

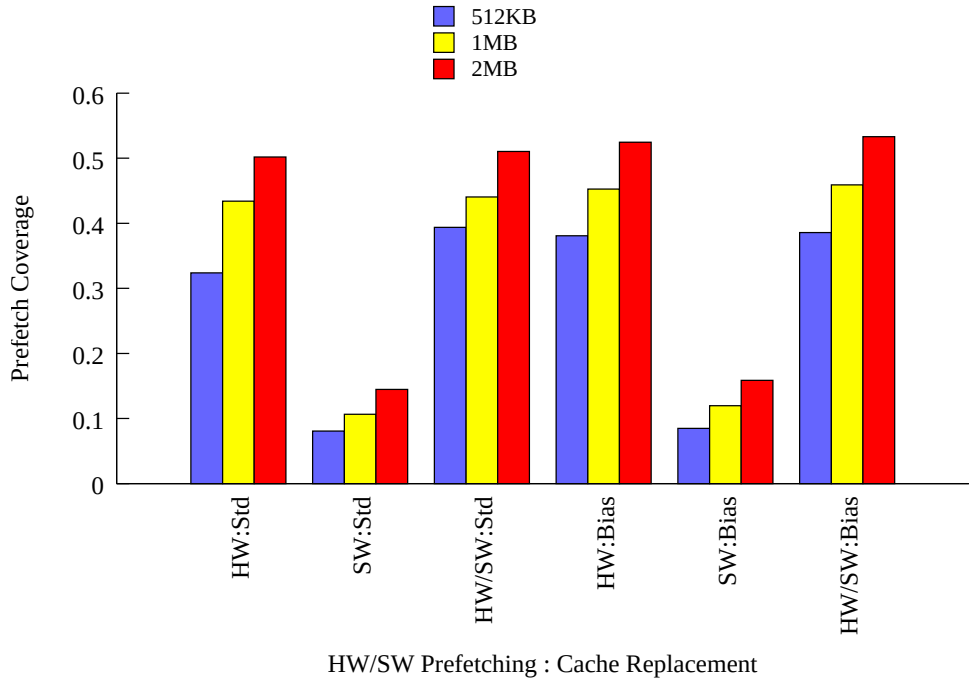

Figure 5.10 Prefetch coverage of various techniques for Java applications with 512KB, 1MB and 2MB L2 caches

overall with slight increases for hardware prefetching, slight decreases in software prefetching and both increases and decreases in combined hardware and software prefetching depending on cache size. Coverage overall increases with cache size also because of reduced conflict and capacity misses. A larger cache has fewer conflict and capacity misses, but will not drastically affect the misses identified by prefetching. Lines prefetched are most likely not already in the cache and, as other miss types decrease, those lines predicted through either software or hardware prefetching become a larger percentage of overall incurred misses.

5.5.7 Bandwidth Aware Performance Model

The read traffic, accuracy and coverage metrics help to visualize some of the costs and benefits of prefetching schemes, but ultimately the goal is increasing performance. To measure performance, we have implemented a performance model based on the Epoch model of Memory-Level-Parallelism (MLP) (17; 18), which takes bandwidth limitations into account. On a real computer system ,accuracy and coverage are not the only important aspects of prefetching. Another important aspect is timeliness. Timeliness refers to the availability of an issued prefetch relative to the time in which it is needed. A prefetch issued too early might be replaced before it is used, which should appear in the accuracy metric. However, a prefetch issued too late to hide all or part of the latency of a cache miss, while accurate, does not improve performance to the degree of a prefetch that has completed before it is needed. Computing timeliness requires a timing model and in the end is directly a component of runtime performance. Rather than report timeliness we resort to a runtime performance metric reported from our MLP timing model. **Figure 5.11** shows average performance impact of hardware prefetching, software prefetching and the combination in both standard and biased cache replacement schemes, for a range of available bandwidths, for three L2 cache sizes. The first observation to be made is that the employed techniques are not sensitive to cache size. While an increase in cache size will increase performance, the ratio of using a particular prefetching technique to not using the technique is fairly constant across the measured cache sizes. The second observation is that hardware prefetching shows increasing effectiveness as bandwidth

increases, while software prefetching is fairly constant across bandwidth values. The combination of hardware and software prefetching does offer a small improvement over hardware prefetching alone. The last observation to make is that biased replacement (even columns in each group) has almost no impact on final runtime performance. In fact, most biased schemes show slight performance degradation. Based on these results, replacement biasing does not offer any significant benefits in Java client applications.

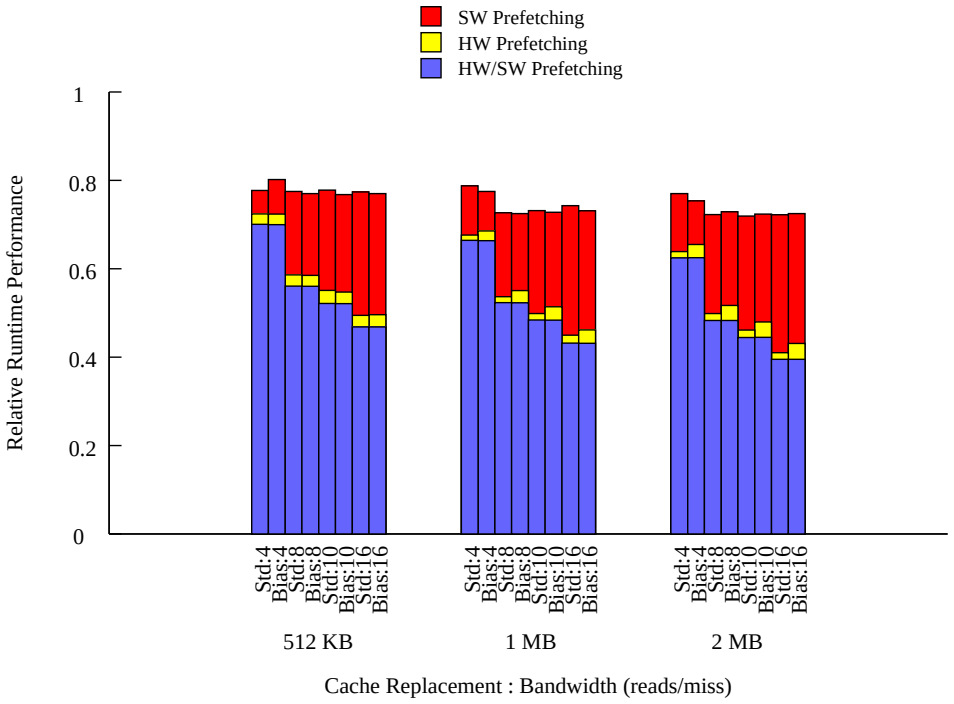

Figure 5.11 Average relative performance improvements for hardware and software prefetching with and without replacement biasing.

Although biased replacement has not shown its effectiveness in our Java benchmarks running on a single core architecture, we show one final example that illustrates the potential for its use in a multi-core shared cache environment. Recall that our contrived example from **Figure 5.5** showed promise in helping to segregate accesses between cores to limit the disruption among the interleaved accesses. SPECjbb2005 is a server application that has been designed to scale to multiple cores. We ran an additional configuration of SPECjbb2005 with four warehouses running on a four core shared cache environment over a wider range of L2 cache

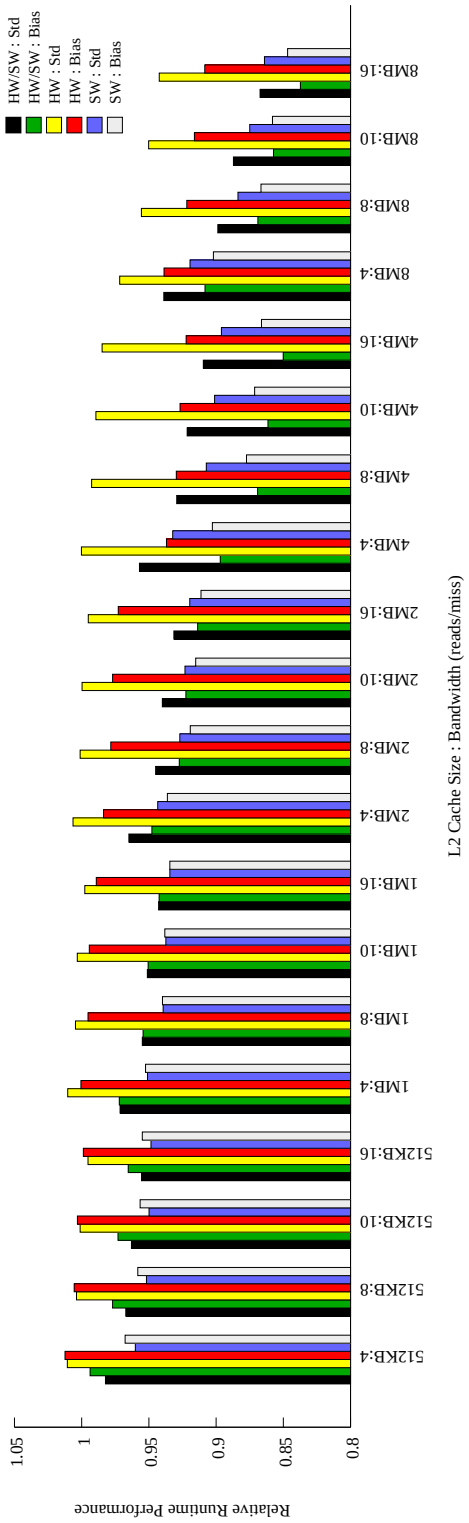

Figure 5.12 Relative performance for SPECjbb2005, 4 warehouses on 4 cores with a shared L2 cache.

sizes. This application was run through our MLP timing model with the same six prefetching configurations evaluated previously. **Figure 5.12** shows the results of that experiment. This figure shows the relative runtime performance for the prefetching and biasing configurations over L2 cache sizes ranging from 512KB to 8MB and bandwidth values of 4, 8 , 10 and 16 reads per miss. As bandwidth and cache size increase, all of the configurations see increased improvement. However, the degree of the improvement is somewhat unintuitive. Hardware prefetching, for example, shows the least improvement of the configurations in every comparison and in some cases is even worse than the baseline where no prefetching is employed. This confirms the practice of disabling hardware prefetchers in server environments. Published results on the SPEC website (69) for SPECjbb2005 often suggest disabling hardware prefetching. The explanation for this behavior is the limited available bandwidth to issue prefetches. While the accuracy of issuing prefetches for this application may be high, sufficient bandwidth is not available for those prefetches to be issued in a timely fashion.

One surprising fact is that biased replacement can offer more than 5% runtime performance improvement for 4MB and 8MB caches. In most arrangements, biased software prefetching offers the best overall performance improvement, as much as 15% for an 8MB cache with bandwidth of 16 reads per miss. For these larger caches, the combination of hardware and software prefetching and biased replacement provides the best performance, but only slightly better performance than biased software prefetching.

While our timing model is simplified to account only for memory operations, some of the trends can be verified against a real architecture. In measuring SPECjbb2005 performance on an Intel Xeon 5160 processor (two core shared cache), enabling hardware prefetching does in fact degrade performance by 2% to 3%. Adding software prefetching on allocation offers an improvement of 22%. Biased replacement is not currently implemented in real architectures, but since the technique alters only the victims chosen from the cache, the timing model we have chosen is not compromised when evaluating this technique.

5.6 Related Work

As this is a system level paper that incorporates many different aspects of both hardware and software, there are numerous related works. In this section, we discuss the work that is most closely related.

The first body of work we include is that of optimizing the memory management system of a Managed Runtime Environment (MRTE). As Java is the most prominent language today that employs an MRTE, much of the work has been conducted specifically for Java. Various techniques have been employed to leverage garbage collection, and or allocation, to achieve better locality within an application and yield better cache performance and ultimately improve runtime performance. Some of these techniques include runtime reordering of objects based on their inter-object locality (37; 84; 85). Others try to allocate objects together to achieve higher locality (16). These techniques are architecture independent and the indirect benefits are based solely on inter-object locality patterns.

An in depth evaluation by Blackburn, et al. found that the most important contributing factor, within garbage collection algorithm selection, to cache performance was the use of a nursery-style generational collector(11). This means that contiguous allocation of new objects yields the highest boost to cache performance due to the high temporal and spatial locality of newly allocated objects. Contiguous allocation is a feature prominent to most high performance JVMs. Our work leverages the contiguous allocation behavior and further increases performance by targeting allocation with prefetching and cache replacement biasing.

In our work, we propose software block prefetching as an addition to a hardware stream prefetcher. The body of work in the area of prefetching is very large. Software prefetching includes a host of work that largely revolves around automating the use of prefetches through compilation techniques. Many of these techniques are employed and evaluated in compilers that directly produce executable object code for a target platform. As our work specifically targets an MRTE, where applications are compiled into an intermediate bytecode form that is platform independent, we discuss only the works, of which we are aware, that target a similar environment. Although software prefetch techniques evaluated in an object-code-producing

compiler might be suitable for use in a Just-In-Time (JIT) compiler in an MRTE, we won't speculate on that migration.

Adl-Tabatabai, et al. investigated the use of prefetch injection within JIT compilation for an in-order architecture for linked data structures (1). Cahoon and McKinley also evaluated prefetching linked data structures (14) as well as arrays (15) in Java. Our work differs in that we are targeting prefetches to a behavior of the JVM itself, specifically the allocation algorithm, while running a Java application. Although not targeted to Java, Mowry (59) proposed a block prefetch but did not evaluate the technique as cache sizes were considerably smaller at the time and cache pollution was an important concern. Zhao et al. applied a block prefetch approach to specific applications in networking where packet sizes are larger than individual cache lines (87). Here we differ in that we leverage prefetching within the context of a virtual machine that can benefit a host of unmodified applications.

A multitude of hardware prefetching techniques exist and again we restrict our comparisons to those that are most similar to our proposed work. The closest hardware prefetching work that we are aware of comes from that based on the work of Lin, et al. where hardware aggressively prefetches whole pages on a demand miss to avoid future misses (55). Wang et al. extend the work by employing a compiler driven software hint approach to tune hardware prefetching (81). These works are similar to ours in that they target aggressive prefetching of a page-sized region. They differ in that both are more aggressive and significantly increase memory read traffic. The hint guided version also requires the compiler to encode the hints on all individual loads to ensure that any missing load carries with it the necessary information to guide prefetching. This mechanism was not evaluated in a runtime environment where the loads are initiated by a combination of virtual machine code and application code compiled under JIT. Our software prefetching is completely independent of loads and specifies an entire page region within a single hint, which has the potential to avoid all demand misses for allocation. Our hints are also tied to the address space of the data, and not the instruction pointers of load operations.

In the area of replacement policies, the work of which we are aware that is most closely

related is that of Wong and Baer (85). They employ a modified tree pLRU scheme to tag lines as either having temporal locality or not. We achieve a similar behavior without the need for an additional state bit per line and without modifying the victimization selection. By performing partial updates to the tree, we are able to bias updates to promote earlier eviction based on the original victimization policy. Lin and others have also proposed bringing prefetches into the LRU slot in a cache set to avoid pollution (55; 80), but our technique offers greater flexibility in set placement and can be used to differentiate prefetches. Wang et al. also proposed directing replacement on a per-load basis through compiler techniques (81).

5.7 Conclusions

Based on the results of this work, we offer the following suggestions in the design of both virtual machine software and architectures in support of Java. While allocation patterns are application dependent, bump-pointer allocation employed by many VM memory managers creates an obvious opportunity for software cache prefetching. We find that prefetching full pages, one page ahead of allocation, yields very high accuracy without disrupting other cache contents for typical cache sizes found today. In the absence of hardware prefetching, software prefetching for allocation can yield substantial performance improvements. In the presence of hardware prefetching, the addition of software prefetching for allocation provides a modest additional improvement. With no additional cost to employ software prefetching for allocation, we recommend it always be employed as the default configuration. Architecture could further support software prefetching techniques by offering a block-based prefetch which has been shown in this work to be feasible for block sizes up to 4KB (standard page size) and remain highly accurate without displacing other contents. We recommend this prefetch bring data only to the L2 level.

We confirm the common practice of disabling hardware prefetching in server environments where bandwidth becomes a limiting factor. Hardware prefetching can, in fact, degrade performance when demand misses saturate available bandwidth. In client computing environments, hardware prefetching can offer substantial benefits, from 25% to 55%, depending on cache

size and available bandwidth. In server environments however, with multiple cores sharing a cache, hardware prefetching can degrade performance by 1% to 2%. Substantially increasing the bandwidth, to double that of what is commonly available today, does not substantially alleviate the bottleneck.

The third technique we evaluate is cache replacement biasing for prefetches. For single-core client computing environments we find no compelling results to suggest such a technique be employed, although, a few select applications were benefitted for a 512KB cache. However, when looking at the results of SPECjbb2005 running in a four-core, shared-cache server environment, replacement biasing seems to offer a measurable improvement over prefetching alone, yielding an additional 5% improvement over the 10% offered by software prefetching. Applications which are sensitive to bandwidth limitations can benefit from replacement biasing, especially when multiple cores share a cache and compete for cache residency.

CHAPTER 6 Real Machine Performance Evaluation of Java and the Memory System

A paper submitted to
ACM Transactions on Architecture and Code Optimization[1]

Carl S. Lebsack and J. Morris Chang

6.1 Abstract

Micro-architecture studies of Java applications are extremely time consuming and complicated because of the sophisticated infrastructure needed to support the language. Simulation environments either do not have support for Java, or limit the evaluation to small segments of runtime and very few configurations. In this work we employ virtual memory page coloring within the operating system in conjunction with hardware performance counters on a real machine to expand performance evaluation of Java applications with respect to the memory system. We show previously unreported results that many Java applications are running close to the time predicted when no cache misses occur and that they do not fully utilize the cache of modern processors. Although we confirm that increasing the heap size improves performance, we also quantify the trade-off of consuming system resources. This trade-off motivates the use of smaller heaps for many applications. Additionally, we also show that some applications, specifically those in which performance appears to be overly sensitive to heap size, can substantially benefit by reducing the nursery size to improve mutator performance. The change in mutator performance is enough to outweigh the increase in garbage collection overhead for small heaps.

[1]Submitted for review, April 2008.

6.2 Introduction

The research community has been evaluating Java technologies because it is a popular language platform for development (75). It offers many beneficial features that are desirable to programmers, project managers, system developers and consumers alike. The code is readily portable between platforms and is robust and secure.

One of the biggest hindrances to the migration to Java from languages such as C is the performance overhead assumed when moving to a language running within a virtual machine. A significant amount of work has been poured into the Java platform to address these overheads. JIT is one particular advancement that substantially improves performance by dynamically converting Java bytecode to the machine code of the platform on which it is executing.

Another major runtime overhead is garbage collection, the process of dynamically reclaiming memory resources for reuse. Significant research has been employed into the development of algorithms for garbage collection, and the runtime overhead has been substantially reduced.

In light of all of the work in these fundamental technologies, we see a lack of research in the reporting of comprehensive performance analysis of the Java system, especially with regard to the memory system. The Java Virtual Machine environment is the culmination of a large number of components, and final runtime performance is determined by their interplay. One important set of results that we seek to provide is how a commercial JVM implementation (JDK 1.6) performs on standard benchmarks based on the conclusions drawn within the research community. We plan to study the detailed interactions of the full Java system on a real machine and report performance trade-offs with regard to the allocation of system resources.

Simulation infrastructures are extremely complicated and several orders of magnitude slower than a real machine executing a Java application. In our own prior work, we have leveraged simulators that require multiple days to execute a single run of a Java application that will run in a few seconds directly on the hardware of the same machine. Although simulation environments are highly flexible in terms of the ability to evaluate theoretical hardware or take measurements that would be infeasible on real hardware, they severely limit the scope of an evaluation in terms of breadth of configuration parameters simply be taking too long. It is

also very difficult to confirm the accuracy of the results from simulation environments because they often make simplifications in the model to make the simulation feasible (e.g. moving from cycle-accurate simulation to functional simulation in cache studies).

We avoid some of the pitfalls of a simulation environment by leveraging a novel application of page coloring within an operating system to vary the cache size available to Java applications. We combine this approach with the monitoring of hardware performance counters to analyze the performance of Java applications over a range of command line parameters that affect the memory layout for an application. To the best of our knowledge, this is the most comprehensive study of the performance of Java applications with respect to the computer memory system. We employ a real machine to gather data from a large number of configurations of heap size, nursery size and cache size.

Common practice, based on published research (35), is to use a fairly large heap to achieve good performance. We show that the runtime of Java applications is not necessarily highly sensitive to the size of the heap. Although increasing the heap size can improve performance, in many applications the performance improvement is highly disproportionate to the additional system resources consumed. This implies that, for many applications, it may be desirable to limit heap size to reserve memory and avoid burdening the virtual memory system in a multitasking environment.

It is also common practice to utilize half of the Java heap space for nursery allocation (8). We confirm that in most applications this practice is ideal. A larger nursery reduces the garbage collection overhead within the JVM. The nursery size does not require any additional resources from the system because it is merely a configuration parameter of the garbage collection algorithm and thus, for most applications, is best set at half the heap space (maximum for a copying collector). However, we find that some applications, whose runtime performance has a high sensitivity to heap size, benefit substantially by selecting a smaller nursery for small heaps. In these applications, the greater locality achieved by performing nursery collections more frequently yields benefits in mutator performance that outweigh the increase in garbage collection overhead. This trade-off, in the best case, can eliminate the heap sensitivity and

allow an application to achieve equivalent performance at a substantially smaller heap, saving system resources.

We also show that not all Java applications are sensitive to cache performance and predict that some applications are already running at near the performance predicted if no cache misses occur. This implies that although Java programs are memory intensive in that they allocate heavily, their performance is not limited by cache misses.

Additionally, we show that many Java applications do not fully utilize L2 caches of modern processors. Most of the applications we evaluate do not need more than 512KB of L2 cache. The profiling technique we propose could be used by application developers to identify how well they utilize cache. The information could also be used to manage cache sharing at the operating system level via page coloring. There is no need to grant Java applications with more cache than they need. This management could eliminate interference between running applications in shared caches of multi-core processors.

The rest of this work elaborates the methodology employed in the performance analysis of Java applications with respect to the memory system and details that lead to the above conclusions. This work is organized as follows. Section 6.3 describes some background on page coloring and how it is utilized to adjust the cache size on a real system. Section 6.4 describes the framework and methodology. Section 6.5 details the results of this work broken down into several different configuration parameters. Section 6.6 discusses related work and Section 6.7 concludes the discussion.

6.3 Background

Page coloring is a technique that was developed to ensure that specific pages did not compete for cache residency in low associativity caches. The technique is particularly useful for direct-mapped caches where poor cache mapping can severely degrade performance due to conflicts (13; 46).

Although caches manage data at the granularity of individual cache lines, a page is the smallest granularity over which an operating system has control. Each page maps to a region

in the cache. For a direct-mapped cache, the cache region will equal the page size. For larger associativities, the cache region will be equal to the associativity multiplied by the page size (e.g. a 4KB page on an 8-way associative cache will be mapped to 32KB region of the cache).

Figure 6.1 illustrates the meaning of coloring with respect to a computer memory system with physically addressed caches. The number of available page colors is the distinct number of pages that will never interfere with one another in the cache and is be equal to the cache size divided by the associativity, divided by the page size of the system. Pages of the same color will compete for residency in the cache, while pages of different colors will not. The diagram shows eight page lists organized by color.

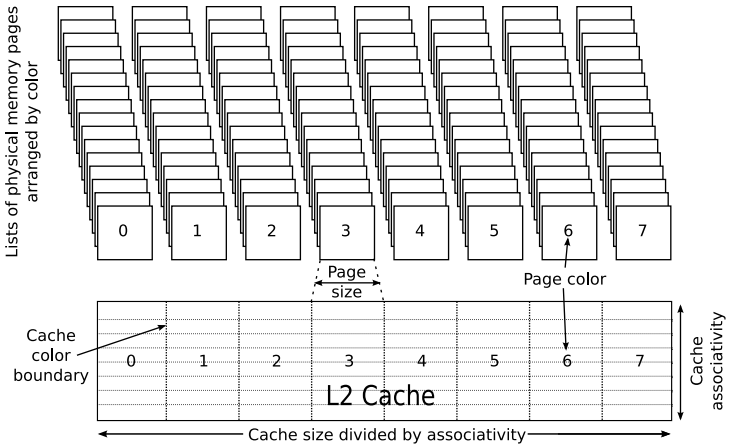

Figure 6.1 Cache partitioning using physical memory page coloring

Color refers to the subset of bits in the physical page identifier of a memory address that maps to distinct regions in the cache as shown in **Figure 6.2**. Coloring must occur at the lowest level of virtual memory control that has access to physical addresses as virtual addresses cannot guarantee proper physical mapping. Thus, if a virtual machine monitor is used, it must either provide the coloring, or provide a physical mapping to the operating system as not to obscure the possibility of page coloring. Page coloring at the application level is not possible unless the operating system provides coloring, or enforces a physical page mapping paradigm that does not preclude coloring at higher levels.

Typical deployments of page coloring are intended to mitigate cache contention on a running

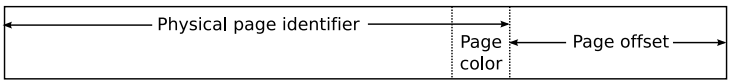

Figure 6.2 Page color bits in the physical address

system by attempting to load balance the cache through even distribution of the page colors. This balancing decision is often made by a compiler for patterned accesses in an application (2; 6; 45; 53; 60).

The implementation in FreeBSD 6.3 (28) (and prior versions) uses page coloring in a similar manner by keeping free pages in lists by color and allocating them in a round-robin order as pages are requested. This allocation policy guarantees that a single large allocation will be aligned in the cache with minimum internal conflict. Imagine the worst case scenario where no policy is in place to share the cache, and a large allocation includes pages of only one color. This scenario would restrict the large block of memory to only a small region of the cache.

Page coloring can be used to vary the size of the cache on a system by using only a subset of the available colors. Leveraging this technique we are able to investigate Java application sensitivity to cache sizes in a linear progression. We make use of a modified virtual memory manager in the operating system to prevent the use of pages of certain colors to reduce the cache available to a Java application. This technique allows us to evaluate cache performance on a real system for a range of cache sizes.

6.4 Experimental Framework and Methodology

In this section we describe the framework constructed to perform our performance evaluation. We include a description of the page coloring technique as applied to our selected platform. We also discuss the Java software stack including the JVM and benchmarks. This section also details the measurement methodology and the metrics gathered in this evaluation.

6.4.1 Page Coloring Details

Because the underlying virtual memory manager in FreeBSD uses a version of page coloring, we select this operating system as the basis for our work. We conduct all of the measurements on a real machine. The machine we selected is a Pentium IV with 1GB of main memory. The Pentium IV was selected specifically because of its hardware cache arrangement. It has a relatively large, 1MB 8-way associative, L2 cache. It also has a small, 8KB 2-way associative, L1 cache. This particular arrangement is ideal for our work because the L2 cache can be partitioned into thirty-two colors[2], without partitioning the L1 cache[3]. This is particularly important when we want to study the effects of L2 cache performance without introducing side effects associated with L1 cache behavior.

Recent processors have larger L2 caches, but also have larger associativity and larger L1 caches. The same experiments on an Intel Core 2 with 4MB, 16-way associative, L2 cache, for example, would require eight times as much main memory and would only be able to achieve 64KB page-color granularity instead of 32KB granularity. A recent AMD X64 processor with 1MB, 16-way associative, L2 cache and 64KB, 2-way associative, L1 cache can only be partitioned into two colors (512KB L2 cache each) without partitioning the L1 cache. Another page coloring study (73) described a problem in their attempt to partition the L2 cache of a PowerPC system with an L3 cache. They were unable to separate the effects of partitioning on each of the L2 and L3 caches. Nearly all of the applications studied show a substantial change in cache sensitivity within the 1MB L2 cache of the Pentium IV which helps to confirm our selection.

We have modified the page coloring implementation in FreeBSD to support a boot-time parameter to enable cache-sizing for the system. This parameter sizes the cache available to the system by removing whole lists of page structures from the available memory pool during the boot up sequence. Through this mechanism the 1MB cache can be reduced to a smaller cache (e.g. removing access to lists 3 through 7 in **Figure 6.1** effectively reduces the cache size by half). We can vary the cache size in 32KB increments, and study cache sizes ranging

[2] 1MB/(8-way * 4KB page) = 32 colors
[3] 8KB/(2-way * 4KB page) = 1 color

from 256KB to 1MB. This procedure allows us to study application sensitivity to cache size on a real system for a large number of configurations. To the best of our knowledge this study provides the most detailed cache analysis of Java applications in the research community.

6.4.2 Paging and Swap Space

Another important factor to consider when utilizing page coloring is the impact on the virtual memory system in terms of swap space. Each page color that corresponds to a portion of the cache also corresponds to a fraction of the physical memory available. In our system, each of the thirty-two colors maps to 32MB of main memory. If more memory is allocated than will fit in physical memory, the operating system will resort to paging, utilizing the hard drive as swap space. If this occurs when inadequate colors are utilized for a specific task, runtime will see a significant increase in overhead due to paging. To avoid allowing the adverse effect of paging to influence our performance measurements, we have disabled swap space on the system. This results in programs simply crashing when memory is exhausted, and thus, some programs are unable to run in certain configurations.

As a result of avoiding paging by forcibly disabling its usage, a hard lower bound is placed on the system in terms of sizing the cache. While the approach we employ could conceivably evaluate a system with 32KB of L2 cache, this would also restrict the entire system to 32MB of main memory. This memory would need to hold all application data and code pages as well as that of the operating system. In order to provide enough memory for our system to run without crashing, we do not test L2 cache sizes smaller than 256KB. Although this size is not the absolute minimum, it provides us with a wide range of cache sizes to evaluate and enough memory for most applications to run.

6.4.3 Java Environment

The focus of this research is on the memory system performance of a Java environment. We selected Sun's JDK 1.6 as the Java Virtual Machine as it is the only commercially available performance tuned JVM in which the source code is made available to the research commu-

nity. We have made modifications to the JDK under the Java Research License (58). The modification we made was to allow the JVM to make use of the user-level cache partitioning provided by our modified FreeBSD kernel. Specifically, we have made alterations which allow segregation of the nursery and the rest of the memory space of the JVM within the cache. This division is supported by our previous simulation works involved in isolating the nursery region in on-chip resources through physical separation (49) and logical separation through cache replacement biasing (50).

The applications we tested include the SPECjvm98 benchmark suite (69) and the DaCapo benchmarks (12). We have specifically excluded SPECjbb2005 (69) because of the substantially large memory requirement making it unsuitable for this study because of the physical memory limits of the system when sizing the cache. While SPECjbb2005 could be run, it could only be analyzed in a small number of configurations which would not produce enough data points to draw any conclusions. For the same reason, *hsqldb* from DaCapo, which requires approximately five times as much memory as the next demanding benchmark, was also excluded from our study.

6.4.4 Measurement Methodology

Performance metrics for Java applications can vary significantly depending on how the measurements are taken. In order to avoid erroneous results, we have taken measurements using the following approach. When a Java application is run for the first time, there are several overheads including class loading, JIT compilation and OS resource allocation. We ignore these startup costs and run the application several times within the same virtual machine instance. As the same application is rerun, the startup effects are eventually eliminated and results stabilize. However, there can be a significant difference in runtime between the early runs and later runs. The first few runs can take substantially more time than those that run once the system has stabilized. Many applications stabilize after only a few runs, but to ensure stability and maintain consistency in measurements, we run each application twenty times and include in our study only the measurements from the last ten runs. Each application is also run with

a range of heap sizes. During the measured runs, the system was configured to prevent other applications and daemons from executing and to avoid interference from network activity.

Georges et al. recently proposed a more rigorous measurement methodology requiring substantially more runs than we measure (30). The breadth of this work makes the higher number of runs less practical. Also, the conclusions drawn in this study are on trends that are quite apparent and not sensitive to minor changes in absolute measurement values. However, to provide a modest comparison between our measurement methodology and that of Georges et. al, we utilize their suggestion on three data points randomly chosen from the thousands included in this study and compare to the methodology we use. For the points selected we compute the confidence intervals and find that the runtime variation for all three points is less than 2%. While not an exhaustive comparison, we contend that the methodology we employ is sufficient for the conclusions drawn.

6.4.5 Metrics

During the course of this work, we focus on performance and use runtime to determine the ultimate impact of any of the modified parameters. Simulation environments often derive metrics to estimate runtime. Since all of these experiments are run directly on a real system, we directly measure the primary metric of interest. The test harnesses for both the DaCapo and SPECjvm98 suites were modified to take measurements for each individual application run within a single JVM instance. Runtime was measured by a microsecond timer in the operating system. As L2 cache performance is suspected to be a key indicator of runtime performance, the test harness also recorded hardware performance counters of the Pentium IV processor for L2 read and write misses. A set of profiling data is also monitored based on the *getrusage* function to identify other potential contributions to runtime overhead, such as excessive reliance on the operating system. All of these measurements are taken only at the start and end of each run to avoid introducing any overhead into the system from the measurements themselves.

Because measurement on a real system is several orders of magnitude faster than measure-

ment on a simulator, we are able to gather a large amount of data and evaluate many different configurations. The two configuration parameters we are most interested in are cache size and heap size because these are both finite resources available on the system. These parameters can be varied independently and we attempt to estimate their individual impacts on runtime. While their contributions to runtime are largely orthogonal, we will also evaluate their impact when varied simultaneously.

Additionally, we also evaluate nursery size, a configuration parameter that affects the behavior of the garbage collection algorithm. The nursery size is not a resource as are heap memory and cache size. The nursery is simply an algorithmic parameter which regulates the frequency of garbage collection within a sub-region of the heap. For the JDK, the minimum heap size allowable is 192KB, and the maximum is half of the heap space.

6.5 Experimental Results

6.5.1 Java minimum heap values

The first step in performing the measurements in this section is to determine the minimum heap requirements for each application in both the DaCapo and SPECjvm98 benchmark suites. There are numbers reported in the literature for both suites (12); however, these values are specific to the Jikes RVM and associated class library used when taking the measurement. The previously reported numbers are used as a guideline, but we report the specific numbers we use for JDK 1.6. The value used for the minimum heap for each application is shown in **Table 6.1**.

The values in **Table 6.1** were determined empirically by running the application with the minimum nursery (192KB for JDK) and narrowing down the heap value to the point where the application just runs. All presented results are in terms of multiples of the minimum heap values reported in this table.

The minimum heap values are subject to some caveats. The determination method is restricted by the lower limit of 1MB imposed by the JDK, which may not be the actual lower limit for an application. The method may also produce some unpredictable results as

Table 6.1 Minimum heap values on JDK 1.6 for DaCapo and SPECjvm98 benchmarks

DaCapo		**SPECjvm98**	
Benchmark	Min. Heap (KB)	Benchmark	Min. Heap (KB)
antlr	1024[1]	_201_compress	6272
bloat	2176	_202_jess	1024[1]
chart	10368	_209_db	8320
eclipse	16512	_213_javac	6272
fop	6272	_222_mpegaudio	1024[1]
hsqldb[2]	75904	_227_mtrt	6272
jython	2176	_228_jack	1024[1]
luindex	1024[1]		
lusearch	2176		
pmd	14464		
xalan	7296		

[1]Minimum heap allowed by JDK.
[2]Excluded from the rest of the study because of the large heap.

_227_mtrt occasionally will run in a very small heap (1MB) and other times will crash with a larger heap (greater than 4MB). Previously reported heap values for *_227_mtrt* are the same as for *_213_javac* and we confirm that *_227_mtrt* runs regularly at this heap size.

6.5.2 Java user and system time

Before evaluating the Java applications and their sensitivity to cache and memory resources, we also provide a profile of the applications with respect to their behavior within the system. The runtime of a Java application will be dependent on code executed on behalf of the application itself, the mutator and overhead from the JVM, including garbage collection and JIT compilation. However, there is also the potential that overhead will come from reliance on the Operating System which can be incurred from substantial file operations, locking mechanisms or from a host of other system calls.

Table 6.2 shows the breakdown of user time and system time for the Java applications from DaCapo and SPECjvm98. Most applications show very little overhead from the system. There are three applications with some moderate system overhead, *antlr*, *eclipse* and *luindex*. Two applications, *fop* and *_228_jack*, incur greater than 40% of their runtime as overhead from system calls. This means that these two applications should show lower sensitivity to configuration parameters such as heap size when compared to other applications. As seen in

Table 6.2 Breakdown of user and system time for DaCapo and SPECjvm98 benchmarks

DaCapo			**SPECjvm98**		
Benchmark	User	System	Benchmark	User	System
antlr	0.829	0.171	_201_compress	0.973	0.027
bloat	0.995	0.005	_202_jess	0.958	0.042
chart	0.986	0.014	_209_db	0.991	0.009
eclipse	0.895	0.105	_213_javac	0.986	0.014
fop	0.330	0.670	_222_mpegaudio	0.994	0.006
hsqldb	0.998	0.002	_227_mtrt	0.997	0.003
jython	0.996	0.004	_228_jack	0.552	0.448
luindex	0.863	0.137			
lusearch	0.916	0.084			
pmd	0.969	0.031			
xalan	0.973	0.027			

Table 6.3 in the next section, both applications show very low sensitivity to heap size.

6.5.3 Java runtime sensitivity heap size

Next, each application is run over a range of heap sizes from twice its minimum-heap requirement to five and a half times the minimum in half minimum-heap increments. Reports in the literature (35) suggest that five and a half times the minimum heap offers performance equivalent to explicit memory management (no garbage collection). Therefore, we expect that this heap value will offer us the best performance in our sweep and we use it as a baseline for comparison.

Aside from the heap size, the nursery size is also an important parameter when running a Java application. If not specified, a JVM will default to a value, but we wish to know its contribution to the performance. One recommendation is to use half of the heap for the nursery (8), and this technique is often the default reported in the literature when using Jikes RVM (39). Sun's JDK also has optimizations for using half of the heap for the nursery (71). Our initial sweep over heap size fixes the nursery to one half the heap. This setting is also the reason that the smallest heap we evaluate is twice the minimum. To guarantee that a program will run, its residency, all of the live objects, must fit within the mature space, the heap not including the nursery.

The results in **Table 6.3** show the runtime sensitivity to heap size of the applications in DaCapo and SPECjvm98. The reported values are the percentage runtime increase at

twice the minimum heap versus the runtime at five and a half times the minimum heap. These results generally confirm that the larger heap has the best runtime. The application, *_209_db*, is the notable exception and it has better performance at the smaller heap. The application, *_222_mpegaudio*, also shows better performance at the smaller heap, but by such a small margin that the conclusion is that the application is not sensitive to the heap size. As *_222_mpegaudio* allocates only about twice its minimum heap total, garbage collection is a negligible contribution to runtime.

Several of the applications are not very sensitive to heap size and their runtimes are affected by less than 5%. While it is true that these applications have better performance at five and a half times the minimum heap, the improvement is very limited. In an environment where resource allocation is important, such as the average multitasking system, it would be beneficial to limit the heap size and pay the small performance penalty rather than rapidly exhaust the physical memory and cause the system to resort to paging, which can severely impact the performance of all running applications.

Table 6.3 Runtime increase at 2X heap over 5.5X heap with a half-heap nursery for DaCapo and SPECjvm98 benchmarks

DaCapo		**SPECjvm98**	
Benchmark	Runtime %	Benchmark	Runtime %
antlr	16.1	_201_compress	2.6
bloat	1.9	_202_jess	8.2
chart	7.6	_209_db	-15.1
eclipse	1.9	_213_javac	86.7
fop	1.4	_222_mpegaudio	-0.1
jython	5.0	_227_mtrt	122.4
luindex	7.0	_228_jack	3.7
lusearch	4.3		
pmd	33.2		
xalan	207.5		

6.5.4 Java runtime sensitivity to nursery size

Some of the applications show an extremely high sensitivity to the heap size. These applications include *xalan*, *_213_javac* and *_227_mtrt*. Before drawing any conclusions, it is important to point out that the results in **Table 6.3** are incomplete. As specified earlier, the nursery is fixed at half of the heap. The size of the nursery is another option in addition to the heap size

that is adjustable at the command line for the virtual machine. The major difference is that setting the nursery size does not require any resource allocation but is simply a configuration parameter to the garbage collector that determines how the heap space is used. **Figures 6.3** and **6.4** show the runtime versus nursery size broken down by heap size for the DaCapo and SPECjvm98 benchmarks respectively.

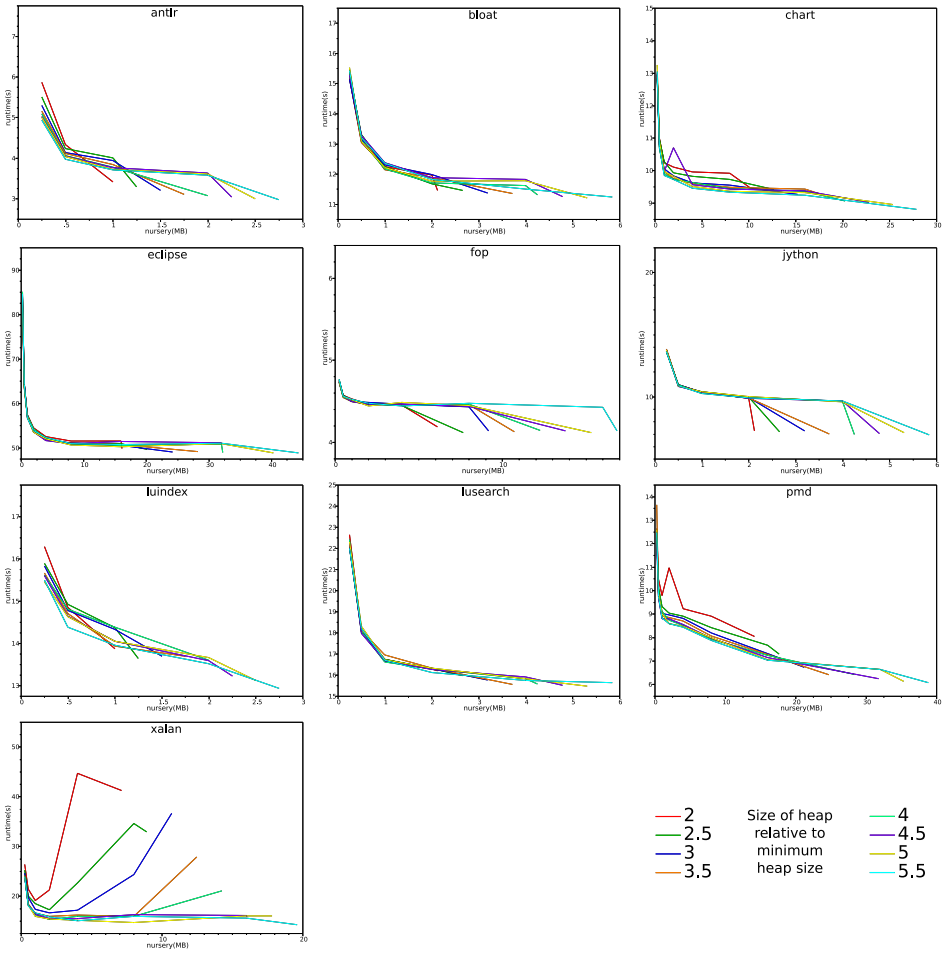

Figure 6.3 Runtime vs. nursery size by heap size – DaCapo

For most of the applications in both suites, the runtime trend follows a reciprocal relationship with the size of the nursery. This confirms the general practice of using an Appel style collector where the nursery is allowed to utilize half of the heap space. The sharp drop for the last point, the value of exactly one half of the heap, indicates specific optimization performed by the JDK at this value. However, the trend is not present for all applications.

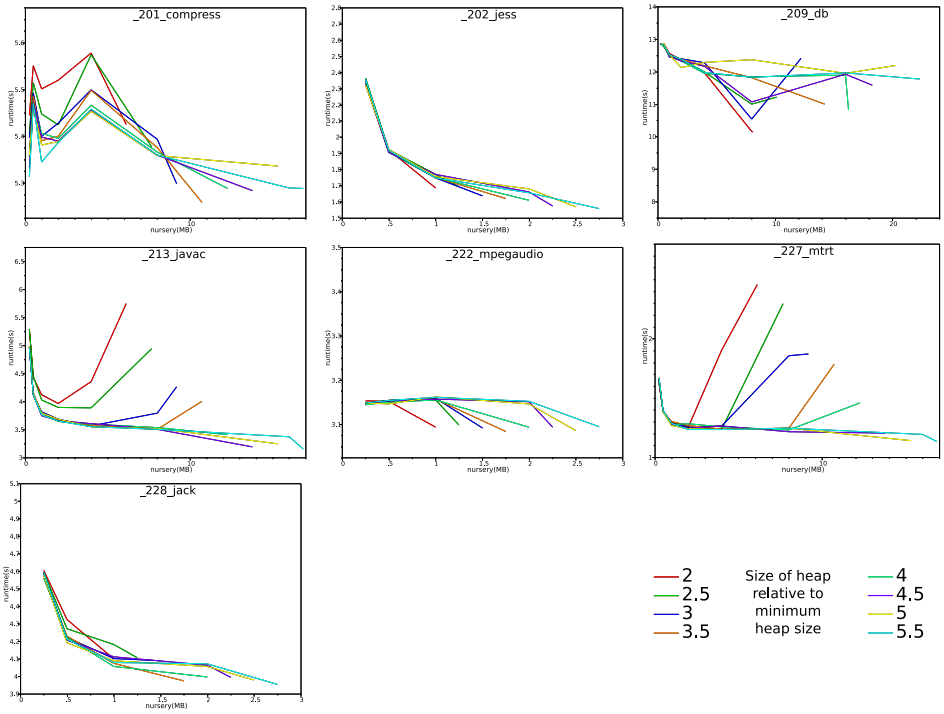

Figure 6.4 Runtime vs. nursery size by heap size – SPECjvm98

First, notice that *_201_compress* (**Figure 6.4**) shows some erratic behavior for small nurseries. This can be explained by the behavior of the program which allocates mostly large buffers. When the nursery size is too small, the buffers are allocated directly to the mature space. Once the nursery size is large enough to accommodate the buffers, extra expense is incurred when they are copied to the mature space. As the nursery size is further increased, the buffers die before they need to be copied. This application still benefits from the half-heap nursery philosophy.

Another program that shows erratic behavior is *_209_db* (**Figure 6.4**). This application performs best with a smaller heap. Also, the nursery size is ideal when it matches the size of the database that is allocated at the start of the application. The whole database is copied, in order, to the the mature space.

One pattern that is important to notice appears in *xalan* (**Figure 6.3**), *_213_javac* and *_227_mtrt* (**Figure 6.4**). These are the same three applications that showed the greatest sensitivity to heap size. As can be seen in the graphs, smaller heap sizes have the best performance

for a nursery that is smaller than half of the heap. If the ideal nursery is taken into consideration when computing an application's sensitivity to the heap size, several applications become less sensitive. Those applications which improve with twice the minimum heap by selecting a smaller nursery are included in **Table 6.4**. This table includes the new measure for heap sensitivity as well as the ideal nursery size when the heap is twice the minimum. For comparison, at twice the minimum heap, the nursery would be the same size as the minimum heap value presented in **Table 6.1**.

Table 6.4 Runtime increase at 2X heap with optimal nursery over 5.5X heap with a half-heap nursery for DaCapo and SPECjvm98 benchmarks

DaCapo			**SPECjvm98**		
Benchmark	Runtime %	Nursery (KB)	Benchmark	Runtime %	Nursery (KB)
fop	1.2	4096	_201_compress	2.5	256
lusearch	3.7	2048	_209_db	-20.8	8192
xalan	32.6	1024	_213_javac	19.0	2048
			_227_mtrt	5.4	2048

The runtime percentage values for some applications change only slightly, however, for the values for applications that showed high sensitivity to the heap size are all reduced significantly. In *_227_mtrt* especially, the change is substantial and the application no longer appears to be very sensitive to the heap size. While the practice of setting the nursery size to one half the heap is largely confirmed, these results show that some applications can benefit substantially by reducing the size of the nursery. The value of the nursery is application dependent and should be determined by profiling. Even for applications with a higher sensitivity to the heap size, it will still be important to consider the level of sensitivity with respect to the value of system memory. A significant amount of memory can be saved by reducing the heap size to twice the minimum with a maximum of 33.2% increase in runtime (*pmd*). Substantially reducing the heap memory has a minimal impact on the runtimes of many applications.

As the heap size is increased, all applications show a reduced sensitivity to nursery size and using half of the heap is sufficient for all applications at five and a half times the minimum heap. Some applications do have a lower runtime with a smaller nursery size, but the difference in runtime is substantially smaller than for smaller heap sizes.

Showing that the runtime of an application can be reduced by shrinking the nursery does not provide an explanation for what is occurring. The reason that a large nursery is generally beneficial is that as the nursery size is increased, garbage collection is invoked less frequently, objects are given more time to die and the overhead of garbage collection is reduced. It is this behavior that causes one to expect that a large nursery is ideal.

However, for the applications *xalan*, *_213_javac* and *_227_mtrt*, the large nursery pattern does not seem to hold for small heaps. The explanation for the reduced runtime for a small nursery in these applications is not obvious. To illustrate what is happening, **Figure 6.5**[4] shows the runtime of *xalan* broken down into mutator time (time spent running the application) and garbage collection time. Garbage collection time does indeed follow the expected pattern where, as nursery size increases, runtime decreases. However, the mutator time increases sharply before stabilizing as nursery size increases. The change in mutator performance can be explained by the fact that when garbage collection occurs, the objects that are live are consolidated into a smaller heap region, increasing their spatial locality. This compaction leads not only to fewer cache misses for the mutator, but also fewer TLB (translation-lookaside-buffer) misses. We recorded TLB misses with respect to nursery size, but to save space we have left out the plot, as it is almost identical to that of runtime because of a very high linear correlation. The total runtime is a combination of garbage collection time and mutator time (and JIT, not shown). Thus, for certain applications, it is more important to allow additional garbage collection overhead because of the locality benefit to the mutator. Even applications that are highly sensitive to the heap size can be run in smaller heaps with a reduced performance impact by selecting the proper nursery size.

6.5.5 Java runtime sensitivity to cache performance

Before looking specifically at the runtime sensitivity of Java applications to cache size, we want to illustrate why we expect cache to be an important resource. In today's research

[4]These graphs are produced from data gathered on a separate machine where additional instrumentation was added to breakdown runtime. The y-axis was removed to avoid confusion with absolute results from the other experiments. These graphs are used for illustration purposes only.

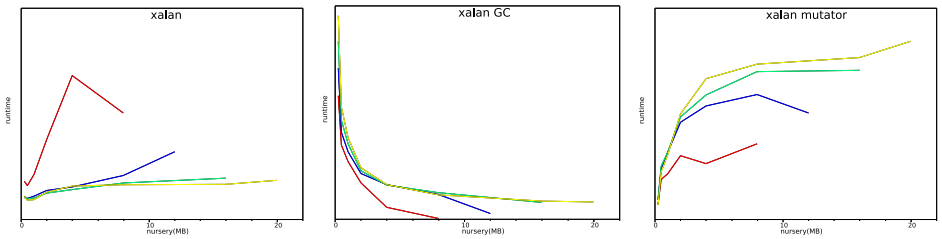

Figure 6.5 Breakdown of runtime into GC time and mutator time vs nursery size by heap size for xalan

environment, it is often taken for granted that Java applications are memory intensive and that they should be sensitive to cache performance. Often, studies in simulation environments focus on cache miss reduction and expect that to equate to some level of runtime reduction. Our own prior work (50) was based on that assumption. Now that we have a real system to evaluate the runtime of a large number of configurations, as well as their cache misses, we can determine the degree to which cache misses and runtime correlate.

We would expect the correlation between runtime and cache misses to be mostly linear, because the time required to service a cache miss is dependent on the latency of memory. Prefetching and out-of-order execution can mask some of the effect of memory latency, but most cache misses will have a fixed associated delay. **Figure 6.6** shows the runtime plotted against cache misses for a large number of configurations where the heap size and cache size were varied for *eclipse*. The fitted line is the linear approximation of the correlation between cache misses and runtime. Visually, the line looks to fit the data well. In order to quantify the fit of the linear approximation, we use residual analysis and report the R^2 value. For *eclipse*, the value is 0.986, which means 98.6% of the variation in runtime is linearly correlated to a variation in cache misses.

Table 6.5 shows the residual analysis for a linear fit of runtime to cache misses for the applications of DaCapo and SPECjvm98. The R^2 value is reported along with the slope of the line in seconds per million cache misses. The slope does vary between applications but should largely be reflective of the memory latency of the system. Some variance can be caused by whether some cache misses are overlapped within the application.

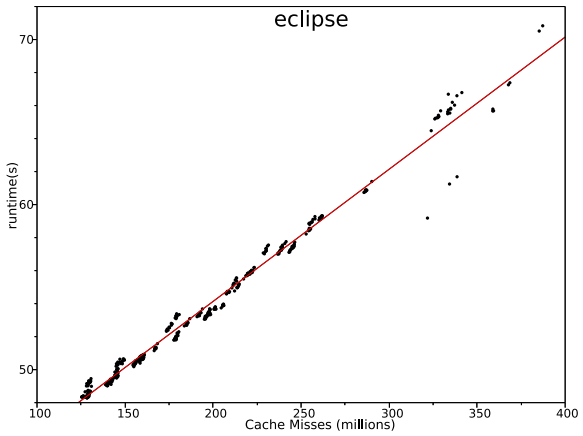

Figure 6.6 Linear correlation between cache misses and runtime for eclipse

Table 6.5 Linear correlation between cache misses and runtime for DaCapo and SPECjvm98

DaCapo			**SPECjvm98**		
Benchmark	R^2	slope	Benchmark	R^2	slope
antlr	0.879	0.0929	_201_compress	0.999	0.092
bloat	0.973	0.0838	_202_jess	0.919	0.1195
chart	0.980	0.0969	_209_db	0.901	0.1036
eclipse	0.986	0.0801	_213_javac	0.946	0.404
fop	0.982	0.0846	_222_mpegaudio	0.973	0.077
jython	0.766	0.0841	_227_mtrt	0.982	0.4389
luindex	0.797	0.1042	_228_jack	0.940	0.0896
lusearch	0.993	0.0739			
pmd	0.813	0.1314			
xalan	0.839	0.1748			

With the exception of *luindex* and *jython*, the applications which have the poorest linear correlation between runtime and cache misses are the applications which have the greatest sensitivity to heap size. This means that the garbage collector causes an increase in runtime but not necessarily an equivalent increase in cache misses. A larger slope indicates that the particular application has a greater sensitivity to cache misses. From this table, one would expect that *_213_javac* and *_227_mtrt* would be the most susceptible to runtime improvement by reducing cache misses.

The linear fit has another interesting application. The fit can be used to predict the runtime of an application with no cache misses. **Table 6.6** shows the predicted runtime for the applications with no cache misses. The accuracy of this prediction is dependent on reliability of the model expressed in the previous table as R^2, as well as the rest of the application remaining unmodified (e.g. no change in GC policy). While there may not actually be a way to get rid of all of the cache misses, it does provide an interesting bound on performance and allows an Amdahl's law evaluation of techniques that reduce cache misses. The table also includes the best runtime achieved in this investigation along with the ratio of the best runtime compared to the predicted runtime with no cache misses. These best runs include the large heap selection as well as full use of the 1MB L2 cache.

Table 6.6 Predicted runtime with no cache misses, best achievable runtime and ratio for DaCapo and SPECjvm98

DaCapo				**SPECjvm98**			
Benchmark	Pred(s)	Best(s)	Ratio	Benchmark	Pred(s)	Best(s)	Ratio
antlr	2.29	2.74	1.20	_201_compress	4.84	5.25	1.08
bloat	10.78	10.71	0.99	_202_jess	1.45	1.50	1.03
chart	8.26	8.69	1.05	_209_db	3.90	9.84	2.52
eclipse	38.10	48.35	1.27	_213_javac	1.37	3.07	2.24
fop	3.98	4.01	1.01	_222_mpegaudio	3.06	3.06	1.00
jython	5.43	5.97	1.10	_227_mtrt	0.58	1.04	1.80
luindex	12.35	12.83	1.04	_228_jack	3.83	3.90	1.02
lusearch	13.63	15.42	1.13				
pmd	4.93	5.92	1.20				
xalan	9.11	13.79	1.51				

Based on the results in this table, the conclusion is that many of the Java benchmark applications are not terribly sensitive to cache misses and offer limited room for improvement. Most applications are less than 10% slower than the predicted values with no cache misses.

Both *antlr* and *pmd* are 20% slower. The applications *xalan*, *_209_db*, *_213_javac* and *_227_mtrt* are all 50% or more slower. Therefore, one would expect that these applications would be the most suited to cache miss improvements.

6.5.6 Java runtime sensitivity to L2 cache size

Now that cache misses have been correlated to runtime, we evaluate cache as a resource that a system could allocate to a program in an attempt to improve runtime performance. Traditionally, cache is not explicitly allocated to a process. Running processes all compete for residency within the cache. For single processor machines, the operating system can tailor the context switch time to help timeshare the cache among processes. However, multi-core processors have become the norm and shared L2 caches are common.

To evaluate how a single Java application uses cache, we have configured our system to vary the cache size through page coloring within the operating system. The cache size of our machine is varied from 256KB to the full 1MB. Initially, we evaluated the change in 128KB increments. We further evaluated areas where there appeared to be an interesting transition in 32KB increments.

Figures 6.7 and **6.8** show the plots of runtime versus L2 cache size for applications from DaCapo and SPECjvm98, respectively. The graphs include plots for heap sizes ranging from twice the minimum heap to five and a half times the minimum. For these tests, all applications are run with a half-heap nursery.

The 256KB lower cache bound is set because of the limitations of the system. Memory is also limited when cache is limited because of page coloring. We avoid paging by disabling swap space, which causes a program to simply crash if not enough memory is available. Evidence of overly constrained memory resources can be seen in the graph for *eclipse* for smaller cache sizes, where data points are missing. The plot for *pmd*, which has a slightly smaller heap than *eclipse*, shows no missing points. Continuing to test smaller caches would gradually cause more programs to fail. We show over the range tested a good picture of cache impact on performance. The absolute lower bound based on page coloring is 32KB, one color. For our

system, this size would also restrict the entire process to 32MB of memory for all data space, heap and stack, as well as all executable code pages for the VM, Java code converted by JIT and any libraries loaded by the system.

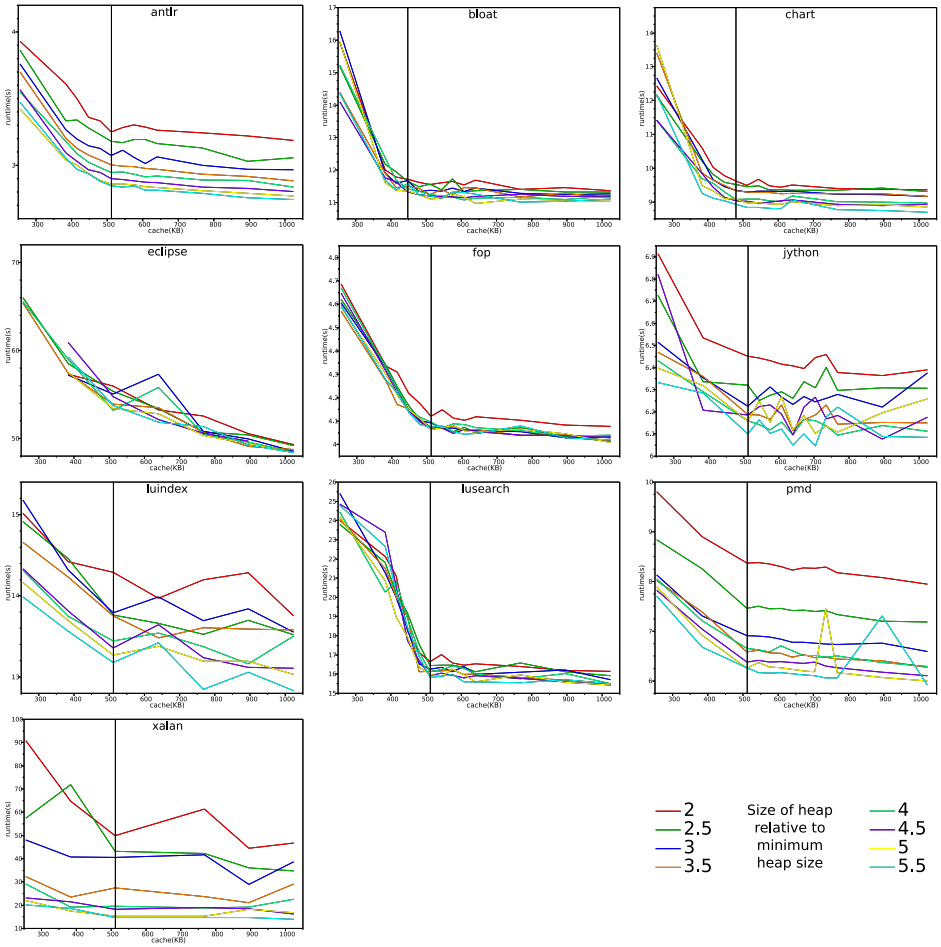

Figure 6.7 Runtime vs. cache size by heap size – DaCapo

The vertical lines in the graphs show the cache size where runtime improvement changes drastically. To the left of the line, the runtime is considerably more sensitive to cache size variation, while to the right, the sensitivity is significantly lower or non-existent. The lines are shown relative to the visual bend shown in the plots. The only two applications that do not exhibit a clear change are *eclipse* (**Figure 6.7**) and *_209_db* (**6.8**). These two applications exhibit a reciprocal relationship between runtime and cache size. This might be what one would expect but as can be seen from the figures, it is not the common case.

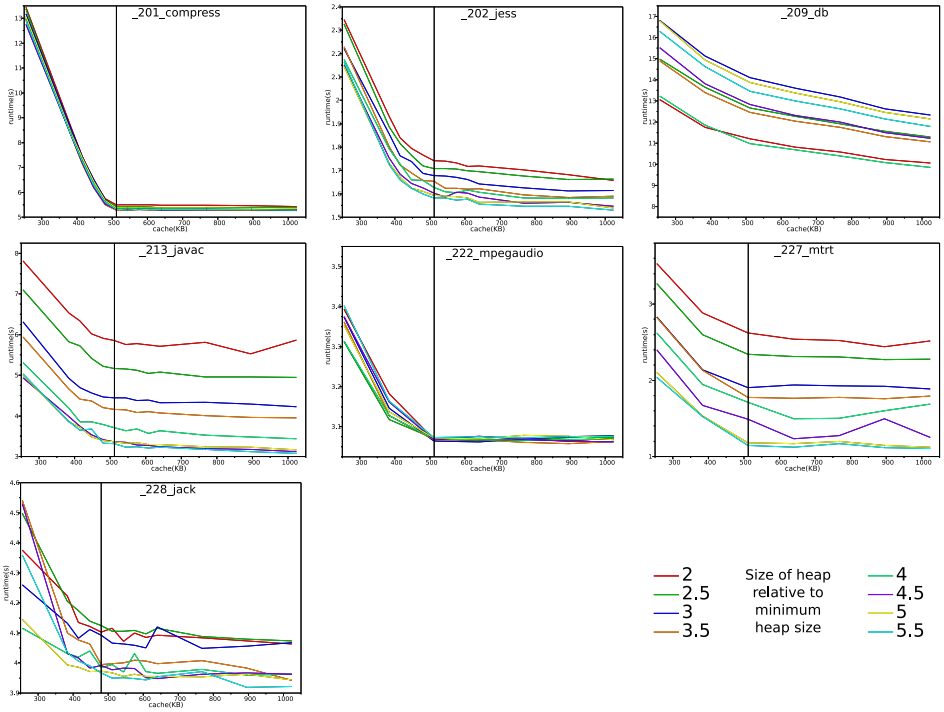

Figure 6.8 Runtime vs. cache size by heap size – SPECjvm98

Another observation from **Figures 6.7** and **6.8** is that heap size and cache size have largely orthogonal effects on runtime. These effects can be seen the most clearly in parallel lines for the applications *antlr*, *_202_jess* and *_213_javac*. The orthogonality means that the effects are largely additive and cache size and heap size can be determined independently.

However, there is also the possibility of a secondary effect where cache size and heap size are not independent. This effect does appear in some configurations, such as *eclipse*, when run with 512KB and 640KB cache. At heap multiples of three and four, the two peaks in the middle of the graph, the smaller cache outperforms the larger cache. In all other heap multiples, as cache size increases, runtime decreases. The graph shows that the larger cache actually shows an abnormally high average runtime for those two heap multiples when compared to the rest. A similar case appears in *pmd* for large heaps (right side of the graph). We suggest that these cases are caused by cache alignment. Changing either the size of the heap or the cache in a linear fashion will alter the alignment of objects within the cache. Because the caches are not fully associative, the limited associativity creates boundaries of contention within the cache.

By slightly changing the alignment, different memory blocks will compete for residency within those boundaries in the cache. Complete orthogonality between cache size and heap size are not guaranteed, and a detailed profile may be required for a specific application to determine if secondary effects occur.

Most applications do not effectively use more than 512KB of cache. **Table 6.7** helps to quantify the cache sensitivity of the applications shown in the above graphs. Each benchmark is reported with the ideal cache size, the size at which no substantial improvement occurs for larger caches. These values are the same for the lines drawn in **Figures 6.7** and **6.8**. The other three columns report the runtime ratios for 256KB cache to 1024KB cache, 256KB cache to the ideal cache size (Best Cache) and the ideal cache size to 1024KB cache, respectively. These values are calculated as the geometric mean of the ratios for all heap sizes for each application.

Table 6.7 Cache size sensitivity for DaCapo and SPECjvm98

DaCapo				
Benchmark	Best Cache	R1[1]	R2[2]	R3[3]
antlr	512	1.259	1.216	1.035
bloat	448	1.353	1.296	1.044
chart	480	1.369	1.324	1.033
fop	512	1.147	1.132	1.013
jython	512	1.055	1.056	0.999
luindex	512	1.090	1.071	1.018
lusearch	512	1.563	1.512	1.034
pmd	512	1.269	1.210	1.049
xalan	512	1.409	1.373	1.027
SPECjvm98				
Benchmark	Best Cache	R1[1]	R2[2]	R3[3]
_201_compress	512	2.462	2.443	1.008
_202_jess	512	1.396	1.346	1.036
_209_db	512	1.353	1.197	1.131
_213_javac	512	1.513	1.433	1.056
_222_mpegaudio	512	1.095	1.095	1.000
_227_mtrt	512	1.669	1.588	1.051
_228_jack	480	1.090	1.070	1.018

[1]Ratio of runtime with 256KB cache to 1024KB cache.
[2]Ratio of runtime at 256KB cache to Best Cache.
[3]Ratio of runtime at Best Cache to 1024KB cache.

Table 6.7 shows that some applications have low sensitivity to cache size for the range tested. The applications *jython*, *luindex*, *_222_mpegaudio* and *_228_jack* all have less than a 10% runtime impact by cutting cache size to 256KB over allowing the programs to use the

entire 1MB. The second column, R2, shows the ratio of the runtime at 256KB to the ideal cache size. Nearly all applications show that most of their cache sensitivity falls within the region between 256KB and their ideal cache size. The one exception is *_209_db*, which is included only for comparison purposes and does not actually exhibit an ideal cache size. This is reinforced by the values in the third column, R3, which show the runtime ratio between the best cache and 1024KB. This value can be interpreted as the runtime increase for the application when the cache is reduced from 1MB to the ideal cache size.

6.6 Related Work

Java performance has been evaluated by a number of researchers. There are a large number of works that investigate contributions to Java performance, and these contributing factors are many. In our investigation, we focus specifically on memory aspects, namely the size of the heap and nursery for a generational collector, and the size of the cache.

First, Hertz et. al studied the relative performance of garbage collection to explicit memory management (35). Their objective was to quantify garbage collection overhead and they found that when using a large enough heap, five and a half times the minimum, an application utilizing garbage collection can run at an equivalent performance as if it had used explicit memory management. This study is one that often prompts people to utilize large heaps when conducting their analysis, although many studies do include varying heap size. Our objective is quite different in that we aim to quantify the trade-off of the greater number of resources used when selecting a larger heap.

It is commonly known that a larger heap often improves performance by reducing garbage collection overhead. Blackburn et. al (11) further report that selecting a heap so large as to avoid garbage collection is not necessarily best for performance, as overall locality will be better if copying garbage collection periodically condenses live objects. They state that the most important factor to runtime is selecting a nursery-style generational collector because of the improved cache behavior. We extend the analysis to quantify not only the trade-off of heap size to runtime but also the trade-off of cache size to runtime.

Several studies include performance counter monitoring to sample cache miss rates as part of their performance evaluations of Java applications (1; 27; 37; 72; 33). However, none have performed a detailed study of the contribution of cache performance to runtime including a sensitivity to cache size. In this regard, our work provides significant insight into the performance of Java applications with respect to cache performance.

Outside of the Java research community, researchers have used page coloring to partition caches of processors between applications. Lin et al. studied page coloring as a method of evaluating proposed dynamic cache partitioning strategies on a real system (54) using SPEC (69)applications. Tam et al. studied static partitioning of the cache on multiprocessors using page coloring for SPEC applications (73). They did include SPECjbb2000, a Java application, but did not report other configuration parameters of interest to the Java community. Our work is similar in that we use page coloring as the mechanism to enable cache partitioning, but our objective is to determine the cache sensitivity of Java applications which is quite different. However, our results could be useful to the same community who may be interested in cache partitioning on shared-cache multiprocessors as we have reported detailed cache analysis of Java applications.

6.7 Conclusions

There are several conclusions to be drawn from the range of experiments conducted in this work. The first is that general rules, such as using five and a half times the minimum heap or a half heap nursery for Java applications, can be helpful, but do not guarantee best results for individual applications. Following general rules can offer a guide on where to start, but may be unnecessarily wasteful of system resources.

Many Java applications can be run with smaller heaps than five and a half times their minimum requirement with little performance impact. Applications that seem to be overly sensitive to heap size may also be sensitive to locality, which can be improved by decreasing the nursery size. The trade-off of increasing garbage collection time can more drastically improve mutator time and, in the best case, allow a program to achieve equivalent performance at a

much smaller heap size.

We also show that although Java programs are demanding in terms of heap allocation, they are not necessarily sensitive to cache misses. Our large number of configurations shows that a linear model of runtime in terms of cache misses is quite accurate for many applications and many applications are already approaching their predicted runtime when no cache misses occur. Thus cache optimizations are expected to provide little improvement for many Java applications.

Many Java applications do not use the cache of modern processors effectively. This finding can lead to several possible applications. First, profiling using page coloring helps to identify how a particular application utilizes cache and can aid an application developer in tuning code. The applications that can be improved by decreasing the nursery size could also be tuned at the application level if objects were allocated in an order that improved their locality rather than relying on garbage collection to compact them.

The cache size sensitivity findings also are important to the operating system and hardware community. We demonstrate that cache can be managed as a resource and show that many Java applications need less cache than available on modern processors. System developers could also use this type of cache profiling when developing embedded systems. A cost analysis could be conducted with regard to performance input that may allow designers to use devices with smaller caches to meet the demands of the target applications.

Cache size and heap size are generally orthogonal parameters. Their contributions to runtime are additive, producing parallel lines when plotted with respect to one another. However, as a secondary effect, both are able to have an impact on cache alignment, which can affect cache performance. We generally find that these secondary effects are minimal with respect to their relative impact on overall performance.

Appendix: Java runtime sensitivity to nursery L2 cache partitioning

In this section we evaluate the use of page coloring to partition the cache for the address space within a single Java application. Based on our previous work where we found that

separating the nursery out for special treatment in an architecture with scratchpad (49), we use a similar division and reserve a small portion of the cache for nursery allocation. [5]

The main difference between this software approach and using a hardware scratchpad is that cache does not require any special treatment within a multitasking environment. However, one of the major benefits of scratchpad, that no traffic is generated to main memory, is lost in a page coloring partitioning scheme. Therefore in this section we are not targeting a reduction in memory traffic but instead attempt to evaluate whether reserving a portion of the cache for the nursery can benefit runtime performance.

The intuition behind this investigation is that Java applications allocate aggressively. They create a large number of objects (e.g. 60GB of heap allocation for xalan) and the survival rate is low. A continuous stream of allocation will displace other cache contents, (e.g. 1MB of allocation could displace the entire contents of a 1MB L2 cache) and depending on the access behavior of the application as a whole, this could disrupt other access patterns.

To separate the nursery from the rest of the memory space we group available page colors into two groups. One group is for the main part of the application, the mature space of the heap, the stack, instruction code of the JVM, bytecode and any other system libraries. The second group of page colors is reserved for use only by the nursery. To gain access to this nursery region the source code for JDK 1.6 was modified to make a separate *mmap* call to the operating system with a special flag to denote that the OS should allocate pages from the reserved page colors. Both regions can be sized to any integer multiple of page colors that cumulatively are equal to the number of available page colors on the system, or fewer if the comparison is for a smaller cache size.

The baseline configuration is when no colors are reserved for the nursery and the JVM allocates its resources as normal. The configurations that are compared directly with one another are those in which all command line parameters to the JVM are the same (e.g. heap size and nursery size) and the total cache available is equivalent (the total number of page

[5] The results of this evaluation do not support the division of cache partitioning for the nursery within a JVM. This section was not included in the submission to ACM Transactions on Architecture and Code Optimization. The evaluation is included in the thesis as the cache partitioning idea prompted the rest of the evaluation found in this chapter.

colors made available to the application is the same). In these configurations the number of colors for the nursery is at most one half the the total colors available, or half of the cache. For example, a configuration with 256KB of cache can be compared directly for configurations with 32KB, 64KB, 96KB and 128KB regions for the nursery.

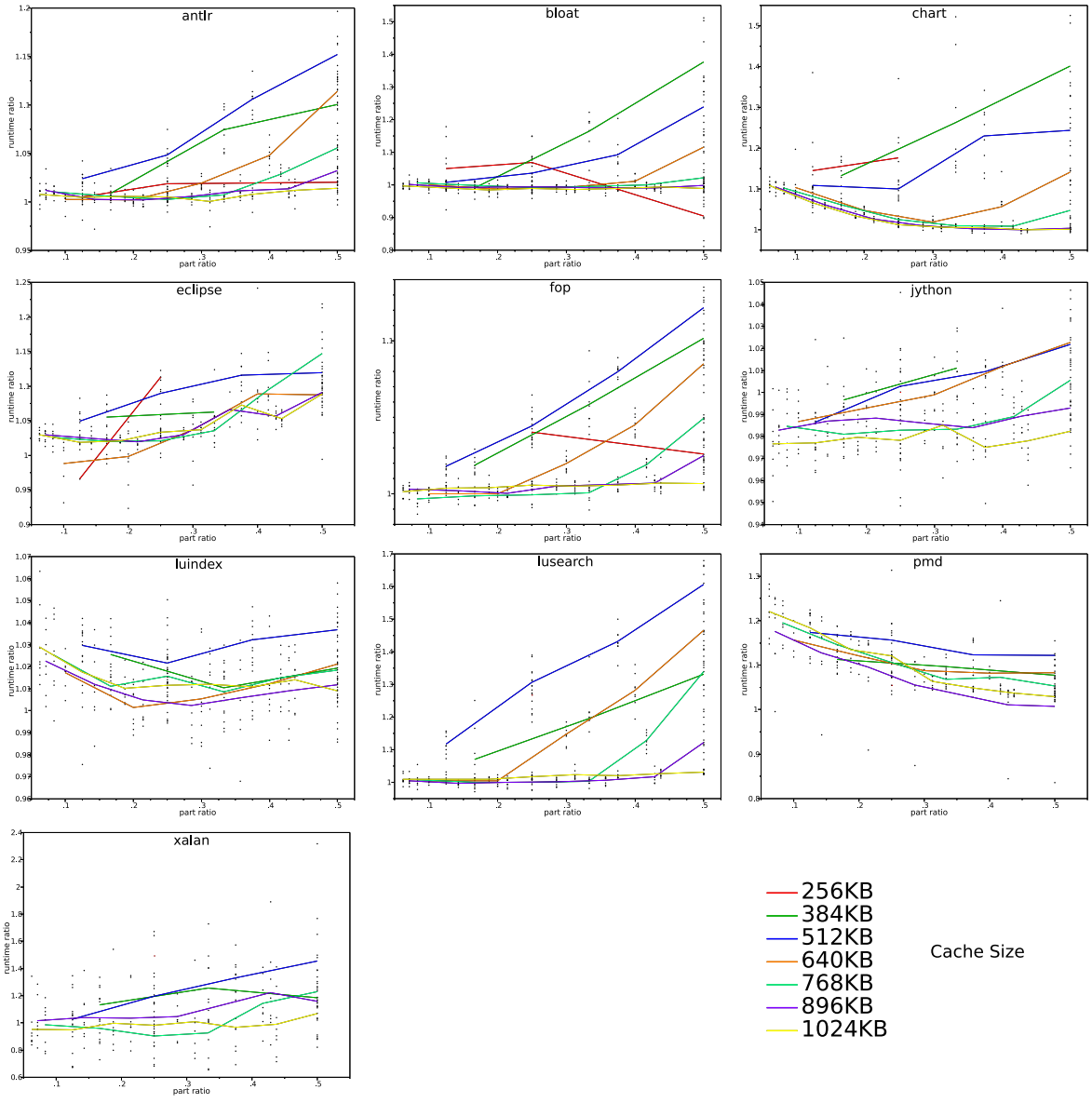

Figure 6.9 Runtime ratio vs. partition ratio by cache size – DaCapo

Figures 6.9 and **6.10** show comparisons of a large number of configurations where partitioning is used for the nursery. The y-axis is the runtime ratio verses an equivalent baseline

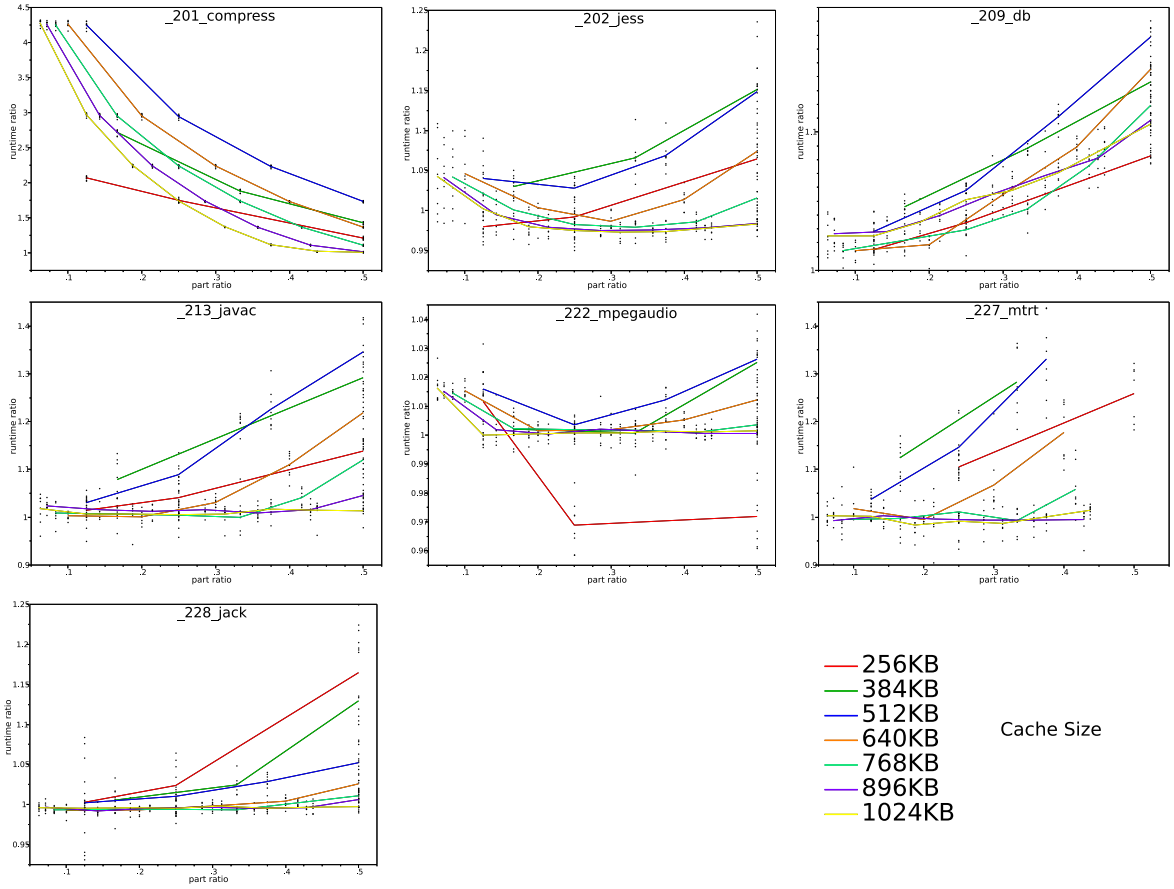

Figure 6.10 Runtime ratio vs. partition ratio by cache size – SPECjvm98

configuration. Runtime ratios greater than one indicate that the partitioning performs worse than the baseline and runtime ratios less than one indicate that partitioning performs better. The x-axis is the ratio of the nursery partition to the size of the cache for the particular configuration.

The scatter plot in each graph shows raw data point averages for the configurations. The fitted lines in these graphs group data by cache size. Most applications show severe performance degradation as the partition size approaches one half. However, *_201_compress* has an opposite trend and the smaller the partition, the worse the performance. Some applications such as *eclipse* and *xalan* show a few points on fitted lines that have improvement, but no trend indicates that this result is expected based on a particular cache size.

There are a couple of applications that do show a continuous improvement for set cache sizes. These applications are *jython* and *_202_jess*. In both applications, cache sizes above 768KB when partitioned yield a very modest 1% to 3% improvement. The best that most applications do is roughly break even when the cache size is above the ideal cache size reported above. This indicates that cache partitioning based on the nursery is generally not beneficial and simply is not affecting the application once the cache size is large enough.

Figures 6.11 and **6.12** show the same partitioning results except this time the fitted lines group data by heap size. The graphs are considerably more erratic, mostly because the data is influenced by both the cache size and heap size, and grouping is not easily shown in two dimensions. However, there are some interesting observations based on grouping by heap. The application *jython* shows several heap sizes that consistently see moderate improvement from nursery cache partitioning. Several other applications also show substantial improvements of 5% to 10% or even greater than 20% as in the case for *xalan* for heap size of three times the minimum heap.

To better show the range of impact of cache partitioning, the minimum and maximum runtime ratios are shown in **Table 6.8**. There are some cases of severe improvement such as for *bloat*, *pmd* and *xalan*. However, there are even more severe degradations such as for *_201_compress*, *lusearch* and again *xalan*.

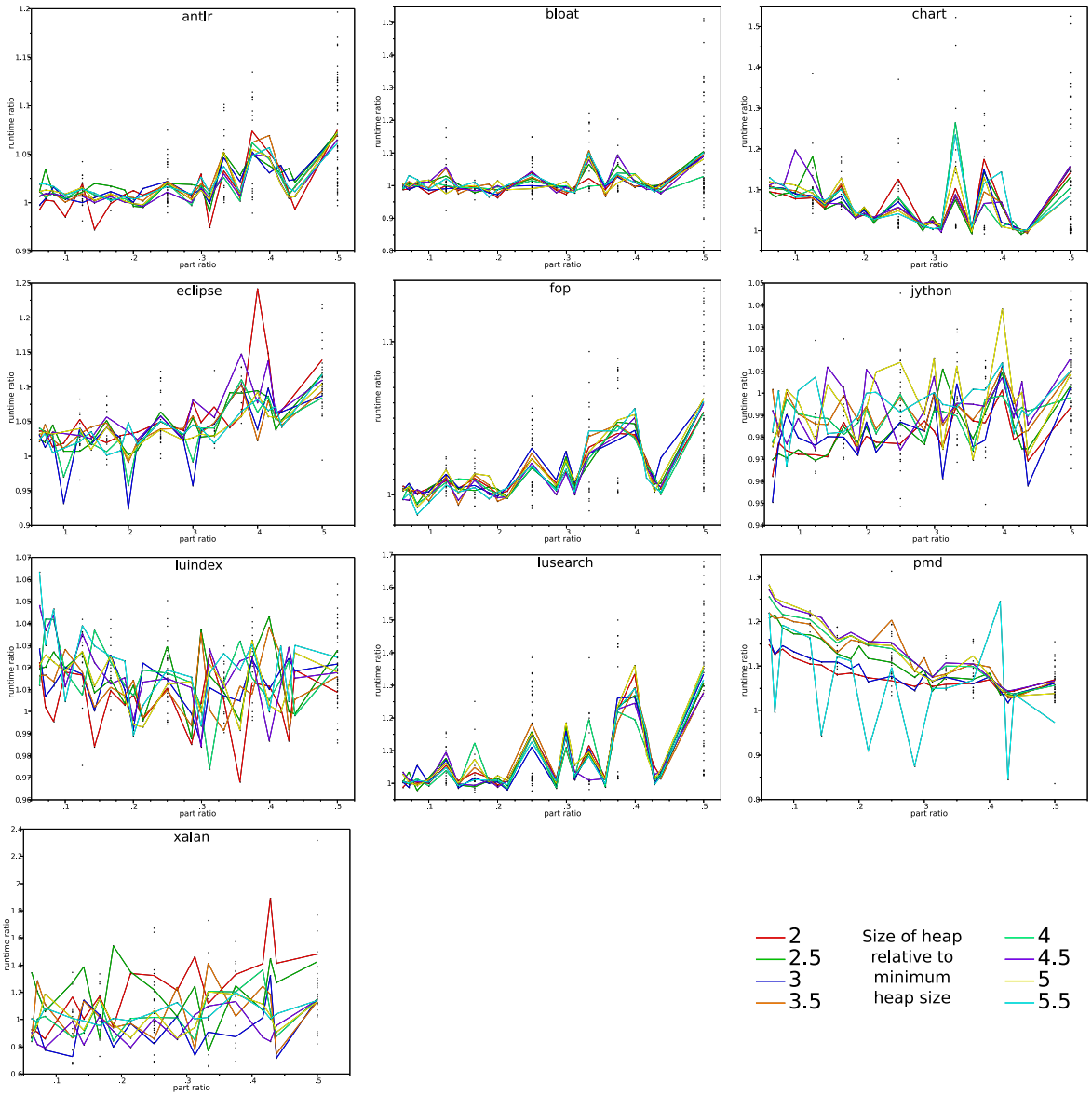

Figure 6.11 Runtime ratio vs. partition ratio by heap size – DaCapo

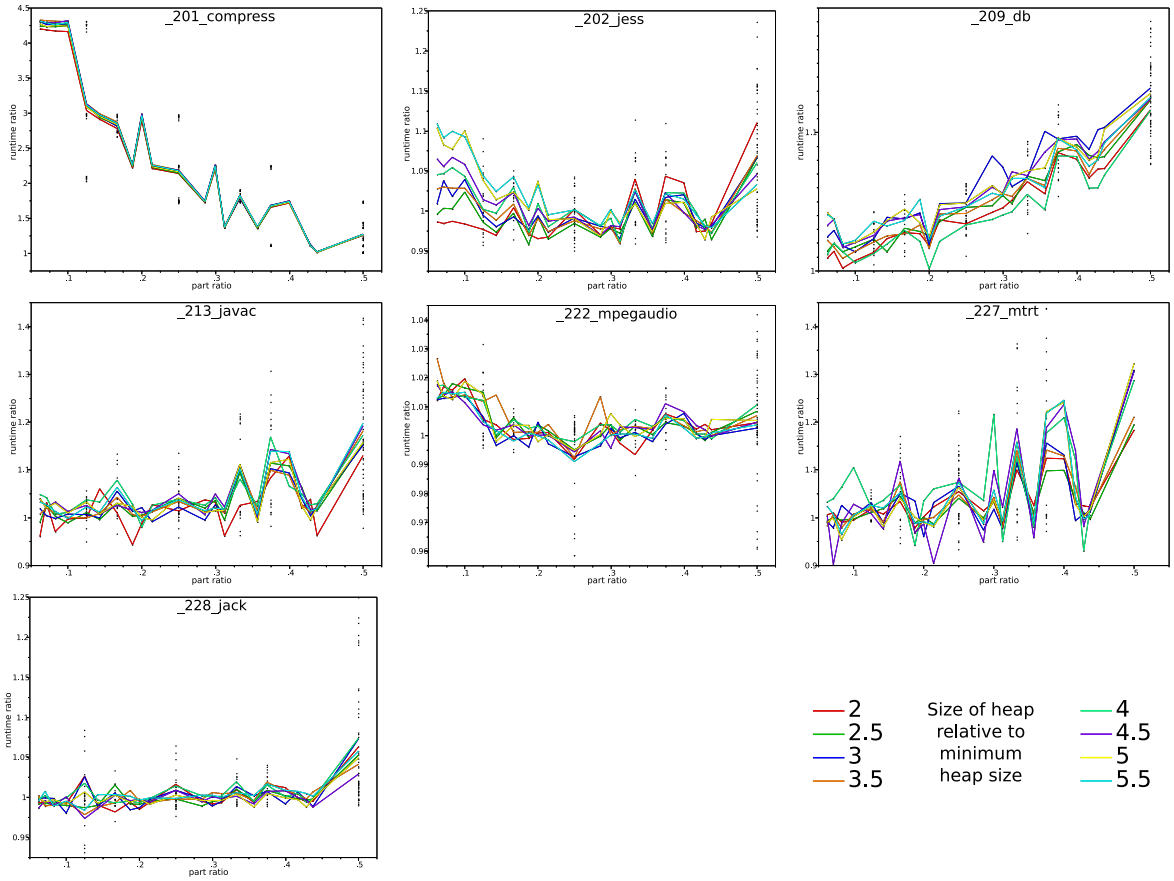

Figure 6.12 Runtime ratio vs. partition ratio by heap size – SPECjvm98

Table 6.8 Best and worst partitioning runtime ratios for DaCapo and SPECjvm98

DaCapo			**SPECjvm98**		
Benchmark	Best	Worst	Benchmark	Best	Worst
antlr	0.972	1.197	_201_compress	1.003	4.321
bloat	0.811	1.512	_202_jess	0.950	1.236
chart	0.991	1.525	_209_db	1.001	1.180
eclipse	0.924	1.242	_213_javac	0.943	1.417
fop	0.987	1.135	_222_mpegaudio	0.959	1.042
jython	0.949	1.046	_227_mtrt	0.901	1.438
luindex	0.968	1.063	_228_jack	0.931	1.249
lusearch	0.971	1.679			
pmd	0.836	1.313			
xalan	0.654	2.317			

There are a few conclusions that can be drawn from this analysis. The first is that cache partitioning based on the nursery is highly unpredictable and largely ineffective. However, as there are some cases where significant improvement is possible, the analysis points to the fact that partitioning may offer significant benefits to the runtime of an application. Because the division based on the nursery is unpredictable, we recommend further analysis on application behaviors that negatively interfere with cache contents and suggest that there may be possible application level cache partitioning that for certain applications may yield performance benefits.

CHAPTER 7 Conclusion

This chapter provides some final concluding remarks with regard to the work conducted in this thesis. All the work revolves around the core idea that allocation in Java applications has a distinct behavior within the system.

We first created an environment to help to identify that behavior on an object-by-object basis. That framework was utilized to take measurements for applications to help identify the locality of a Java system. We created a metric, *access density*, that helped to quantify the relative locality between objects of varying ages and sizes. We found that allocation of short-lived objects was indeed the source of a significant amount of the memory access behavior of Java applications.

Subsequently, we began an investigation into adding a scratchpad to a computer system to help isolate the allocation behavior and reduce the memory traffic of the system. This study found that a significant amount of traffic could be reduced through hardware segregation.

However, scratchpad is a difficult resource to manage in a multitasking environment and not generally applicable to the majority of systems in which Java is utilized. To build upon the work, we began to investigate other hardware techniques to exploit the allocation behavior of Java. We developed a flexible approach to alter cache replacement based on a common cache replacement technique. The approach, while it is effective in eliminating streaming interference within a cache, yields very limited results for allocation in Java applications. While it does not hurt performance, it provides small benefits only when bandwidth is limited.

We also evaluated a prefetching technique for Java allocation which yields substantial benefits in comparison to a configuration in which prefetching is not used. However, we also identify that hardware prefetching of modern systems is capable of handling much of the

allocation behavior in Java, and substantial benefits are only available in server environments where hardware prefetching is generally disabled.

Finally, in order to help reinforce the findings of our previous work, we also conducted an exhaustive performance analysis of Java applications with respect to the memory system on a real machine. To incorporate cache behavior, we utilize both hardware performance counters and a novel use of page coloring within the virtual memory system of an operating system to evaluate application sensitivity to cache size. These final results help to explain the results of prior simulations. A large number of configurations were possible because of the dramatic increase in speed over simulation. We were able to confirm the general practices and suggestions for running Java applications but also provide a very detailed cost analysis of performance improvements. We suggest that cache is also a resource that can be managed within the system and show that Java applications generally are incapable of effectively utilizing the cache of modern processors.

Although many contributions and new ideas are included in this thesis, the biggest contributions are the methodologies used in evaluating performance and the insights provided in the cost analysis of performance techniques.

BIBLIOGRAPHY

[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] A. Agarwal, D. Kranz, and V. Natarajan. Automatic partitioning of parallel loops and data arrays for distributed shared memory multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, 1993.

[3] J. Alakarhu and J. Niittylahti. Quick memory stream locality analysis. In *The 2002 45th Midwest Symposium on Circuits and Systems*, pages 519–522, 2002.

[4] J. Alakarhu and J. Niittylahti. Scalar metric for temporal locality and estimation of cache performance. In *Proceedings of Design, Automation and Test in Europe Conference and Exhibition*, pages 730–731, 2004.

[5] ALI Research Group - UMass Amherst/ UTexas Austin, http://osl-www.cs.umass.edu/DSS. *Dynamic Simple Scalar*, 2005.

[6] J. Anderson and M. S. Lam. Global optimizations for parallelism and locality on scalable paralel machines. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation*, 1993.

[7] M. Annavaram, G. Tyson, and E. Davidson. Instruction overhead and data locality effects in superscalar processors. In *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 95–100, 2000.

[8] A. W. Appel. Simple generational garbage collection and fast allocation. *Software–Practice and Experience*, 19:171–183, February 1989.

[9] 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.

[10] BEA Systems, Inc., http://www.bea.com. *BEA JRockit 5.0*, 2006.

[11] 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.

[12] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. Eliot, B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The dacapo benchmarks: java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 169–190, New York, NY, USA, 2006. ACM Press.

[13] E. Bugnion, J. Anderson, T. Mowry, M. Rosenblum, and M. S. Lam. Compiler-directed page coloring for multiprocessors. In *Proceedings of the Fifth International Symposium on Architectural Support for Programming Languages and Operating Systems*, 1996.

[14] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in java. In *In Proceedings of the International Converence on Parallel Architectures and Compiler Techniques*, 2001.

[15] B. Cahoon and K. S. McKinley. Simple and effective array prefetching for java. In *In Proceedings of the ACM Java Grande Forum*, 2002.

[16] 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.

[17] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 76, Washington, DC, USA, 2004. IEEE Computer Society.

[18] Y. Chou, L. Spracklen, and S. G. Abraham. Store memory-level parallelism optimizations for commercial applications. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 183–196, Washington, DC, USA, 2005. IEEE Computer Society.

[19] T. Conte and W. m W. Hwu. Benchmark characterization for experimental system evaluation. In *Proceedings of the 1990 Hawaii International Conference on System Sciences*, pages 6–18, 1990.

[20] R. Cucchiara and M. Piccardi. Exploiting image processing locality in cache pre-fetching. In *5th International Conference On High Performance Computing*, pages 466–472, 1998.

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

[26] J. Edler and M. D. Hill. *Dinero IV Trace-Driven Uniprocessor Cache Simulator*. http://www.cs.wisc.edu/ markhill/DineroIV, 2005.

[27] L. Eeckhaut, A. Georges, and K. D. Bosschere. How java programs interact with virtual machines at the microarchitecture levels. In *Proceedings of the International Conference on Object-Oriented Programming Systems Langages and Applications*, pages 169–186, October 2003.

[28] FreeBSD Foundation. The FreeBSD Project - Stable Release version 6.3, 2006.

[29] E. Gagnon. *SableVM*. http://sablevm.org, 2005.

[30] A. Georges, D. Buytaert, and L. Eeckhaut. Statistically rigorous java performance evaluation. In *In Proceedings of the International Conference on Object-Oriented Programming Systems Languages and Applications*, 2007.

[31] P. Grun, N. Dutt, and A. Nicolau. Access pattern based local memory customization for low power embedded systems. In *In Proceedings of Design, Automation and Test in Europe*, pages 778–784, 2001.

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

[33] M. Hauswirth, A. Diwan, P. F. Sweeney, and M. C. Mozer. Automating vertical profiling. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 281–296, New York, NY, USA, 2005. ACM Press.

[34] M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind. Vertical profiling: understanding the behavior of object-priented applications. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 251–269, New York, NY, USA, 2004. ACM Press.

[35] M. Hertz and E. D. Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 313–326, New York, NY, USA, 2005. ACM Press.

[36] M. Hertz, S. M. Blackburn, J. E. B. Moss, K. S. McKinley, and D. Stefanović. Error-free garbage collection traces: how to cheat and not get caught. In *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 140–151. ACM Press, 2002.

[37] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng. The garbage collection advantage: improving program locality. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 69–80, New York, NY, USA, 2004. ACM Press.

[38] S. Iacobovici, L. Spracklen, S. Kadambi, Y. Chou, and S. G. Abraham. Effective stream-based and execution-based data prefetching. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 1–11, New York, NY, USA, 2004. ACM Press.

[39] IBM T. J. Watson Research, http://oss.software.ibm.com/developerworks/oss/jikesrvm. *Jikes Research Virtual Machine*, 2005.

[40] R. Jones and R. Lins. *Garbage Collection - Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Ltd, 1996.

[41] I. Kadayif and M. Kandemir. Quasidynamic layout optimizations for improving data locality. *IEEE Transactions on Parallel and Distributed Systems*, 15(11):996–1011, 2004.

[42] Kaffe.org. *Kaffe*. http://www.kaffe.org, 2005.

[43] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. Improving locality using loop and data transformations in an integrated framework. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 285–296, 1998.

[44] S. F. Kaplan. Collecting whole-system reference traces of multiprogrammed and multi-threaded workloads. In *Proceedings of the fourth international workshop on Software and performance*, pages 228–237. ACM Press, 2004.

[45] K. Kennedy and U. Kremer. Automatic data layout for high performance fortran. In *Proceedings of Supercomputing*, 1995.

[46] R. Kessler and M. D. Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems*, 10(4):338–359, 1992.

[47] 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.

[48] K. Lawton, B. Denney, G. Alexander, T. Fries, D. Becker, and T. Butler. *bochs: The cross-platform IA-32 emulator*. http://bochs.sourceforge.net, 2005.

[49] C. S. Lebsack and J. M. Chang. Using scratchpad to exploit object locality in Java. In *ICCD '05: Proceedings of the International Conference on Computer Design*, 2005.

[50] C. S. Lebsack, Q. A. Jacobson, M. Sun, S. Srinivas, and J. M. Chang. Cache prefetching and replacement for java allocation. Under Review, 2008.

[51] J. H. Lee, J.-S. Lee, and S.-D. Kim. A selective temporal and aggressive spatial cache system based on time interval. In *In Proceedings of the International Conference on Computer Design*, pages 287–293, 2000.

[52] J.-H. Lee, S. woong Jeong, S.-D. Kim, and C. Weems. An intelligent cache system with hardware prefetching for high performance. *IEEE Transactions on Computers*, 52(5):607–616, 2003.

[53] W. Li and K. Pingali. Access normalization: Loop restructuring for numa computers. *ACM Transactions on Computer Systems*, 11(4), 1993.

[54] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *Proceedings of the 14th International Symposium on High-Performance Computer Architecture*, 2008.

[55] W.-F. Lin, S. K. Reinhardt, and D. Burger. Reducing dram latencies with an integrated memory hierarchy design. In *In Proceedings of the International Symposium on High Performance Computer Architecture*, 2001.

[56] J. S. Lizy Kurian John, Purnima Vasudevan. Workload characterization: Motivation, goals and methodology. In *Workload Characterization: Methodology and Case Studies*, 1998.

[57] G. Memik, M. Kandemir, A. Choudhary, and I. Kadayif. An integrated approach for improving cache behavior. In *Design, Automation and Test in Europe Conference and Exhibition*, pages 796–801, 2003.

[58] S. Microsystems. Java research license version 1.6. http://java.net/jrl.csp.

[59] T. C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD dissertation, Stanford University, Stanford, California, March 1994.

[60] T. N. Nguyen and Z. Li. Interprocedural analysis for loops sceduling and data allocation. *Parallel Computing, Special Issue on Languages and Compilers for Parallel Computers*, 24(3), 1998.

[61] P. Petrov and A. Orailoglu. Performance and power effectiveness in embedded processors customizable partitioned caches. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(11):1309–1318, 2001.

[62] C. Pyo, K.-W. Lee, H.-K. Han, and G. Lee. Reference distance as a metric for data locality. In *High Performance Computing on the Information Superhighway(HPC Asia)*, pages 151–156, 1997.

[63] T. Rauber and G. Runger. Program-based locality measures for scientific computing. In *Proceedings of the Parallel and Distributed Processing Symposium*, 2003.

[64] 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.

[65] Q. Samdani and M. Thornton. Cache resident data locality analysis. In *In Proceedings the 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 539–546, 2000.

[66] F. Sanchez, A. Gonzalez, and M. Velero. Static locality analysis for cache management. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 261–271, 1997.

[67] 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.

[68] SimpleScalar LLC., http://www.simplescalar.com. *SimpleScalar*, 2006.

[69] Standard Performance Evaluation Corporation, http://www.spec.org/. *SPECjvm98, SPEC2000, SPECjbb2005 Benchmarks*, 2006.

[70] D. Stefanović, M. Hertz, S. M. Blackburn, K. S. McKinley, and J. E. B. Moss. Older-first garbage collection in practice: evaluation in a Java virtual machine. In *Proceedings of the workshop on Memory system performance*, pages 25–36. ACM Press, 2002.

[71] Sun Mircrosystems, http://java.sun.com. *Sun HotSpot II*, 2005.

[72] P. F. Sweeney, M. Hauswirth, B. Cahoon, P. Cheng, A. Diwan, D. Grove, and M. Hind. Using hardware performance monitors to understand the behavior of java applications. In *In the Proceedings of the International Conference on Virtual Machines*, 2004.

[73] D. Tam, R. Azimi, L. Soares, and M. Stumm. Managing shared l2 caches on multicore systems in software. In *WIOSCA'07*, 2007.

[74] O. Temam. An algorithm for optimally exploiting spatial and temporal locality in upper memory levels. *IEEE Transactions on Computers*, 48(2):150–158, 1998.

[75] TIOBE Software, http://www.tiobe.com/tpci.htm. *TIOBE Programming Community Index*, 2005.

[76] 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.

[77] Tom's Hardware, http://www.tomshardware.com/2006/10/26/ intel_woodcrest_and_amd_opteron_battle_head_to_head/page8.html. *Intel Xeon and AMD Opteron Battle Head to Head*, 2006.

[78] R. Uhlig, R. Fishtein, O. Gershon, I. Hirsh, and H. Wang. Softsdv: A presilicon software development environment for the ia-64 architecture. In *In Intel Technology Journal*, 4th quarter 1999.

[79] Virtutech, http://www.virtutech.com. *Simics*, 2006.

[80] Z. Wang, D. Burger, K. S. McKinley, S. Reinhardt, and C. C. Weems. Guided region prefetching: A cooperative hardware/software approach. In *In Proceedings of the 30th International Symposium on Computer Architecture*, pages 388–398, 2003.

[81] Z. Wang, K. S. McKinley, A. Rosenberg, and C. C. Weems. Using the compiler to improve cache replacement decisions. In *In Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, pages 199–208, 2002.

[82] D. Weikle, S. McKee, and W. Wulf. Caches as filters: a new approach to cache analysis. In *Proceedings of the Sixth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 2–12, 1998.

[83] 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.

[84] 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.

[85] W. A. Wong and J.-L. Baer. Modified lru policies for improving second-level cache behavior. In *In Proceedings of the International Conference on High Performance Computer Architecture*, 2000.

[86] J. Xue and X. Vera. Efficient and accurate analytical modeling of whole-program data cache behavior. *ACM Transactions on Computers*, 53(5):547–566, 2004.

[87] L. Zhao, S. Makineni, R. Illikkal, R. Iyer, and L. Bhuyan. Efficient caching techniques for server network acceleration. In *Advanced Networking and Communications Hardware Workshop*, 2004.

ACKNOWLEDGEMENTS

During the course of my research, there were several people who lent their time and expertise to me. Bryan Venteicher, as an undergraduate research assistant, provided excellent help in modifying FreeBSD to support software cache partitioning. Michael Ciccotti, another undergraduate student, also aided in research framework development. Quinn Jacobson, Mingqiu Sun and Suresh Srinivas were an excellent team of collaborators during a research internship with Intel Corporation. I am pleased to have been given the opportunity to work with them and appreciate all of the guidance and feedback they provided. I am grateful to my committee, Arun Somani, Akhilesh Tyagi Zhao Zhang and Donna Kienzler for their guidance and feedback on research, teaching and career goals. Finally, I would like to thank my advisor, Morris Chang, who lured me to a beautiful campus in the middle of Iowa to pursue grand dreams.

My work also was supported through graduate student stipends supplied by the Electrical and Computer Engineering Department through teaching assistant positions, by Dr. Morris Chang and the National Science Foundation[1] under Grant Nos. 0296131 (ITR) 0219870 (ITR) and 0098235, by a graduate research internship with Intel Corporation and by the United States Department of Education through the Graduate Assistance in Areas of National Need grant.

[1]Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.