The Dissertation Committee for Jennifer Bedke Sartor
certifies that this is the approved version of the following dissertation:

Exploiting Language Abstraction to Optimize Memory Efficiency

Committee:

Kathryn S. McKinley, Supervisor

Stephen M. Blackburn, Supervisor

Stephen W. Keckler

Emmett Witchel

Martin Hirzel

Exploiting Language Abstraction to Optimize Memory Efficiency

by

Jennifer Bedke Sartor, B.S., M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of
The University of Texas at Austin
in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN
August 2010

Dedicated to my Grandpa Bedke who put education first,
and to my Mom who believes in me.

Acknowledgments

I wish to thank the multitudes of people who helped me to get through this challenging process of obtaining my PhD.

First, I would like to sincerely thank my advisors, Kathryn McKinley and Steve Blackburn. Kathryn took a chance on me when I came into her office and told her as a Masters student that she was the only person with whom I wanted to do research. Her recommendation letter is the main reason I was admitted into the CS PhD program at UT. Her enthusiasm and unwavering confidence in me, especially when my own failed, and her guidance and mentoring about the world of research and publishing has made me the academic I am today.

I would not have been able to make it through my PhD without Steve's tireless hours of help and patient teaching. While Kathryn was my high-level mentor, Steve met with me sometimes daily to work out low-level details of implementation and experimentation. Although Steve lives across the globe in Australia, he always made time for me, pushing me a little further than I thought I was capable of.

I would like to thank Martin Hirzel as a collaborator and committee member. His invaluable ideas are what started me along the path to my thesis research. Thank you to Steve Keckler for supervising me as a teaching assistant and serving on my committee. Thank you to Emmett Witchel for being a member of my committee.

I would not have been able to achieve the research contributions I have without the Jikes Research Virtual Machine. Thank you to the development team, and especially Daniel Frampton for your abundant help with this beast of a runtime. Also, thank you to the DaCapo group. Thank you also to Intel and IBM for my research internships and funding assistance. I appreciate the opportunity afforded to me by the National Science Foundation to go study in Australia with the EAPSI program.

I would like to thank my undergraduate professors in Computer Science who encouraged and inspired me to first teach, then purse degrees in CS in addition to my studies in mathematics. Thank you to my undergrad compatriots in CS - particularly Christine Jabara for your support and awesomeness.

I would not have enjoyed grad school or progressed as much without the support of many graduate school peers, including Alison, Serita, Shel, Harry, Greg, Matt, Nick, Justin, Allen, Peter, Kurt, Chris, Peggy, Igor, Mike, Ben, Byeong, Jungwoo, Suriya, Katie, Bert, Dong, Yang, Phoebe, Naveen and others. I especially want to thank Maria Jump for being there for me through countless frustrations, always offering a shoulder, an ear, an exercise diversion, and a trip to the tea house to make things better. I would like to thank the fellow women in CS for showing me that our path is possible in this male-dominated field. I would also like to thank the invaluable help of our CS administration, including Gem Naivar, Gloria Ramirez and Lydia Griffith.

I would not be sane today without friends outside of CS who have encouraged my hobbies and athletic pursuits. In particular, I have shared many hours of fun and understanding with Negin and Carol. Other friends include those in Kung Fu, my half-marathon and triathlon training, Hill Country Outdoor (HCO) friends, friends from coffee shops where I have worked long hours, friends from the farmer's market, roommates, and others.

Finally, I need to thank my family for being there for me. Grandpa Bedke started to teach me math when I was entering grade school, and valued education above all else. He inspired me to be a dedicated student and life-long learner. Grandma Bedke has been a strong role-model for me throughout my life, always putting her family first. She is the kindest, most generous person I know, and I would not be who I am without her calm presence and support in my life. I have also received the support of my siblings, brothers Matt and Scott and sister Mindy, my Dad who never forgets to tell me how proud he is of me, and my Bedke aunts and cousins who have always inspired excellence. Finally, I could not have completed this PhD without my Mom, my best friend. She has seen me through thick and thin, ready to drop everything to be there for me. Her love and support have not only made me the hard-worker I am today, but have shown me that I can do anything I put my mind to, including this PhD.

Exploiting Language Abstraction to Optimize Memory Efficiency

Publication No. ____________

Jennifer Bedke Sartor, Ph.D.
The University of Texas at Austin, 2010

Supervisors: Kathryn S. McKinley
Stephen M. Blackburn

The programming language and underlying hardware determine application performance, and both are undergoing revolutionary shifts. As applications have become more sophisticated and capable, programmers have chosen managed languages in many domains for ease of development. These languages abstract memory management from the programmer, which can introduce time and space overhead but also provide opportunities for dynamic optimization. Optimizing memory performance is in part paramount because hardware is reaching physical limits. Recent trends towards chip multiprocessor machines exacerbate the memory system bottleneck because they are adding cores without adding commensurate bandwidth. Both language and architecture trends add stress to the memory system and degrade application performance.

This dissertation exploits the language abstraction to analyze and optimize memory efficiency on emerging hardware. We study the sources of memory inefficiencies on two levels: heap data and hardware storage traffic. We design and implement optimizations that change the heap layout of arrays, and use program semantics to eliminate useless memory traffic. These techniques improve memory system efficiency and performance.

We first quantitatively characterize the problem by comparing many data compression algorithms and their combinations in a limit study of Java benchmarks. We find that arrays are a dominant source of heap inefficiency. We introduce z-rays, a new array layout design, to bridge the gap between fast access, space efficiency and predictability. Z-rays facilitate compression and offer flexibility, and time and space efficiency.

We find that there is a semantic mismatch between managed languages, with their rapid allocation rates, and current hardware, causing unnecessary and excessive traffic in the memory subsystem. We take advantage of the garbage collector's identification of dead data regions, communicating information to the caches to eliminate useless traffic to memory. By reducing traffic and bandwidth, we improve performance.

We show that the memory abstraction in managed languages is not just a cost to be borne, but an opportunity to alleviate the memory bottleneck. This thesis shows how to exploit this abstraction to improve space and time efficiency and overcome the memory wall. We enhance the productivity and performance of ubiquitous managed languages on current and future architectures.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Chip manufacturing has recently undergone a major shift as silicon technology hits its physical limits. Dennard's scaling rule that says power and switching time shrink as fabrication technology shrinks, has come to a halt [20]. Processors can no longer depend on smaller transistors yielding less power and faster clocks which translate into better performance. Now manufacturers are creating chip multiprocessors to keep attaining performance benefits, and producing more embedded devices to meet demand for new product capabilities. The consequences of this shift are an increase in power consumption, and increased stress on the memory subsystem which has always lagged behind processor speeds. Previously, hardware has been able to hide some of the memory latency behind non-blocking caches, prefetching, and out-of-order processors. As hardware is hitting its physical limits, the memory bottleneck is more exposed. Industry's trends towards more cores which require parallel communication has throttled scalability and performance. Since cache and memory consume a disproportionate amount of area and are expensive [65], the demand for memory efficiency is likely to remain constant or increase in the future.

To program these machines, developers are increasingly turning to managed languages, such as Java [70], due to their *productivity* benefits, which include reduced errors through memory management, reliability due to pointer disciplines, and portability. Managed languages are not known for their memory efficiency and are therefore in conflict with hardware trends. However, because Java provides a high-level abstraction to the programmer, it is able to change the underlying implementation and management of memory in the managed runtime, and therefore offers opportunities for dynamically optimizing applications.

In this thesis, we explore sources of memory inefficiency both in the Java heap and in how it uses the underlying memory system. We exploit the memory transparency of managed languages to change the layout of the heap and communicate with hardware to reduce memory traffic, optimizing memory efficiency and improving the performance of modern programs on current and future architectures.

1.1 Heap Data Organization

We first analyze the inefficiencies of modern program heap data. Heap organization inefficiencies are a symptom of the complexity of software and a by-product of high-level language abstraction.

1.1.1 The Problem

Researchers have characterized Java memory composition and usage patterns [13, 28, 54], and found that the heap is intricately connected and bloated with spurious data and connection glue. To address this problem, many researchers have proposed and measured specific compression approaches [5, 6, 21, 53, 66, 78]. We instead study memory efficiency with a global view. We create models for known and new heap data compression techniques, and then quantitatively compare them in a limit study of Java benchmarks [63]. We periodically snapshot the heap during garbage collection, and post-process these snapshots to calculate memory savings for different compression models and for the first time, hybrid techniques combining compression models. Overall we see that arrays take up the majority of space in the heap and yield larger compression opportunities, so we focus our optimization efforts on arrays. We find that zero-based array compression saves the most memory as an individual compression model: on average 41% of the application heap. Our novel hybrid technique combines six compression models, and saves 52% of the application heap. The results of this limit study suggest focusing on reorganizing and compressing arrays to help contain bloat and make managed languages more memory efficient.

Arrays are the ubiquitous method for organizing indexed data; first invented by Konrad Zuse [10] in 1946, they are used in every modern programming language. Traditional implementations use contiguous storage, which often wastes space and leads to unpredictable performance. For example, large arrays cause fragmentation, which can trigger premature out-of-memory errors and make it impossible for real-time collectors to offer provable time and space bounds. Over-provisioning and redundancy in arrays wastes space, as shown in our limit study. In managed languages, garbage collection uses copying to coalesce free space and reduce fragmentation. However, copying and scanning arrays incur expensive unpredictable collector pause times, and make it impossible to guarantee real-time deadlines.

Managed languages, such as Java and C#, give programmers a high-level contiguous array abstraction that hides implementation details and offers virtual machines (VMs) an opportunity to ameliorate the above problems. To meet space efficiency and time predictability, researchers proposed *discontiguous* arrays, which divide arrays into indexed chunks [8, 21, 67]. Siebert's design organizes array memory in trees to reduce fragmentation, but requires an expensive tree traversal for every array access [67]. Bacon et al. and Pizlo et al. use a single level of indirection to fixed-size *arraylets* [8, 57]. Chen et al. contemporaneously invented arraylets to aggressively compress arrays during allocation and collection, and decompress on demand for memory-constrained embedded systems [21]. All prior work introduces substantial overheads due to indirection upon access. Regardless, three production Java Virtual Machines (JVMs) already use discontiguous arrays to achieve real-time bounds: IBM WebSphere Real Time [40, 8], AICAS Jamaica VM [3, 67], and Fiji VM [31, 57]. Thus, although discontiguous arrays are needed for their *flexibility*, which achieves space bounds and time predictability, so far they have sacrificed throughput and time *efficiency*.

1.1.2 Our Solution

We present *z-rays*, a discontiguous array design and JVM implementation that combines flexibility, memory efficiency, and performance [62]. Z-rays store indirection pointers to arraylets in a *spine*. Z-rays optimizations include: *a novel* first-*N optimization, lazy allocation, zero compression, fast array copy,* and *copy-on-write*. Our novel first-*N* optimization inlines the first *N* bytes of the array into the spine for direct access. First-*N* eliminates the majority of pointer indirections because access statistics show that many arrays are small and most array accesses, even to large arrays, fall within the first 4KB. These properties are similar to file access properties exploited by Unix indexed files, which organize *inodes* to have direct pointers to initial data, taking less indirection to access small files and the beginning of large files [58]. First-*N* is our most effective optimization. Besides making indirections rare, it makes other optimizations more effective. For example, with lazy allocation, the allocator lazily creates arraylet upon the first non-zero write. This additional indirection logic degrades performance in prior work, but improves performance when used together with first-*N*.

Our design and implementation are configurable and users may choose from the optimizations and tune first-*N* and arraylet sizes. Z-rays thus bridge the time and space gap between contiguous and discontiguous layouts. Our experimental results on 19 SPEC and DaCapo Java benchmarks show that our best z-ray configuration adds an average of 12.7% overhead, including a *reduction* in garbage collection cost of 11.3% due to reduced space consumption. In contrast, we show that previously proposed designs have overheads *two to three times* higher than z-rays.

Because making arrays discontiguous has ramifications on heap space efficiency, we further investigate the effect on fragmentation, a first-order concern in memory management in high-level languages. We present formal equations for quantitatively measuring two kinds of fragmentation, internal and external. We then discuss how to measure the worst-case theoretical limit of fragmentation in our system, as well as how to measure fragmentation levels in practice with our benchmarks.

Z-rays are immediately applicable to discontiguous arrays in embedded and real-time systems, since they improve flexibility, space efficiency, and add time efficiency. Our results demonstrate that z-rays achieve both performance and flexibility, making them an attractive building block for language implementation on general-purpose architectures.

1.2 Memory Subsystem Traffic

As mentioned above, industry's trend toward chip multiprocessors (CMPs) has put pressure on memory bandwidth and traffic, which is exacerbated by modern application's frequent allocation. We analyze the impact of high-level language's rapid allocation of short-lived data on the cache memory hierarchy.

1.2.1 The Problem

As applications evolve, they have a seemingly insatiable need for memory. For example, recent studies of managed and native programs executing on CMPs show that memory bandwidth limits performance and scaling [41, 61, 77]. In particular, they identify an *allocation wall* due to high allocation rates. Programmers are writing in a style that rapidly allocates short-lived objects. They have been encouraged in this style in part because generational garbage collectors for managed languages and region allocators make allocation relatively cheap on uniprocessors. Unfortunately, this style creates an *object stream* [17], a small irregular window of temporal and spatial reuse stemming from allocation. This irregularity makes traditional remedies for streaming data unsuitable. Object streams march linearly through the cache, displacing other useful data. Because object streams start with a write, when the cache evicts a line in an object stream, the line is dirty and must be written. Since objects in the object stream are usually short-lived, the line containing them is often dead when it is evicted and never read again. The write-back is therefore useless.

We first quantify this problem by measuring the number of *essential* and *useless* write-backs to memory using the DaCapo Java benchmarks. A write-back is essential when the program subsequently reads the written data. A write-back is useless when the program never again reads the written data. On average, a surprising 88% of write-backs are *useless*.

1.2.2 Our Solution

This result motivates a software-hardware cooperative approach because hardware alone can never predict which program writes are useless. The memory manager does know, however, which regions of memory are dead. We design an *invalidation* approach in which the memory manager communicates *candidate regions* and when they are invalid. The hardware support uses existing cache line valid bits. It adds a candidate byte to each cache line and a few control registers to store candidate region information. We focus on using the nursery in a generational collector as the candidate region for three reasons. (1) Most useless writes are due to fresh allocation in the nursery (see Section 2.1). (2) Frequent nursery collections offer an opportunity to communicate dead regions. (3) The nursery is usually stored in one contiguous region, which simplifies the hardware. However, the approach generalizes to region allocators used in both automatic and explicit memory managers and to multiple disjoint coarse-grained regions.

We design several powerful optimizations with our simple interface for invalidation. For our *invalidation* optimization, the memory manager identifies a dead region and the hardware invalidates resident cache lines in the region. Invalidation expedites early eviction of these cache lines, improving cache efficiency. Furthermore, it prevents useless write-backs of resident cache lines that contain written, but dead objects. We design *in-cache zeroing* of all objects at a cache line granularity in a candidate region in a uniprocessor setting. Java and most managed languages require zero-initialization of all objects. Automating initialization in hardware eliminates the software initializing instructions that write, read, and fetch zeros, and their cache effects. We also experiment with a less invasive approach that reduces cache pollution by changing the cache line placement of lines in the candidate region; *priority biasing* a candidate region favors placing them lower in the least-recently-used (LRU) positions of the cache sets for eager eviction. This approach is not nearly as effective as invalidation. These cooperative optimizations rely on program semantics to improve cache efficiency and eliminate useless memory traffic.

We implement our software design in Jikes RVM [4] with the default generational immix garbage collector [18] and measure DaCapo Java benchmarks [13]. We implement the hardware design in Valgrind [56], a binary re-writer and high-level simulator, and in PTLsim [76], a cycle-level simulator. Since Valgrind is several orders of magnitude faster than PTLsim, we use it to characterize cache effects across many configurations. Using a range of cache sizes and memory manager configurations, Valgrind results show that cache invalidation of a candidate region eliminates on average 26% of all write-backs, and for some configurations invalidation saves 46% of write-backs. These improvements come equally from improving cache replacement decisions and eliminating useless write-backs. PTLsim shows these savings translate into a 17% reduction in total memory traffic, an average performance improvement of 7% over all configurations, and 13% when bandwidth is limited. For the DaCapo jython benchmark, cooperative invalidation eliminates 78% of write-backs and when bandwidth is limited, improves performance by 35%.

Although other researchers have proposed cooperative cache replacement hints [74, 73], as far as we know, only Isen and John use software semantics for invalidation [42]. They show improvements in energy by reducing useless write-backs for C programs, but not performance or cache replacement decisions. Their approach works at a cache line granularity. Each allocation and free communicates with the hardware, and the hardware stores a map of allocations to cache lines. This map requires fine-grained tracking of objects and their alignment in cache lines. In contrast to a map proportional to allocations, our approach adds only a few registers to hardware and one byte to each cache line. We exploit contiguous region allocation, used exclusively for the nursery in high performance generational collectors and for explicit region management in demanding C programs, such as the Apache web server.

The contributions of this work are thus (1) to identify an enormous opportunity to eliminate useless memory traffic using program semantics, and (2) the design and demonstration of cooperative software-hardware invalidation. This approach significantly reduces this traffic and improves performance with very modest software and hardware modifications. Memory, power, and energy efficiency are becoming increasingly important as industry moves into the CMP and power-constrained era. Achieving high memory efficiency requires software and hardware to cooperate to exploit program semantics. We show cooperation is a fruitful optimization area and ripe for exploration.

1.3 Impact

It is commonly accepted that the tradeoff for higher productivity in wide-spread managed languages is reduced performance, in time and space, especially due to its interaction in memory with current hardware's use of more cores. However, we take advantage of both the high-level abstraction that the managed language provides to the programmer, and the dynamic optimization opportunity of the underlying runtime environment to reduce this sacrifice. To ameliorate heap inefficiency problems, we change the layout of arrays to be more flexible, with both time and space efficiency. To reduce memory traffic and bandwidth on current hardware, which has been shown to be a major bottleneck to scalability and performance, we introduce a software-hardware cooperative optimization to eliminate costs from programmatic dead data in the memory subsystem. Combined, we improve memory efficiency and performance for sophisticated modern applications running on emerging hardware, easing the conflict between the productivity of managed languages and current computer architectures.

1.4 Organization

This thesis is organized as follows. Chapter 2 details background necessary to understand our research, including details of Java and the virtual machine, and studies of traffic and bandwidth between the cache hierarchy and memory. Chapter 3 discusses our heap data compression limit study in detail, including experimental results. We present z-rays, our optimized discontiguous array design in Chapter 4, and study the ramifications of our discontiguous layout on fragmentation in detail in Chapter 5. Chapter 6 describes our research on our software-hardware cooperative optimization to reducing traffic in the memory subsystem. We conclude in Chapter 7.

Chapter 2

Background

This section briefly discusses the background for studying both heap and memory inefficiencies in the context of Java. Our techniques are applicable to other managed languages as they take advantage of the memory virtualization to perform optimization. For our limit study, implementation of z-rays, and saving useless write-backs, we use Jikes Research Virtual Machine (RVM) [4], a high performance Java-in-Java virtual machine, but our use of Jikes RVM is not integral to our approach.

We first summarize heap organization and memory management pertinent to our optimizations. The chapter then presents simulation results for Java benchmarks that show that a large fraction of all write-backs are useless. It presents time series measurements for two Java Virtual Machines (JVMs) that show these write-backs are responsible for peak bandwidth.

Java Heap. We consider a conventional representation for dynamically allocated objects in a program. The heap contains two kinds of objects: class instances with fields and arrays with elements. Each object occupies a contiguous chunk of memory that consists of its fields or elements plus a header. We assume a conventional two-word object header with type information, garbage collector (GC) bits, and bookkeeping information for locking and hashing. Arrays have a third header word to store the length. Since we perform our experiments with Jikes, a Java-in-Java virtual machine (JVM), the heap contains both application and JVM objects [4]. Experimental results for our compression limit study will have numbers for both the total heap with application and JVM objects, and just the application heap.

Java Arrays. Because we are looking at the space inefficiencies particularly of arrays, and we want to change their layout in memory, it is important to know how Java currently represents arrays. All arrays in Java are one-dimensional; multi-dimensional arrays are implemented as arrays of references to arrays. Hence, Java explicitly exposes its discontiguous implementation of array dimensions greater than one. Accesses to these arrays require an indirection for each dimension greater than one, whereas languages like C and Fortran compute array offsets from bounds and index expressions, without indirection. Java directly supports nine array types: arrays of each of Java's eight primitive types (`boolean`, `byte`, `float`, etc.), and arrays of references. Java enforces array bounds with bounds checks, and enforces co-variance on reference arrays by cast checks on stores to reference arrays. The programmer cannot directly access the underlying implementation of an array because (1) Java does not have pointers (unlike C), and (2) native code accesses to Java must use the Java Native Interface (JNI). These factors combine to make discontiguous array representations feasible in managed languages, including not only Java, but others such as C#, JavaScript, and Python.

Allocation. Java memory managers conventionally use either a bump-pointer allocator or a free list, and may copy objects during garbage collection. The contents of objects are zero-initialized. Because copying large objects is expensive and typically size and lifetime are correlated, large objects are usually allocated into a distinct non-moving space that is managed at the page granularity using operating system support. One of the primary motivations for discontiguous arrays in prior work is that they reduce fragmentation, since large arrays are implemented in terms of discontiguous fixed-size chunks of storage. The base version of Jikes RVM we use has a single non-moving large-object space for objects 8KB and larger.

Garbage Collection. Garbage collection starts with the roots (statics, stacks and registers) and performs a transitive closure, tracing through the object reachability graph to identify live data in the heap. Objects not marked as live can be reclaimed and used for future allocation. The garbage collector must be aware of the underlying structure of arrays when it scans pointers to find live objects, possibly copies arrays, and frees memory. Discontiguous arrays in general, and z-rays in particular are independent of any specific garbage collection algorithm. We chose to evaluate our implementation in the context of a generational garbage collector, which is used by most production JVMs. A generational garbage collector leverages the weak generational hypothesis that most objects die young [50, 72]. The collector bump-allocates objects into a nursery which is frequently collected. When the nursery fills up, the collector copies surviving objects into a mature space. Most objects do not survive, and the nursery space is freed en masse to be used again. The mature space has its own allocation and reclamation organization, often segregating objects based on size classes and using a free list to keep track of memory (a mark-sweep organization). To avoid scanning the mature space for a nursery collection, a generational write barrier records pointers from the mature space to the nursery space [50, 72]. By adding the source of these pointers to the root set, the garbage collector can collect just the nursery, reclaiming a large portion of the heap quickly. Once frequent nursery collections fill the mature space, a *full heap collection* scavenges the entire heap from the original root set.

Read and Write Barriers. Read and write barriers are actions performed upon every load or store, respectively. Java has barriers for bounds checks on every array read and write (shown in Figure 4.3(a)), cast checks on every reference array write, and the generational write barrier described above. Java optimizing compilers eliminate provably redundant checks [19, 43]. Jikes RVM implements a rich set of read and write barriers on arrays of references. Z-rays require additional barriers for arrays of primitives, which presented a significant engineering challenge since these are a new type of barrier for our and most JVMs (for more details see Section 4.3.2).

Allocation Regimes and Object Streams. The highest performance memory managers for explicit and garbage collected languages have converged on a similar allocation regime. Explicit memory managers use region allocators when possible [11] and garbage collectors use generational collectors with a copying nursery [14]. Both allocate contiguously into a large region of memory. Thus, objects allocated together in time are adjacent in virtual and physical memory, which provides the program with cache locality benefits [14].

In explicitly managed languages, this region allocation is only possible when all the objects die together. However, this case is relatively frequent [11] and is a popular approach in servers for managing transactions, such as the Zend PHP VM and the Apache web server.

Both allocation regimes result in an *object stream* [17]. An object stream is a small irregular window of temporal and spatial reuse stemming from an initial allocation. Traditional remedies for streaming data, such as cache bypassing and vector registers, are unsuitable because the window of accesses is too irregular, spanning multiple instructions that are highly dependent on the allocation and usage context. Object streams march linearly through the cache and displace other useful data. They have a short window of use. When the cache later evicts these lines, they are dirty and must be written to memory. Because most objects in these streams and their cache lines are short-lived, the line is usually — but not always — dead and therefore never read again.

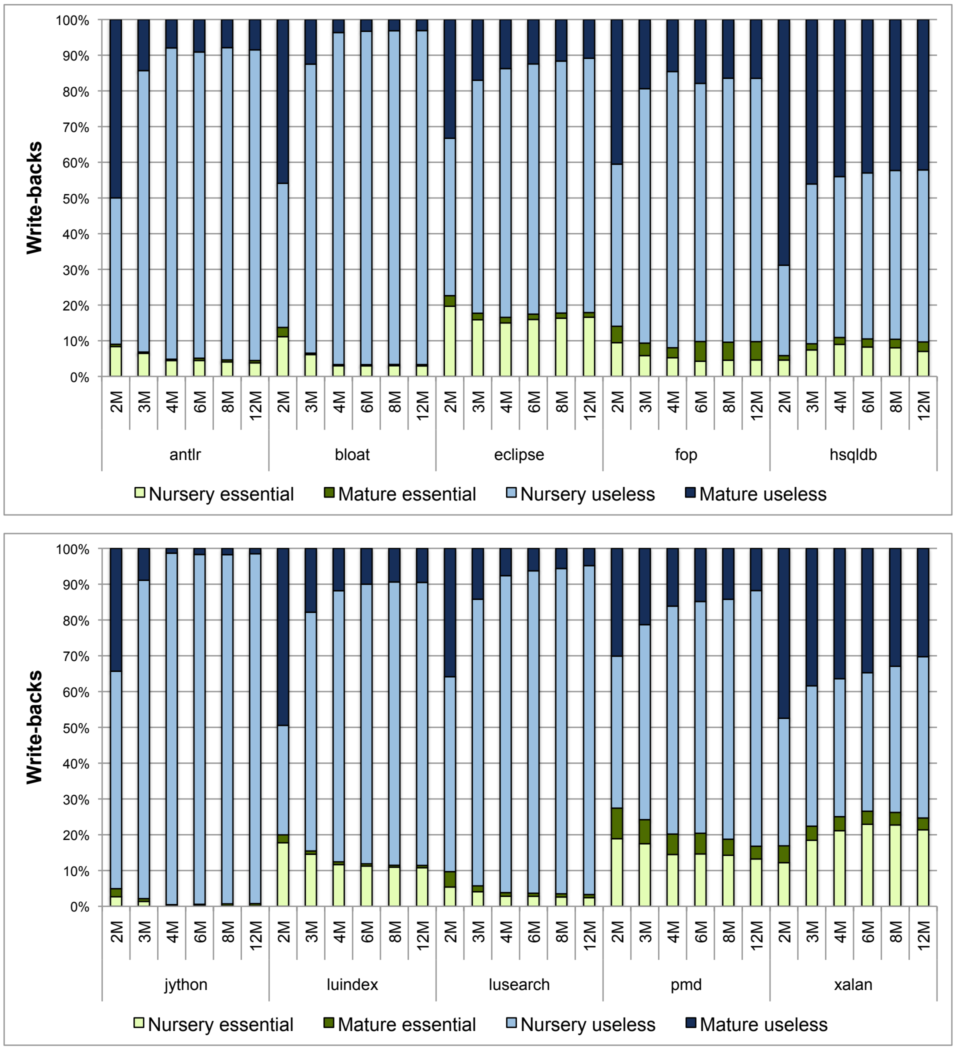


Figure 2.1: Breaking down essential versus useless write-backs into mature and nursery data at various nursery sizes for a 4MB L2.

2.1 Useless Write-Backs

This section analyzes how much traffic between the last level cache and memory is necessary. We perform our experiments in the context of a generational garbage collector, using the default generational immix collector in Jikes RVM [4, 13]. We show that the vast majority of write-backs are useless. Most of these accesses come from the nursery, motivating an approach that focuses on the nursery.

We examine write-backs on a word-granularity and place them into two categories. An *essential write-back* occurs when a word is written to main memory upon eviction from the last level cache, and the application subsequently reads the evicted data. A *useless write-back* occurs when the data written out upon eviction is never read again by the application–the application may well overwrite the data with a new value. We gather these statistics in the Valgrind simulator [56] using a bitmap for each word in memory. We keep track of when a word is written to memory, and whether it is subsequently read, over-written without being read in the cache, or never touched again.

Figure 2.1 shows the essential versus useless write-backs to memory at the granularity of a word. We model a split data and instruction L1 cache each with 32KB, 64 byte line, and 8-way set associativity, and a shared L2 with 4MB, 64 byte line, and 16-way set associativity. In these experiments, we vary the nursery size, and keep the L1 and L2 constant. We experiment using the DaCapo benchmark suite, version 2006-10-MR2 [13]. (Section 6.2.1 contains more details on methodology.) The figure breaks essential and useless write-backs down into addresses in the nursery and mature address ranges. We vary the nursery size from 2MB to 12MB. Across all nursery sizes, the overwhelming majority of all write-backs are useless, on average 88%. 68% of all useless write-backs come from nursery addresses.

Most essential data written back to memory on average is also from nursery addresses: 9% of all write-backs. Only 2.2% of all write-backs are essential and mature. These results demonstrate that the object stream is displacing useful data, mostly interfering with itself in the nursery, but also in the mature space. Benchmarks such as hsqldb and xalan have a high percentage of mature write-backs in general, and may benefit from an approach that targets the mature space. The percentage of essential write-backs for all benchmarks and nursery sizes never goes above 28% for any configuration, but benchmarks with higher essential percentages include eclipse, pmd, and xalan.

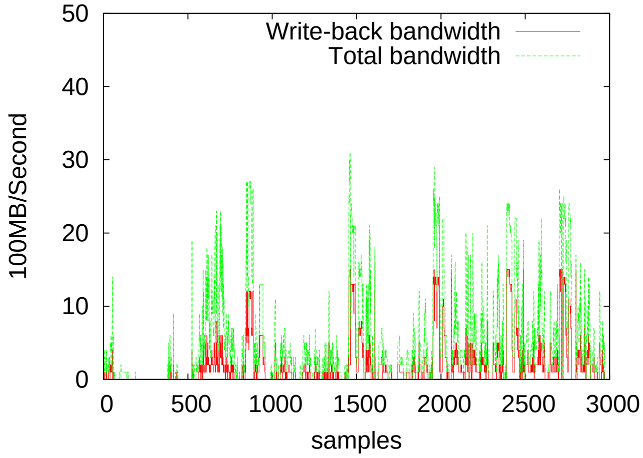
For nursery sizes 4MB and greater, the percentage of mature write-backs is fairly stable. As the nursery grows much larger than the last level cache, the fraction of the nursery that is cache-resident when our optimization is invoked grows smaller, reducing its effectiveness. Zhao et al. point out that Sun's HotSpot JVM is very sensitive to nursery size, and needs enormous nurseries (up to 2GB) to perform well when large heaps are used [77]. It is not clear *why* HotSpot requires such large nurseries, although we hypothesize that the problem stems from HotSpot's use of card-marking, which imposes a cost proportional to the heap size at each nursery collection. This makes each nursery collection expensive when the heap is large, encouraging larger nurseries and less frequent collections. In our own experiments with Jikes RVM, we find that 2M and 4M nurseries perform only slightly worse than all larger sizes. Our results are consistent with Blackburn et al. results on older hardware that show nursery sizes of 4M to 8M balance mutator locality with lower collection costs [14].

The percentage of mature useless write-backs increases at lower nursery sizes. Since the L2 cache is 4MB here, lowering the nursery size below 4MB fills more of the cache with long-lived mature data, and consequently a greater proportion of evictees are from the mature space. Very small nursery sizes reduce, but do not eliminate, the opportunities to eliminate useless write-backs, since the nursery remains cache resident. Small nursery sizes have better cache behavior, but the cost of more frequent collection dominates when the nursery is too small. The large number of useless nursery write-backs across a range of practical, moderate to large nursery sizes motivates our invalidation approach.

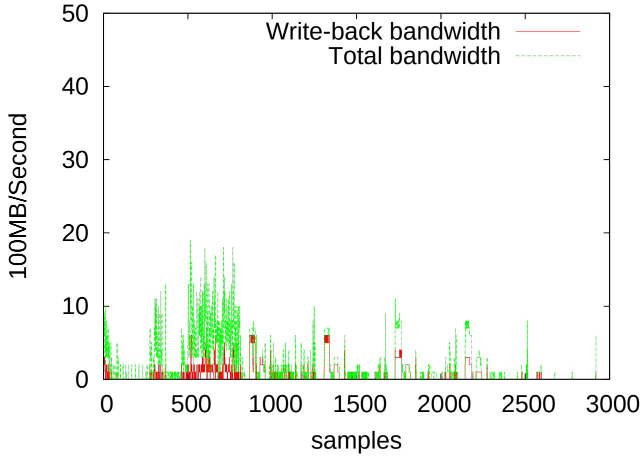
2.2 Measured Bandwidth

Whereas the previous section characterizes the semantics of write-backs, this section presents time series bandwidth measurements. Since bandwidth requirements are bursty, these measurements seek to characterize the bandwidth needs beyond averages across a benchmark run. We use performance counters to sample the number of full cache line write-backs and all full cache line front side bus (FSB) transactions every 10ms on the Core 2 Quad CPU, model Q6600, 2.40GHz, 2GB of memory, and 1.06GHz front side bus. We measure five iterations of each benchmark on HotSpot 1.6.0 and Jikes RVM 3.1.0. Figures 2.2 and 2.3 presents the total measured bandwidth and the write-back bandwidth for two representative benchmarks, jython and xalan. The jython benchmark has a lot of useless nursery write-backs. The xalan benchmark has fewer useless write-backs as shown in the previous subsection. Whereas xalan's total bandwidth is mostly due to reads as evidenced by the large gap between the upper and lower lines, the total bandwidth of jython is mostly generated by write operations since write-back bandwidth is almost half of total bandwidth. Furthermore, each write miss on this architecture first requires a fetch and we see this correlation in jython and other benchmarks (omitted due to space limitations).

Comparing Figures 2.2(a) and 2.2(b), and Figures 2.3(a) and 2.3(b) shows that HotSpot has much higher bandwidth needs than Jikes RVM. This result is ex-

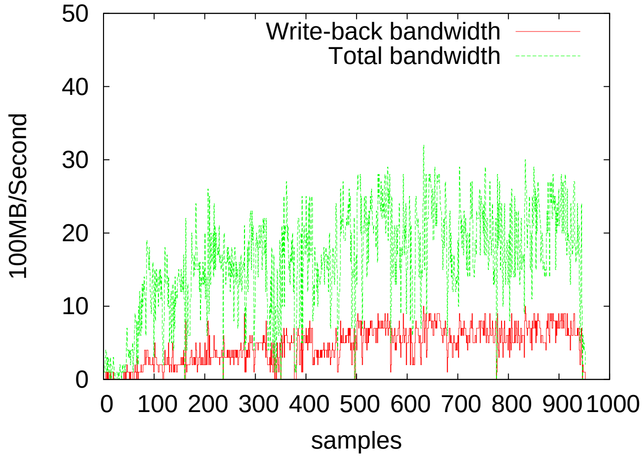


(a) Jython on HotSpot

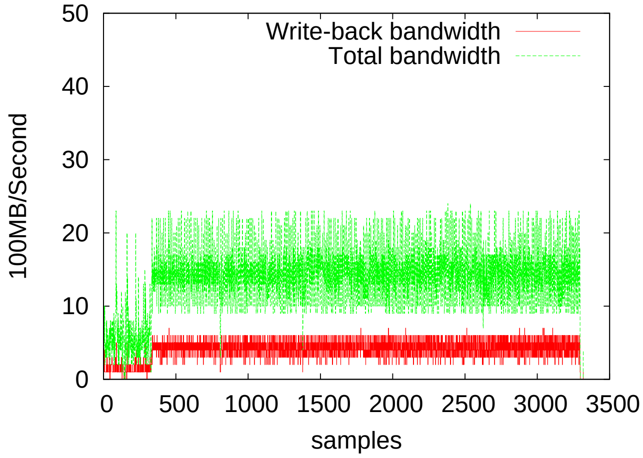


(b) Jython on Jikes RVM

Figure 2.2: Measured write-back and total full cache line FSB transactions for Jython on Sun HotSpot and Jikes RVM.



(a) Xalan on HotSpot



(b) Xalan on Jikes RVM

Figure 2.3: Measured write-back and total full cache line FSB transactions for Xalan on Sun HotSpot and Jikes RVM.

pected when the application runs faster since the application's work is condensed in time and thus could have higher bandwidth spikes, as shown above on the x-axis for xalan. However, bandwidth JVM trends hold even when JikesRVM executes slightly faster, such as in the benchmark jython. We hypothesize that HotSpot's large nursery sizes could be part of the cause of this excess bandwidth. We use Jikes RVM for our work, but the impact of our optimizations may likely be higher on HotSpot.

Chapter 3

Heap Data Compression

This section describes a limit study that examines the sources and types of memory inefficiencies in the heap for a set of Java benchmarks. We generalized and quantitatively compare many previously-proposed heap data compression techniques for the first time. Here we focus on array compression, which we found most fruitful, but our ISMM paper has complete results including object compression [63]. Our limit study shows that substantial memory reductions are possible: removing zero-bytes from arrays reduces the applications memory footprint by 41% for our benchmarks. When we combine many array compression techniques on the same data in new ways, we achieve an application heap reduction of 52% on average. First, we present some related work on previous compression techniques.

3.1 Related Work

High-level languages abstract memory management and object layout to improve programmer productivity, usability, and security, but abstraction usually costs. Here we describe related work on characterizing heap data and investigating specific compression techniques.

Modeling and characterization. Mitchell and Sevitsky categorize fields by the role they play in an object (header, pointer, null, primitive), and objects by the role they play in a data structure (head, array, entry, contained) [54]. They studied bloat, or spurious memory consumption, and found that a large amount of the heap is bloat. These measures together with scaling formulas predict the heap data reductions of manual program changes. Whereas Mitchell and Sevitsky focus on providing human heap understanding, we focus on heap compression that can be performed in the JVM.

Dieckmann and Hölzle study object lifetimes, size, type, and reference density for the SPECjvm98 benchmarks [28]. In addition to these measures, Blackburn et al. study time varying heap, allocation, and lifetime behaviors of the SPECjvm98, SPECjbb2000, and DaCapo benchmarks [13]. They show that DaCapo is significantly richer in code and data resource utilization than SPEC, which is why we use DaCapo here. While these studies provide general insights on heap memory composition, we measure specific limits of heap data compression.

Heap data compression. The following research implements specific heap data compression techniques. Appel and Gonçalves use generational garbage collection for deep-equal acyclic object sharing [6]. Objects are deep-equal if they have the same type and equivalent data, making sure reference data points to equivalent objects as well. Ananian and Rinard use offline profiling and an ahead-of-time compiler to implement a variety of techniques, including bit-width reduction [5]. Bit-width reduction compresses by storing only bits that are needed to hold particular data values, instead of the larger, standard bit-size for a type. Chen et al. use compacting garbage collection for zero-based object compression and speculative trailing zero array trimming [21]. For zero-based compression, Chen et al. elides zero bytes, storing a map, each bit indicating whether an original byte was zero and elided, or non-zero and stored. Zilles uses speculative narrow allocation for character array bit-width reduction [78]. Instead of using all 16 bits per character, Zilles recognized it is common for most character arrays to hold only English characters and thus be able to be reduced to 8 bits per element. All these techniques trade time for space, incurring time overheads to reduce space consumption in embedded systems.

Whereas the above research overcomes some real-world challenges of heap data compression, the others propose optimizations without evaluating full implementations. Shaham et al. hand-optimize benchmarks with object-level lifetime optimizations [66]. Marinov and O'Callahan hand-optimize benchmarks with deep-equal object sharing [53]. We explore the limits of compression techniques, and go a step further by empirically comparing a wide variety and combinations of techniques. Z-rays, our discontiguous array design, offer a much better building block for compression and future array optimization needs.

3.2 Heap Data Compressibility Analysis

Figure 3.1 shows our analysis steps for measuring the potential of heap compression. When gathering heap data for our limit study, we ignore fragmentation and only consider live objects in our analysis because we assume the garbage collector reclaims dead objects rather than compressing them.

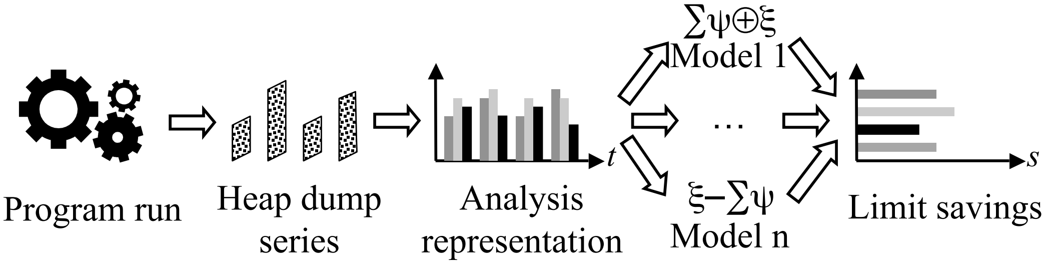


Figure 3.1: Heap data compressibility analysis.

Since a program's heap changes over time, its memory efficiency is also a function of time. A perfectly accurate heap analysis would compute savings on all live objects after every write and object allocation, but this analysis is prohibitively expensive. Instead, our analysis takes periodic heap snapshots during program execution ("Heap dump series" in Figure 3.1). It therefore over-approximates heap compression because, for example, a field value may be zero at every heap snapshot, but take on non-zero values between snapshots. We modify the garbage collector to print out a heap snapshot during live object traversal. In addition to its usual work, during a heap dump the garbage collector also prints object data (excluding bookkeeping information from the header) as it visits each live object on every collection.

Since heap snapshots require a lot of I/O, they take a lot of storage and time to generate. More heap snapshots yield more accurate compression measurements, but require more time and space. We empirically selected 25 as our target number of heap snapshots per execution of an application. We execute the benchmarks with two times their minimum heap size using a mark-sweep collector, and print around 25 heap snapshots at regularly-spaced intervals during normal full heap collections. Our benchmarks perform between four and three hundred garbage collections at this heap size. For those benchmarks with fewer than 25 collections, we force more frequent collections to obtain the desired number of heap snapshots. Because we use Jikes RVM for our experiments, we differentiate between JVM and application object allocations by adding a small amount of instrumentation (see [63] for details).

Given a series of heap snapshots, a post-processing step applies analytical models that compute potential compression opportunities. The post-processor iterates over the heap snapshot, entering each object instance's data into a large hash table ("Analysis representation" in Figure 3.1). We then apply a variety of compression models to compute potential compression opportunities. Each model calculates the memory savings from a particular heap compression technique ("Limit savings" in Figure 3.1). Section 3.3 describes and presents formulas for all considered techniques. We calculate potential memory savings for each unique class at a particular point in time, i.e., based on a particular heap snapshot taken during a garbage collection, thus including all live objects. For the snapshot, we count the number of object and application instances and bytes seen in order to calculate savings percentages.

Helper functions. Many of our savings models require helper functions. Function $\mathit{sizeof}(T)$ returns the size of a primitive type in bytes. Some compression techniques require a hash table at runtime, for example, to find equivalent objects. Their models subtract the size of the hash table from the raw savings. Function $\mathit{hashTableSize}(n, \mathit{entrySize})$ estimates the size of a hash table with n entries of size *entrySize* each. We assume a hash table with open addressing, since they have no memory overheads for boxes or pointer chains for overflowing elements. We also assume that $\frac{2}{3}$ of the hash table is occupied. This assumption is conservative. For example, the Java library writers use a load factor of $\frac{3}{4}$ before doubling their size, although they use chaining instead of open-addressing. The helper function works as follows, where *arrayHeaderSize* is 12 bytes and *keySize* is 4 bytes:

$$\begin{aligned}&\mathit{hashTableSize}(\mathit{numberOfEntries}, \mathit{entrySize}) = \\ &\quad \mathit{arrayHeaderSize} + \left\lceil \tfrac{3}{2} \cdot \mathit{numberOfEntries} \cdot (\mathit{entrySize} + \mathit{keySize}) \right\rceil\end{aligned}$$

3.3 Memory Compression Models

A compression model is a formula that computes how many bytes of heap data that technique can save at an instance in time. Table 3.1 overviews all the models

Compression technique	Granularity	Reference
Lempel-Ziv compression	Whole Heap	
Trailing zero array trimming	Instance	[21]
Bit-width reduction	Instance	[5, 69, 78]
Zero-based array compression	Instance	[21]
Strictly-equal array sharing	Type	
Deep-equal array sharing	Type	[6, 53]
Value set indirection	Type	[26, 71]
Value set caching	Type	

Table 3.1: Compression techniques modeled.

considered in this thesis, listing the granularity they are performed at (per array instance or type), and listing references to prior work for that compression technique. To obtain the total savings of a model, we compute the savings for each instance and then sum them up over all types.

3.3.1 Holistic heap data size and information content

Models in this section quantify the size of all the data in the heap. Because the heap contains redundancies and bookkeeping data, the actual information content is smaller than its conventional representation.

Total heap size

We measure the total heap size by summing all objects, fields, object headers, and array elements in the heap, assuming a conventional representation, and excluding fragmentation, static objects, and the stack. The below models compute savings from this baseline.

Lempel-Ziv compression

We first consider the memory savings achieved by simply zipping the contents of all heap objects to illustrate the potential for heap reduction. The size given by “`bzip2`” is a rough estimate of the true “information content” of the heap. We expect this savings to be larger than for any of the more realistic models below. Like the other models, Lempel-Ziv compression is non-lossy, in other words, the original data can be fully recovered by decompression. Unlike the data representations for most of the other models, Lempel-Ziv compressed data does not permit random access, let alone in-place update. To compute this model as accurately as possible, we perform online compression on the actual heap dump in the JVM at garbage collection time. We apply Lempel-Ziv compression and report the compressed size as a percentage of the uncompressed size [63]. Total heap compression is fairly consistent, reducing the heap between 73 and 83%, on average for our benchmarks 75%. For only application objects we see larger compression opportunity, up to 99% for fop and pseudojbb, on average 90%. Though this shows that there are excessive amounts of redundancy in the heap that can be reduced, we do not expect this much compression in practice.

3.3.2 Array Instance Compression

This section presents compression techniques that operate on individual array instances.

Trailing zero array trimming

Programs often over-provision the capacity of arrays used as buffers, leading to unused trailing zeros [21]. These can be trimmed, provided that the trimmed array remembers the nominal and true length. Assuming it takes an additional 4 bytes to store both lengths, the savings for array type $T[\,]$ are:

$$\sum_{a \in T[\,]} \big(\mathit{trailingZeros}(a) \cdot \mathit{sizeof}(T) - 4\big)$$

Array bit-width reduction

If all elements of an array instance have small values, then they can all be represented with a smaller bit-width [69]. Bit-width reduction saves memory by allotting fewer bytes for each array element than the type requires. Array bit-width reduction sums up savings per instance to compute the total savings.

In general, if all values need at most B bits, compression can optimistically represent an array of type $T[\,]$ as a B-bit array. The total byte savings is:

$$\sum_{a \in T[\,] \wedge onlyUsesBits(a,B)} \left\lfloor \frac{8 \cdot sizeof(T) - B}{8} \cdot a.length \right\rfloor$$

Bit-width reduction works for many types of arrays, such as int, short, or long. In the past, boolean and char arrays have been a focus because compression can yield large savings due to Java's language specifications [78]. By default, though booleans only need one bit of memory, Java virtual machines use a byte to represent a boolean because for bookkeeping it is easier to have memory byte-aligned. Compressing each boolean array could save $(\frac{7}{8})^{ths}$ of the space taken by each array. Similarly, Java represents characters using a 16-bit encoding for unicode, but English-language applications tend to use mostly characters that require only the lower 8 bits. The accordion arrays bit-width compression optimization represents each array that consists entirely of 8-bit characters using a byte array [78], thus cutting the size of each char array in half.

Zero-based array compression

Zero-based array compression reduces array size by removing bytes that are zero. We assume an implementation that uses a per-array bit-map to indicate which bytes in the original array are entirely zero [21]. The compressed array representation consists of the header, the bit map, and the values of all non-zero bytes. The size of the bit map is the number of non-header bytes in the original array. The bit-map for array a occupies $\lceil totalBytes(a)/8 \rceil$ bytes. The savings for all arrays in the heap are therefore:

$$\sum_{a \in Arrays} \left(zeroBytes(a) - \left\lceil \frac{totalBytes(a)}{8} \right\rceil \right)$$

Note that this compression scheme can be applied to both primitive and reference arrays. We compute memory savings per instance, and then add them up for each array type.

3.3.3 Array Type Compression

This section presents compression techniques that operate across all arrays of a particular type.

Strictly and deep-equal array sharing

Two arrays are *strictly-equal* if they have the same type, length, and data. Equality is strict because even pointer fields must be identical [53]. When arrays are strictly equal, they can share all their memory. The JVM may allocate only one instance and then point all references of strictly-equal arrays to the same instance. In principle, two arrays can not be shared if they are used for pointer comparison or as an identity hash code in the future. In addition, the period of time for sharing may be limited if the program modifies a strictly-equal array later. Our analysis ignores these cases for the purpose of this limit study, but our z-ray implementation handles this case correctly.

Deep-equal compression offers additional space saving opportunities. If two reference arrays have internal pointers that point to different objects (and are thus not strictly-equal) but the objects to which they point are equivalent, those reference arrays can be shared via *deep equality* [6, 53]. Every strictly-equal array pair is also deep-equal, thus there are more deep-equal array pairs than strictly-equal ones.

Since different length arrays have different sizes, we iterate over all arrays to add up their sizes before compression, construct the hash table, and then iterate over all D distinct arrays to find the unique size. The model must also subtract the memory used for the hash table itself. The resulting savings model for array type $T[\,]$ for both strictly and deep-equal is:

$$\sum_{a \in T[\,] \wedge a \notin D} sizeof(a) - hashTableSize(D, pointerSize)$$

Since fewer distinct arrays are deep-equal as compared with strict equal because the former exposes more sharing opportunities, the D in the equation is smaller.

Array value set indirection

Array value set indirection saves memory by holding a “dictionary” of values for elements separately from array instances, enabling instance elements to hold a smaller index into the dictionary. In our limit study, compression requires that all elements of all arrays of a given type are drawn from a small set of distinct values, so the dictionary is for the whole type. Compression is performed per array instance, and replaces each element with a small index into the dictionary that stores the actual values. For example, if all instances of an array type $T[\,]$ contain at most $K < 256$ different values, array elements can store an 8-bit index into a K-entry table of values of type T. The memory savings are:

$$\sum_{a \in T[]} a.length \cdot \big(sizeof(T) - 1\big) - arrayHeaderSize - K \cdot sizeof(T)$$

This optimization makes no assumptions about the element type. It applies equally well for int, float, pointer, etc. Where bit-width reduction requires all field values to be small, value set indirection only makes requirements on the number of field values. This model does reduce element size, but because it is more generally applicable than array bit-width reduction, it incurs the overhead of storing the dictionary. In general, the dictionary of values could be taken from all array elements of a particular type (possibly selecting the 256 most common values), and only the instances of that type that have all of their elements contained in the dictionary could be compressed.

Array value set caching

Array value set indirection can be generalized to the case where there are more values than fit in the dictionary. Caching puts the 255 most common values in a dictionary, like indirection, but also reserves one dictionary index (of 256) to indicate additional, aberrant values. Value set caching stores the rarer values into a secondary hash table. We use a combination of the original array's object ID and the index of the array element for the secondary hash table's key. An array access `a[i]` in this case is as follows:

```
if a[i] == aberrant_indicator:
  return secondary_hash.get(a, i)
else:
  return dictionary[a[i]]
```

Let A be the total number of aberrant array elements in all arrays of type $T[\,]$. Then the savings are:

$$\begin{aligned}&\sum_{a \in T[\,]} a.length \cdot \big(sizeof(T) - 1\big) - arrayHeaderSize - K \cdot sizeof(T) \\ &- hashTableSize'\big(A, sizeof(T)\big)\end{aligned}$$

The *hashTableSize'* function assumes that keys are 8 bytes, because the key represents both an array and an index.

3.3.4 Hybrids

Hybrids combine multiple compression techniques to obtain more savings than one technique alone.

Maximal hybrid

The maximal hybrid chooses the compression technique that saves the maximum amount of memory for each piece of data. For arrays, the maximal hybrid starts by choosing the maximal array instance compression techniques:

$$maxArrayISavings(T[]) = \max_{o \in ArrayIOpts} \sum_{i \in T[]} savings(T[], i, o)$$

The set of array instance optimizations, *ArrayIOpts*, contains trailing zero trimming, bit-width reduction, and zero-based compression. Next, the maximal hybrid picks the best optimizations for each array type $T[]$, picking the maximum savings between the best instance technique and the savings that could be achieved from the whole-type techniques:

$$maxArrayTSavings = \sum_{T[] \in Arrays} \max \left\{ maxArrayISavings(T[]), \max_{o \in ArrayTOpts} savings(T[], o) \right\}$$

The set of array type optimizations *ArrayTOpts* contains strictly-equal array sharing, value set indirection, and value set caching[1]. The overall maximum hybrid savings is therefore the sum of the best technique over all types of arrays.

Combined hybrid

In some cases, after applying an optimization o_1 to a piece of data, it is possible to apply o_2 as well on the same data to obtain additional savings. For example, let o_1= "trailing zero array trimming" and o_2="array bit-width reduction", then we may achieve more savings with the hybrid $o_1 \circ o_2$ than either o_1 or o_2.

We calculate combined-hybrid heap compression by applying multiple models in sequence. For the maximum potential savings per array instance, we apply the optimizations from *ArrayIOpts* in the following order: (1) trailing zero trimming, (2) bit-width reduction, and (3) zero-based compression. Throughout these calculations, we keep track of changes to the array length, array size, element size, and number of zero entries to feed into later optimizations. We follow the instance optimizations by type optimizations in *ArrayTOpts* to explore further compression. Even if instance optimizations have been performed to reduce the array footprint, strictly-equal array sharing, array value set indirection, and caching could realize further savings. However, we do not need to recalculate the array sharing hash table, as instance optimizations only elide zeros and do not change element values. After we calculate combined savings for each array type, we add them to compute the total *combinedArrayTSavings*.

---

[1] Due to implementation limitations, we exclude deep-equal array sharing from this hybrid calculation.

3.4 Limit Study Results

This section evaluates and compares the compression models.

3.4.1 Methodology

We added heap data compressibility analysis to Jikes RVM [4] version 2.9.1. We used the “FastAdaptiveMarkSweep” configuration, which optimizes the boot image (“Fast”) and uses a mark-sweep GC. We disabled the optimizing compiler during the application run to reduce compiler objects in the heap. Since Jikes RVM itself is written in Java, it allocates JVM objects in the Java heap alongside application objects; we show both total and application-only results. Our benchmark suite consists of the DaCapo benchmarks [13] version “dacapo-2006-10-MR1”, and of pseudojbb, a variant of SPECjbb2000 (see `www.spec.org/osg/jbb2000/`). We used Ubuntu Linux 2.6.20.3.

Benchmark	GC	# Types		Size [KB]			Size [Instances]		
		Cls	Arr	Total	Arr	App	Total	Arr	App
antlr	14	529	69	49,811	70%	5%	785,561	33%	0.5%
bloat	122	593	79	56,724	67%	16%	967,360	32%	17%
chart	23	675	85	56,616	68%	8%	970,070	33%	3%
eclipse	44	1,136	175	87,799	68%	25%	1,534,580	35%	18%
fop	22	784	73	52,184	70%	4%	830,111	34%	0.6%
hsqldb	13	538	78	201,375	43%	76%	7,231,418	21%	89%
jython	218	892	78	63,706	67%	9%	1,067,359	33%	6%
luindex	11	533	70	50,185	70%	7%	794,633	33%	2%
lusearch	26	536	73	70,994	78%	34%	850,828	33%	8%
pmd	71	644	72	59,220	67%	9%	992,510	34%	6%
xalan	125	711	88	71,148	76%	27%	940,201	36%	9%
pseudojbb	18	495	72	74,180	73%	35%	1062,901	36%	25%

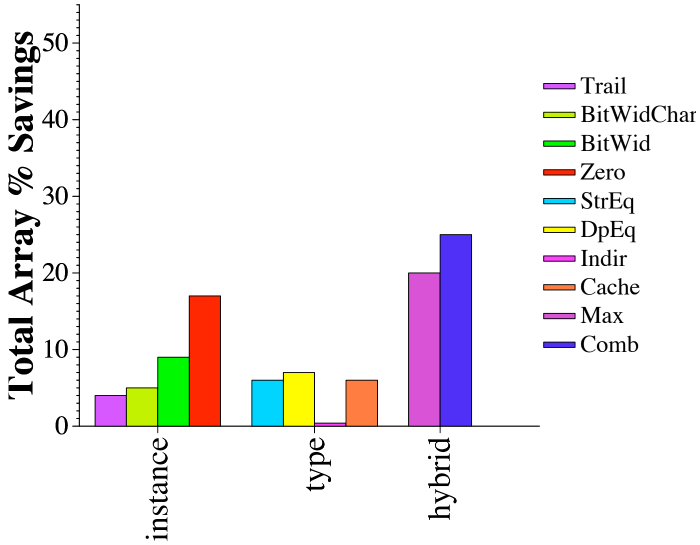
Table 3.2: Benchmark and heap dump characterization.

We pick one representative heap dump mid-way through the gathered snapshots (around 25) to calculate the majority of the space savings. Section 3.4.3 validates the generalizability of one heap snapshot for all benchmarks, comparing savings over many snapshots. Table 3.2 characterizes our benchmark suite, including the GC number mid-way through the run, the number of class and array types represented in the measured heap dump, and the size of the measured heap dump. For all benchmarks, arrays occupy more bytes but have fewer instances than classes. The application occupies between 4% and 76% of the amount of bytes occupied by Jikes RVM. All tables and figures represent total memory savings as a percentage of total KB, and represent application memory savings as a percentage of application KB.

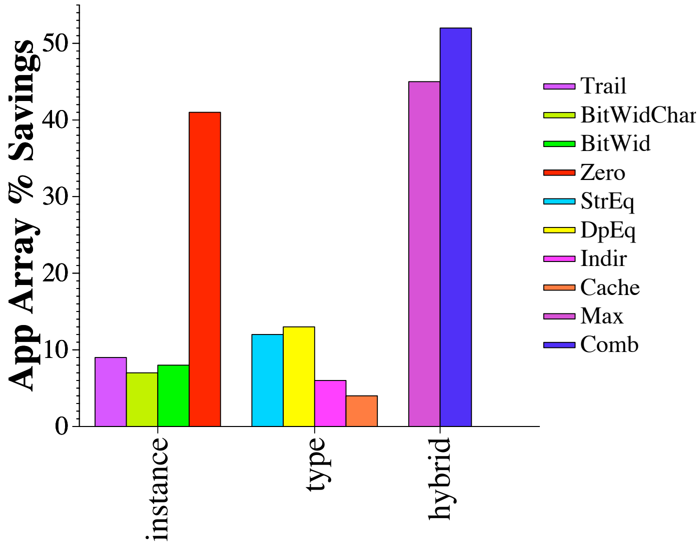
3.4.2 Array Compression

Figure 3.2 shows bar graphs of the average savings over all benchmarks per compression model. Figure 3.2(a) shows the total heap while Figure 3.2(b) shows the application heap. Comparing the two graphs, we see that on average we can save more space with our techniques in the application heap. The individual technique that can save the most heap space is zero-byte compression by far, saving 17 and 41% of the total and application heaps. Bit-width reduction, trimming trailing zeros, and array equality compression are fairly effective at reducing the heap as well, saving 8, 9 and 13% of the application heap respectively. The graphs also show that our hybrid techniques achieve the highest savings by performing many compression techniques on each piece of data, with maximal hybrid saving 45% and combined saving 52% of the application heap.

Tables 3.3 and 3.4 shows memory savings from the array compression techniques, which are overall more effective than object techniques, for total heap and application, respectively. Many benchmarks have a significant amount of arrays allocated with padding that can benefit from trailing zero trimming. *Xalan*, with a



(a) Total heap average



(b) Application heap average

Figure 3.2: Average Compression Savings

Benchmark	Total									
	Trl	Bitwidth		Zero	Equal		Value set		Hybrid	
	Zro	Ch	GC	Compr	Str	Deep	Indr	Cch	Max	Comb
antlr	3	4	8	14	4	5	0.3	5	16	20
bloat	3	5	8	14	8	8	0.5	6	17	21
chart	3	4	8	13	4	5	0.3	5	16	19
eclipse	2	6	9	14	4	4	0.2	6	17	22
fop	6	4	7	16	4	5	0.5	5	18	22
hsqldb	1	0.9	2	12	3	3	0.1	1	12	13
jython	2	5	8	13	6	7	0.4	6	16	21
luindex	5	4	8	15	5	5	0.3	5	17	21
lusearch	2	5	7	32	4	4	0.2	5	34	39
pmd	3	5	9	14	7	7	0.5	6	16	22
xalan	18	4	10	29	21	22	2	7	33	38
pseudojbb	3	14	18	20	4	5	0.4	14	24	37
average	4	5	9	17	6	7	0.4	6	20	25

Table 3.3: Percent total memory savings from array compression including hybrids.

large part of its heap being arrays, can save 18% by trimming zeros. For just application arrays, this model helps most for *chart*, *luindex*, and *xalan*. Our benchmarks spend very little space on boolean arrays, hence we do not show a separate column for bit-width compressing them; the numbers were all zero. However, many character arrays benefit from bit-width reduction – up to 14% with *pseudojbb*. *Eclipse* and *pseudojbb* application character arrays are prime candidates for savings. Although application savings vary per benchmark, similar to Zilles' results which were computed with a different methodology, we see up to 32% compression possibility [78]. Interestingly, arrays of types other than character can also benefit significantly from bitwidth reduction, with savings between 2 and 18%. Value set indirection helps little for arrays, because it is too strict: at least some array types have more values than the allotted dictionary. However, using the dictionary as a cache and placing aberrant values in a secondary hash increases the opportunities significantly, and so value set caching saves more memory. In particular, *pseudojbb* can compress the

Benchmark	Application									
	Trl	Bitwidth		Zero	Equal		Value set		Hybrid	
	Zro	Ch	GC	Compr	Str	Deep	Indr	Cch	Max	Comb
antlr	3	2	2	48	12	12	2	0	48	50
bloat	3	9	10	25	6	7	10	0	28	38
chart	11	2	4	41	2	2	0	2	43	46
eclipse	0.9	13	13	25	2	2	0.1	13	28	41
fop	0.1	0.1	0.4	57	14	15	0.5	0	58	60
hsqldb	0.4	0	0.3	11	3	3	0	0.3	12	12
jython	2	3	4	28	8	10	0.2	5	30	35
luindex	25	0.9	1	58	27	27	0.9	0.2	61	66
lusearch	0.9	5	6	72	4	5	6	0	73	80
pmd	8	5	6	33	9	9	5	1	34	37
xalan	49	6	17	66	58	59	13	24	78	84
pseudojbb	4	32	34	33	4	4	32	0	42	74
average	9	7	8	41	12	13	6	4	45	52

Table 3.4: Percent application memory savings from array compression including hybrids.

heap by 14% with value-set caching.

As expected, deep-equal sharing results in more savings than strictly-equal sharing. Because most arrays are primitive, strict equality does very well for arrays, and deep-equality saves only a percent more on average. *Xalan* in particular benefits greatly from deep-equal array sharing, saving 22% of the total heap and 59% of the application heap. Zero-based compression gets some good savings for the total heap, ranging from 12 to 32% for arrays. Application-specific zero-based compression savings greatly depend on the benchmark. Overall zero-based compression achieves the highest individual savings: on average 17% for the whole heap and for just the application 41%. Zero compression comes at the cost of having to decompress individual object instances before use.

The savings from the maximal hybrid exceed the savings of any individual optimization, because it picks the best compression technique for each individual piece of data. The savings from the combined hybrid exceed those from the maximal hybrid, because each piece of data may be optimized by multiple techniques. Maximal compression added an additional 4% savings over our best individual technique (zero-compression), whereas combined compression on the same data added 11% over the best individual technique. Using models presented in this paper, we see that the combined hybrid for arrays is able to compress the total heap by up to 39%, and the application heap by up to 84%, on average 25 and 52%. In general, application numbers vary more widely than total numbers, showing that the JVM is fairly consistent. Overall, zero-based object compression for arrays yields the largest amount of savings, but value set compression, bit-width reduction, and equal array sharing are all competitive and contribute reasonable heap savings as well.

3.4.3 Compressibility over time

Most of the results in this section are for one mid-run heap dump only. We investigated whether one heap dump can be representative for the entire run by plotting a compressibility time series for all benchmarks in Figures 3.3, 3.4, 3.5, and 3.6[2]. We show series graphs for both total heap and application heap with the x-axis as time, and the y-axis as the percent memory savings. The lines are mostly horizontal, validating that compressibility changes little from heap dump to heap dump. More variation is seen at startup and shutdown as expected, but the middle of the run is fairly stable. This shows our per collection savings for arrays at a middle heap dump should be representative. We see more variation across the run for application data compression, especially with *bloat* and *pmd*. Perhaps these benchmarks have

---

[2] These graphs include both object and array compression

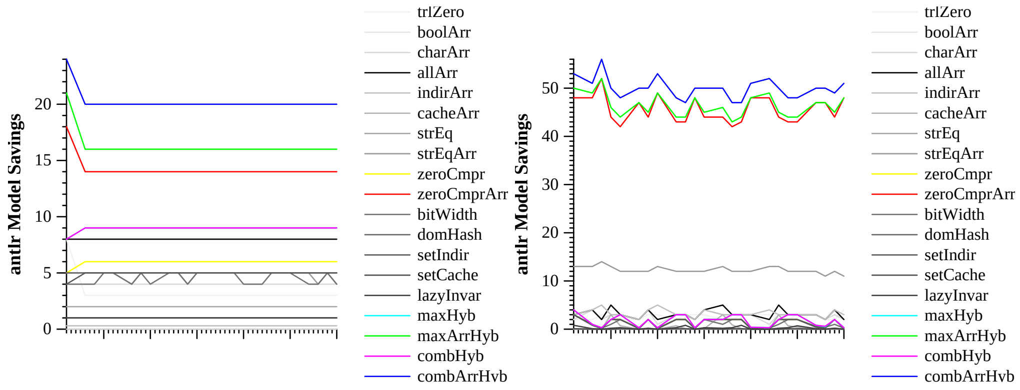
a heap that changes over time as the code goes through phases. Looking at *fop's* application graph, for example, we do not know if the number of objects in the heap is decreasing or if the number of compressible objects is decreasing. When implementing compression techniques in a JVM, we will need to take care to recover in case compression is no longer possible.

In a separate run with one benchmark, *fop*, we forced frequent heap dumps every 512KB of allocation, collecting 148 heap dumps. We calculated model savings over all heap dumps and plotted a series graph, as above. We found that it had a very similar shape to the series graph plotting only 20 heap dumps. We believe this shows that there is little bias in when we gather our heap snapshots during the program run.

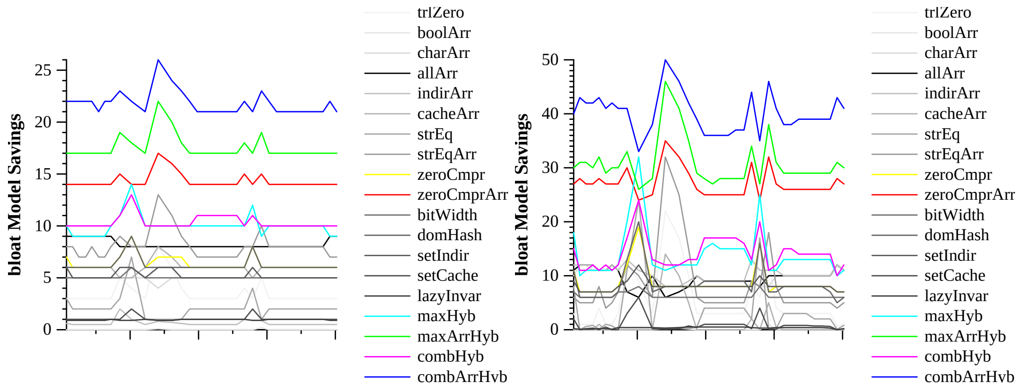
3.5 Compression Conclusion

For our benchmarks, 43 to 78% of the heap is taken up by arrays, and our models show the greatest potential compression with arrays (versus objects) [63]. Removing bytes that are zero is particularly effective, saving on average 41% of the application heap and up to 72%. Although previous researchers have analyzed many compression techniques, we are the first to apply many models successively to each piece of data in the heap. Our maximal array hybrid compression yielded a 45% reduction in the application's heap size on average while our superior combined hybrid yielded 52%. Our hybrid analysis shows great potential to reduce the heap bloat causing Java memory inefficiency. Overall, our limit study comparing various heap data compression techniques and novel hybrids shows that there is a lot of bloat and redundancy in arrays that can be alleviated. Reducing space usage in the heap can make memory more efficient, enticing more constrained and varied applications and systems to use high-level languages. Programmers could then obtain time and space efficiency in combination with faster development time and fewer errors.

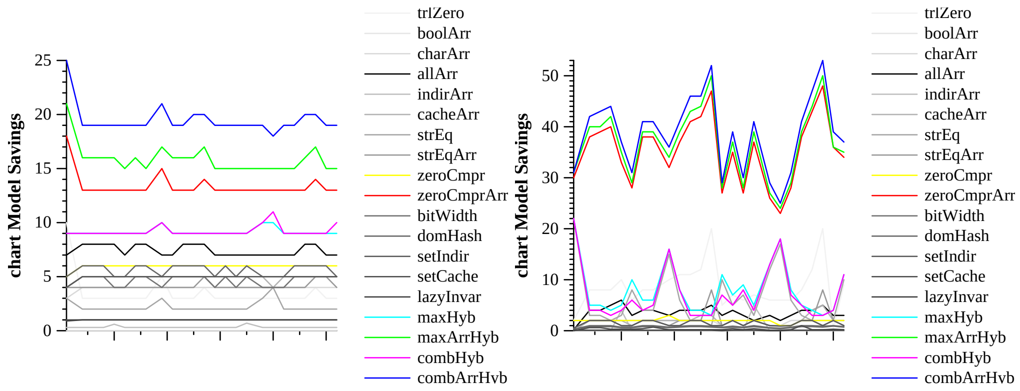
Although the compression techniques we evaluated cannot reach the 90% average compression that bzip can achieve, they can achieve over half of that savings by compressing arrays, while being able to access and update individual elements. However, there are many real-world implementation challenges for compression. For example, relying on offline profiling and static analysis reduces applicability to languages such as Java with dynamic class loading, reflection, and native code, and limits opportunity for dynamic compression. Java's abstract memory model allows for more flexible compression schemes that adapt to application phases. Some previous techniques target only specialized domains, such as interpreters, English-language characters, or acyclic data. Speculative optimizations require a back-out mechanism when compressed data properties are violated and this mechanism must be thread-safe. These challenges and runtime overheads expose space-time tradeoffs that are particular to applications running in a managed language setting. However, our limit study of techniques are a start in the direction of determining the most effective compression schemes that could be performed online per data item with low overhead. We explore some of the space-time tradeoff of the particular dynamic compression techniques we studied in our discontiguous z-ray work, and leave a more detailed analysis to future work.



(a) Total *antlr* (b) Application *antlr*

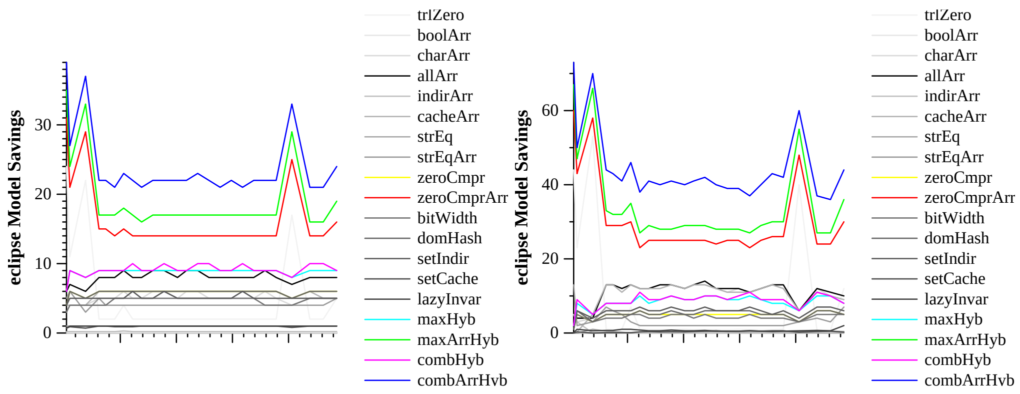


(c) Total *bloat* (d) Application *bloat*

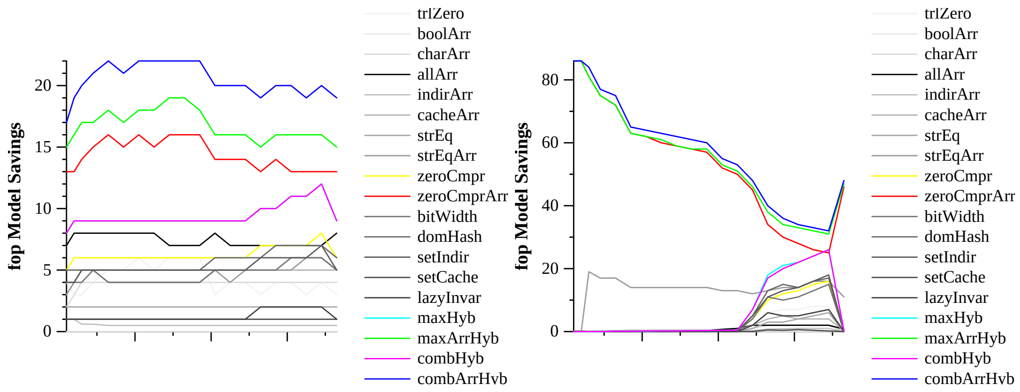


(e) Total *chart* (f) Application *chart*

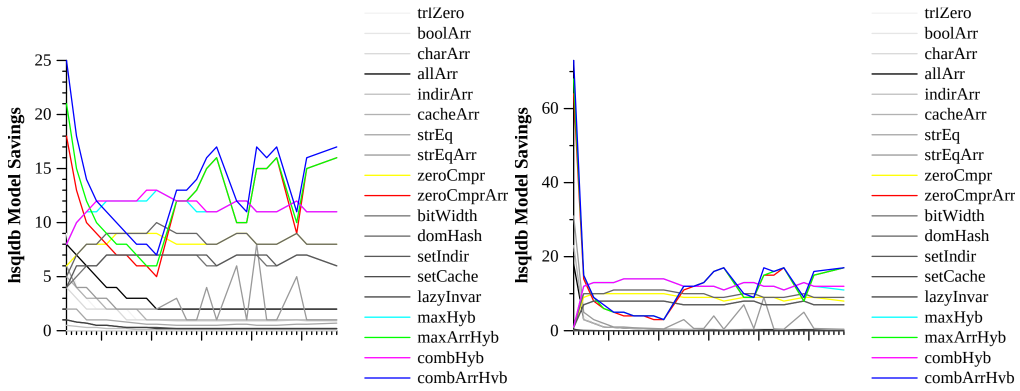
Figure 3.3: Compressibility over time for antlr, bloat, chart



(a) Total *eclipse* (b) Application *eclipse*

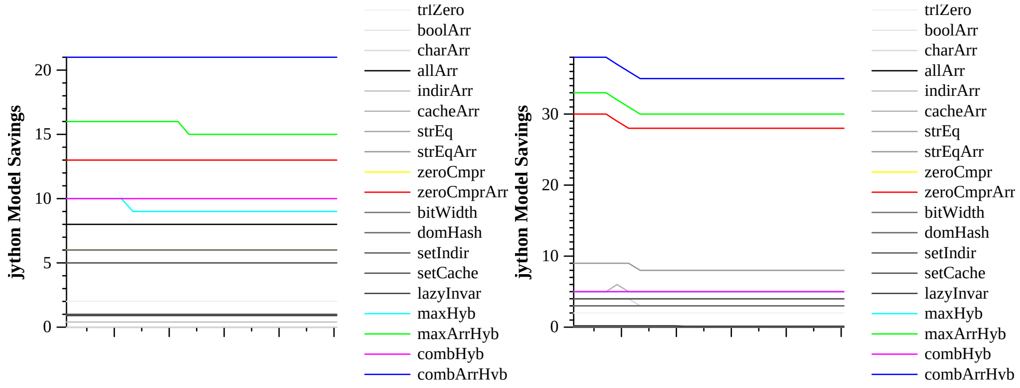


(c) Total *fop* (d) Application *fop*

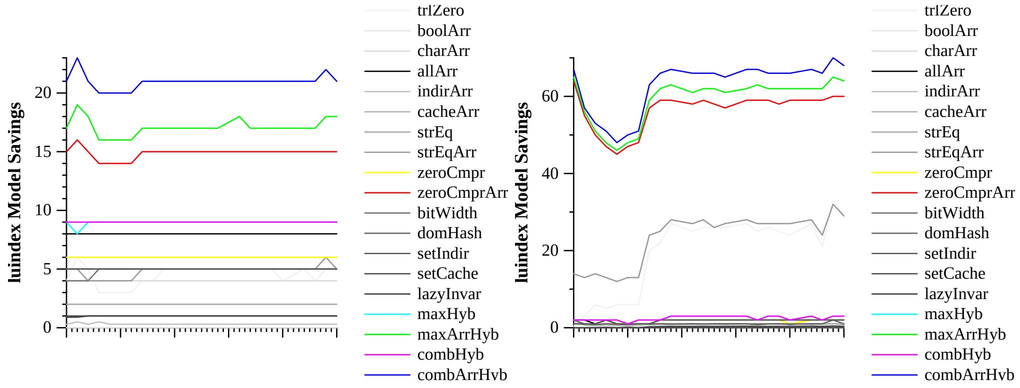


(e) Total *hsqldb* (f) Application *hsqldb*

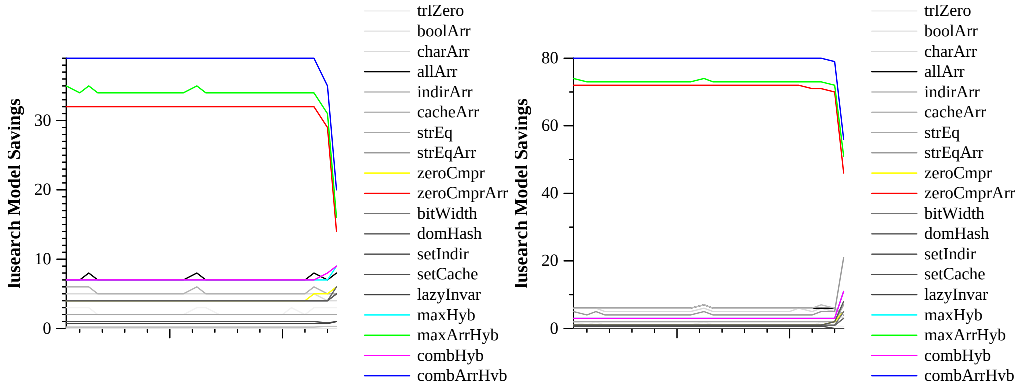
Figure 3.4: Compressibility over time for eclipse, fop, hsqldb



(a) Total *jython* (b) Application *jython*

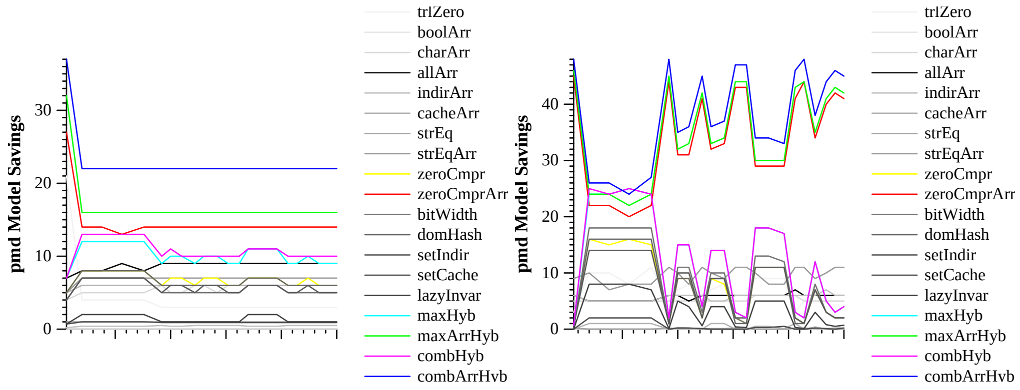


(c) Total *luindex* (d) Application *luindex*

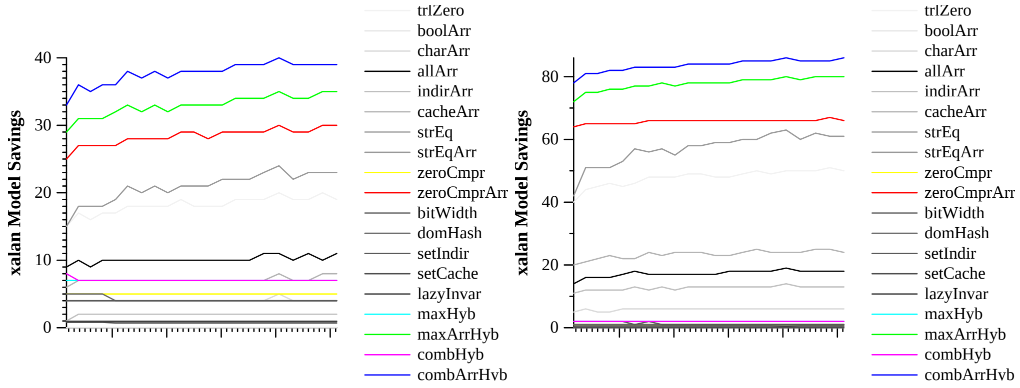


(e) Total *lusearch* (f) Application *lusearch*

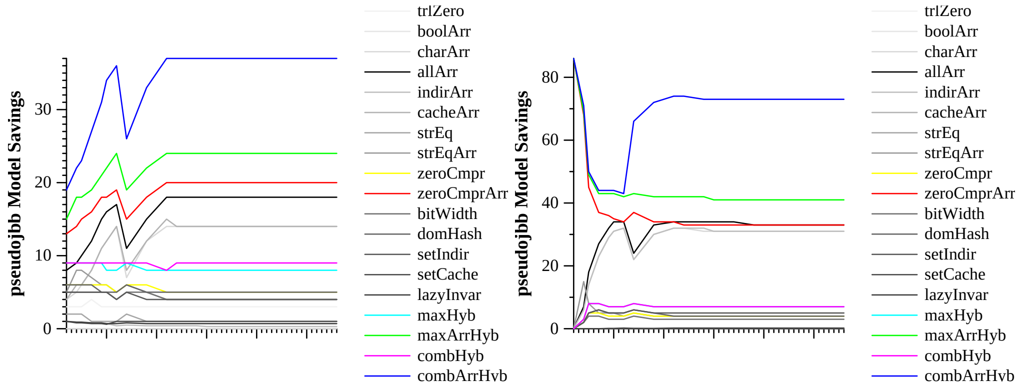
Figure 3.5: Compressibility over time for jython, luindex, lusearch



(a) Total *pmd*
(b) Application *pmd*



(c) Total *xalan*
(d) Application *xalan*



(e) Total *pseudojbb*
(f) Application *pseudojbb*

Figure 3.6: Compressibility over time for pmd, xalan, pseudojbb

Chapter 4

Z-rays: Efficient Discontiguous Arrays

The limit study in the previous chapter motivates optimizing arrays and their memory layout. Since large contiguous arrays cause many problems for memory management, such as fragmentation and large mutator pauses for memory management, researchers have proposed discontiguous array implementations [7, 21]. In the space-time tradeoff, discontiguous arrays sacrifice time for better space usage, and are more amenable to compression opportunities. Previous designs have optimized for space or real-time predictability, but incur large overheads on all array accesses, making discontiguous arrays a bad choice for widespread JVM incorporation. We first detail previous related work in this area. Then this section describes how we efficiently divide arrays up using our flexible z-ray design [62]. We describe first a naive design with a spine and arraylets. We then present our optimized z-ray design of discontiguous arrays and our five optimizations. We show that these optimizations yield similar space benefits as prior work, but with a fraction of their overhead.

4.1 Related Work

This section surveys work on implementations of discontiguous arrays, and describes work on optimizing read and write barriers, which are necessary and key to performance for discontiguous layouts.

Implementing discontiguous arrays. File systems pioneered the idea of pairing a contiguous abstraction (the file) with a discontiguous implementation [58]. Unix performs this organization on indexed files, which use indirection pointers to fixed size blocks for laying out large files on disk. Indexed files optimize for fast access to small files, and limit disk fragmentation due to large files. Furthermore, they provide constant access time for random and contiguous access patterns. Researchers use discontiguous arrays for similar purposes: improving predictability of reclamation and reducing space consumption for embedded systems.

Siebert's tree representation for arrays limits fragmentation in a non-moving garbage collector for a real-time virtual machine [3, 67]. Both Siebert's and our work break arrays into parts, but Siebert requires a loop for each array access, whereas we require at most one indirection.

Even though Cheng and Blelloch represent arrays contiguously, they synchronize and copy them one chunk at a time [24]. This design gives some of the benefits of discontiguous arrays (bounded pause times), but not all benefits (no fragmentation control or compression).

Discontiguous arrays provide a foundation for achieving real-time guarantees in the Metronome garbage collector [7, 8, 9]. Metronome uses a two-level layout, where a spine contains indirection pointers to fixed-size arraylets and inlined remainder elements. The authors state that Metronome arraylets are "not yet highly optimized" [8]. Metronome is used in IBM's WebSphere Real Time product [40] to quantize the garbage collector's work to meet real-time deadlines. Our performance optimizations are immediately and directly applicable to their system. Similar to the Metronome collector, Fiji VM [31, 57] also uses arraylets to meet real-time system demands, but require immutable spines to achieve space bounds. Because their layout still requires a lot of indirection, their implementation has high throughput overhead.

The use of discontiguous arrays in many production Java virtual machines establishes that arraylets are required in real-time Java systems to bound pause-times and fragmentation [3, 31, 40]. Applications that use these JVMs include control systems and high frequency stock trading. To provide real-time guarantees, these VMs sacrifice throughput. Z-rays provide the same benefits, but greatly reduce the sacrifice.

Chen et al. use discontiguous arrays for compression in embedded systems, independently from Metronome developing a spine-with-arraylets design [21]. If the system exhausts memory, their collector compresses arraylets into non-uniform sizes by eliding zero bytes and storing a separate bit-map to indicate the elided bytes. They also perform lazy allocation of arraylets. In contrast to our work, their implementation does not support multi-threading, and is not optimized for efficiency. They require object handles, which introduce space overhead as well as time overhead due to the indirection on every object access.

Read and write barriers. A key element of our design is efficient read and write barriers. Read and write barriers are actions performed upon every load or store. Hosking et al. were the first to empirically compare the performance of write barriers [36]. Optimizations and hardware features such as instruction level parallelism and out-of-order processors have reduced barrier overheads over the years [15, 16, 32]. If needed, special hardware can further reduce their overheads [25, 35]. We borrow Blackburn and McKinley's fast path barrier inlining optimization and Blackburn and Hosking's evaluation methodology. Section 4.3 discusses the potential added performance benefit of compiler optimizations such as strip-mining in barriers. We exploit recent progress in barrier optimization to make z-rays efficient.

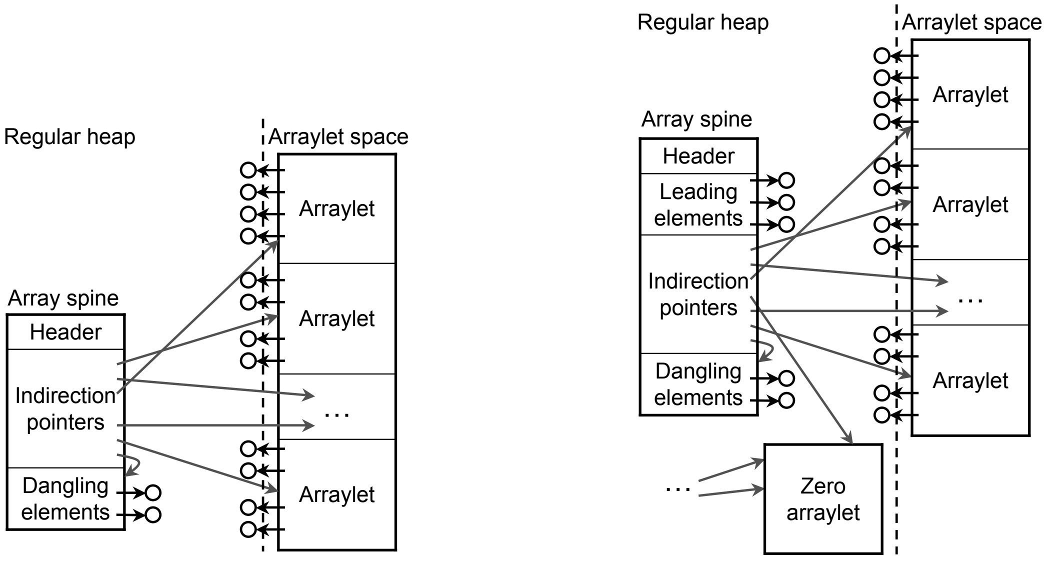
To summarize, discontiguous array representations have proven useful in embedded systems, but previously have been too costly for commodity use. Our contribution is to demonstrate that discontiguous arrays can be very efficient, as well as providing opportunities for compression and real-time quantization.

4.2 Z-rays

This section first describes a basic discontiguous array design with a spine and arraylets. The basic design heavily uses indirection and performs poorly, but it does address fragmentation, responsiveness, and space efficiency [7, 21]. Next, this section presents the z-ray memory management strategy and the five z-ray optimizations.

4.2.1 Simple Discontiguous Arrays Using Arraylets

Similar to previous work, we divide each array into exactly one *spine* and zero or more fixed-size *arraylets*, as shown in Figure 4.1(a). The spine has three parts. (1) It encapsulates the object's identity through its header, including the array's type, the length, its collector state, lock, and dispatch table. (2) It includes indirection pointers to arraylets which store actual elements of the array. (3) It may include inlined data elements. Spines are variable-sized, depending on the number of arraylet indirection pointers and the number of inlined data elements. Arraylets themselves have no header, contain only data elements, and are fixed-sized. Because arrays may not fit into an exact number of arraylets, there may, in general, be a *remainder*. Similar to Metronome [7], we *inline* the remainder into the spine directly after indirection pointers (see Figure 4.1(a)), which avoids managing variable-sized arraylets or wasting any arraylet space. We include an indirection pointer to the remainder



(a) Basic Discontiguous Arrays (b) Z-rays

Figure 4.1: Discontiguous reference arrays divided into a spine pointing to arraylets for prior work and optimized Z-rays.

in the spine, which ensures elements are uniformly accessed via one level of indirection, as in Metronome. We found that the remainder indirection is cheaper than adding special case code to the barrier. For an array access in this design, the compiler generates a load of the appropriate indirection pointer from the spine based on the arraylet size, and then loads the array element at the proper arraylet offset (or the remainder offset), as shown in lines 5-10 of Figure 4.3(b). The arraylet size is a global constant, and we explore different values in Section 4.4.

4.2.2 Memory Management of Z-rays

Because all arraylets have the same size, we manage them with a special-purpose memory manager that is simple and efficient. Figure 4.1(b) shows the *arraylet space*. The arraylet space uses a non-copying collector with fixed-sized blocks equal to the arraylet size. The liveness of each arraylet is strictly determined by its parent spine. The collector requires one liveness bit per arraylet that we maintain in a side data structure. The arraylet allocator simply inspects liveness bits to find free blocks as needed. The arraylets associated with a given z-ray may be distributed across the arraylet space and interleaved with those from other z-rays according to where space is available at the time each arraylet is allocated. When the arraylet size is an integer multiple of the page size, OS virtual memory policies avoid fragmentation of physical memory. For arraylet sizes less than the page size, the live arraylets may fragment physical memory if they sparsely occupy pages. For more discussion, see Section 5.2.2. In principle the arraylet space can easily be defragmented since all arraylets are the same size (see Metronome's size-class defragmentation [7]), but we did not implemented this optimization.

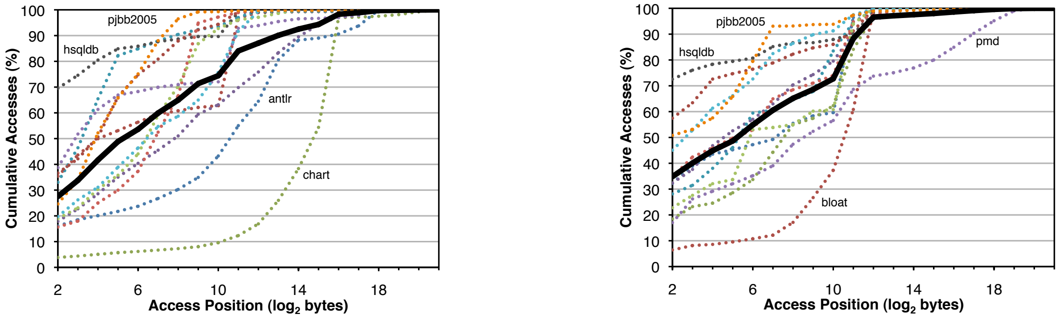
Z-rays help us side-step a standard problem faced when managing large objects within a copying garbage collector. While on the one hand it is preferable to avoid copying large objects, on the other hand it is convenient to define *age* in terms of object location. Historically, generational copying collectors either: (a) allocate large objects into the nursery and live with the overhead of copying them if they happen to be long-lived, (b) *pretenure* all large objects into a non-moving space and live with the memory overhead of untimely reclamation if they happen to be short-lived, or (c) separate the header and the payload of large arrays, via an indirection on every access, and use the header to reflect the array's age [36]. Jikes RVM currently adopts the first policy, and in the past has adopted the second. We adopt a modified version of the third approach for z-rays, avoiding untimely reclamation and expensive copying. We allocate spines into the nursery and arraylets into their own non-moving space. Nursery collections trace and promote spines to the old space if they survive, just like any other object. If a spine dies, its corresponding arraylets' liveness bits are cleared and the arraylets are immediately available for reuse. This approach limits the memory cost of short-lived and sparsely-populated arrays.

4.2.3 First-N Optimization

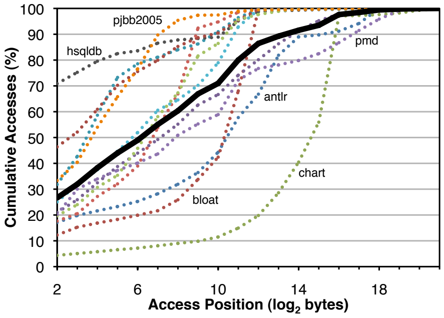
The basic arraylet design above does not perform well. While trying to optimize arraylets, we speculated that array access patterns may tend to be biased toward low indices and that this bias may provide an opportunity for optimization.

Motivation: Array Access Patterns. We instrumented Jikes RVM to gather array size and access characteristics. Figure 4.2 shows the cumulative distribution plots for all array accesses for 12 (DaCapo and pjbb2005) benchmarks (faint) and the geometric mean (dark). We plot 12 of 19 benchmarks to improve readability; the remaining 7 from SPECjvm98 have the same trend. Figures 4.2(a) and 4.2(b) show data for primitive arrays and reference arrays separately. Each curve shows the cumulative percentage of accesses as a function of access position, expressed in bytes, since types have different sizes. These statistics show that the majority of array accesses are to low access positions. Not surprisingly, Java programs tend to use many small arrays, in part because Java represents strings, which are common, and multi-dimensional arrays as nested 1-D arrays. Even for large arrays, many accesses bias towards the beginning due to common patterns such as search, lexicographic comparison, over-provisioning arrays, and using arrays to implement priority queues. Figure 4.2(c) shows that nearly 90% of all array accesses occur at access positions less than 2^{12} bytes (4KB). These results motivate an optimization that provides fast access to the commonly accessed, leading elements in the array.

To eliminate the indirection overhead on leading elements, the *first-N* optimization for z-rays inlines the first *N* bytes of each array into the spine, as shown



(a) Accesses To Primitive Arrays (b) Accesses To Reference Arrays



(c) Access To All Arrays

Figure 4.2: Cumulative distribution of array access positions, faint lines show 12 representative benchmarks (of 19) and solid line is overall average.

in Figure 4.1(b). By placing the first N bytes immediately after the header, the program directly accesses the first $E = \frac{N}{elementSize}$ elements as if the array were a regular contiguous array. We modify the compiler to generate conditional access barrier code that performs a single indexed load instruction for the first E elements and an indirection for the later elements (lines 7 and 9 respectively of Figure 4.3(c)). Arrays with fewer than E elements are not arrayletized at all. Compared to the basic discontiguous design, using a 4KB first-N saves an indirection on 90% of all array accesses. N is a global compile time constant, and Section 4.4 explores varying N. The first-N optimization significantly reduces z-ray overhead on every benchmark.

With $N = 2^{12}$, this optimization reduces the average total overhead by almost *half*, from 26.3% to 14.5%, which is discussed further in Section 4.4.1.

4.2.4 Lazy Allocation

A key motivation for discontiguous arrays is that they offer considerable flexibility over contiguous representations. Others exploit this flexibility to perform space optimizations. For example, Chen et al. observe that arrays are sometimes over-provisioned and sparsely populated, so they perform lazy allocation and zero-byte compression [21]. We borrow and modify these ideas.

Because accesses to arraylets go through a level of indirection, it is relatively straightforward to allocate an arraylet lazily, upon the first attempt to write it. Unused portions of an over-provisioned or sparsely populated array need never be backed with arraylets, saving space and time. A more aggressive optimization is possible in a language like Java that specifies that all objects are zero-initialized. We create a single immutable global *zero arraylet*, and all arraylet pointers initially point to the zero arraylet. Any non-zero arraylets are only instantiated after the first non-zero write to their index range. The zero arraylet is depicted in Figure 4.1(b). Lazy allocation introduces a potential race condition when multiple threads compete to instantiate an arraylet. Whereas Chen et al. do not describe a thread-safe implementation [21], we implement lazy allocation atomically to ensure safety. Section 4.4.1 shows that lazy allocation greatly improves space efficiency for some benchmarks, thereby reducing collector time and improving performance.

4.2.5 Zero Compression

Chen et al. perform aggressive compression of arraylets at the byte granularity, focusing only on space efficiency [21]. Their collection-time compression and application-time decompression on demand add considerable overhead, and make arraylets variable-sized. We employ a simpler approach to zero compression for z-rays. When the garbage collector scans an arraylet, if it is entirely zero, the collector frees it and redirects the referent indirection pointer to the zero arraylet. As with lazy allocation, any subsequent writes cause the allocator to instantiate a new arraylet.

Whereas standard collectors already scan reference arrays, zero-compression additionally needs to scan primitive arrays. Scanning for all zeros, however, is cheap, because it has good spatial locality and because the code sequence for scanning power-of-two aligned data is simple and quickly short-circuited when it hits the first non-zero byte. Our results show that the extra time the collector spends scanning primitives is compensated for by the reduction in the live memory footprint. Section 4.4.1 shows that this space saving optimization improves overall garbage collection time and thus total time.

4.2.6 Fast Array Copy

The Java language includes an explicit `arraycopy` API to support efficient copying of arrays. The API is general: programs may copy subarrays at arbitrary source and target offsets. When arrays or copy ranges are non-overlapping, as is common, the standard implementation of `arraycopy` uses fast, low-level byte copy instructions. In other cases, correctness requires that the copy be performed with simple element-by-element assignments. Furthermore, textjavaarraycopy must notify the garbage collector when reference arrays are copied since the copy may generate new inter-space pointers (such as old-to-young) of which the garbage collector must be aware.

Discontiguous arrays complicate the optimization of `arraycopy` because copying must respect arraylet boundaries. In practice, fast contiguous copying is limited by the alignment of the source and destination indices, the arraylet size, and the first-*N* size. Our default `arraycopy` implementation performs simple element-by-element assignments using the general form of the arraylet read and write barriers. We also implement a fast `arraycopy` which strip-mines for both the first-*N* (direct access) and for each overlapping portion of source and target arraylets, hoisting the barriers out of the loop and performing bulk copies wherever possible. Since `arraycopy` is widely used in Java applications, optimizing for z-rays is crucial to attaining high performance, as we show in Section 4.4.1.

4.2.7 Copy-on-Write

Z-rays introduce a copy-on-write (COW) optimization for arrays. In the special case during an `arraycopy` where the range of both the source and the destination are aligned to arraylet boundaries, we elide the copy and share the arraylet by setting both indirection pointers to the source arraylet's address. Figure 4.1(b) shows the topmost arraylet being shared by three arrays. To indicate sharing, we *taint* all shared indirection pointers by setting their lowest bit to 1. When the mutator or collector reads an array element beyond *N*, they mask out the lowest bit of the indirection pointer. If a write accesses a shared arraylet, our barrier lazily allocates a copy and atomically installs the new pointer in the spine before modifying the arraylet. COW is a generalization of lazy allocation and zero compression techniques to non-zero arraylets. We find that COW reduces performance slightly, but reduces space usage.

4.3 Runtime Implementation

We now describe key details of our z-ray implementation. Our design affects three key aspects of a runtime implementation: allocation, garbage collection, and array loads and stores. Static configuration parameters turn on and off our five optimizations, and set the size of arraylets and first *N* bytes.

Array Allocation. For z-rays, we modify the standard allocation sequence. If the array size is less than the first-*N* size, then the allocation sequence allocates a regular contiguous array. Otherwise, the allocator establishes the size of the spine and number of arraylets based on array length, arraylet size, and the first-*N* size. It allocates the spine into the nursery and initializes the indirection pointers to the zero arraylet. The allocator points the last indirection pointer to the first remainder element within the spine. The spine header records the length of the entire array, not the length of the spine, thus array bounds checks proceed unchanged.

Garbage Collection. We organize the heap into a copying nursery, an arraylet space, and a standard free-list mature space for all other objects [14]. Spines initially reside in the copying nursery space. A nursery collection reclaims or promotes spines just like any other object, copying surviving spines to the mature space. The only special action for the spine is to update the indirection pointer to the remainder such that it correctly reflects its new memory location. Recall that the remainder resides within the spine. The scan of z-rays traces through the indirection pointers, ignoring pointers to the zero arraylet. The collector performs zero compression, as discussed in Section 4.2.5. For each non-zero arraylet, the collector marks the liveness bit. During lazy allocation, we mark the liveness bit of arraylets whose spines are mature so that they will not be collected during the next nursery collection. Full heap collections clear all arraylet mark bits before tracing. Our arraylet space manager avoids an explicit free list and instead lazily sweeps through the arraylet mark bits at allocation time, reusing unmarked arraylets on demand.

Read and Write Barriers We modify the implementation of array loads and stores to perform an indirection to an arraylet and remainder when necessary. With

```
void arrayStore(Address array, int index, int value) {
  int len = array.length;
  if (index >= len)
    throw new ArrayBoundsException();
  int offset = index * BYTES_IN_INT;
  array.store(offset, value);
}
```

(a) Array store (contiguous array).

```
void arrayStore(Address array, int index, int value) {
  int len = array.length;
  if (index >= len)
    throw new ArrayBoundsException();
  int offset = index * BYTES_IN_INT;
  int arrayletNum = index / INTS_IN_ARRAYLET;
  int spineOffset = arrayletNum * BYTES_IN_ADDRESS;
  Address arraylet = array.loadAddress(spineOffset);
  offset = offset % ARRAYLET_BYTES;
  arraylet.store(offset, value);
}
```

(b) Array store (conventional arraylet).

```
void arrayStore(Address array, int index, int value) {
  int len = array.length;
  if (index >= len)
    throw new ArrayBoundsException();
  int offset = index * BYTES_IN_INT;
  if (offset < FIRST_N_BYTES)
    array.store(offset, value);
  else
    arrayletStore(array, offset, value);
}
```

(c) Array store fast path (z-rays)

```
@NoInline          // force this code out of line
void arrayletStore(Address spine,int offset,int value){
  int arrayletNum =
    (offset - FIRST_N_BYTES) / BYTES_IN_ARRAYLET;
  int spineOffset =
    FIRST_N_BYTES + arrayletNum * BYTES_IN_ADDRESS;
  Address arraylet = spine.loadAddress(spineOffset);
  if (arraylet & SHARING_TAINT_BIT != 0)
    ...            // atomic copy on write
  else if (arraylet == ZERO_ARRAYLET)
    if (value == 0)
      return;      // nothing to do
    else
      ...          // lazy allocation and atomic update
  offset = (offset - FIRST_N_BYTES) % BYTES_IN_ARRAYLET;
  arraylet.store(offset, value);
}
```

(d) Array store slow path (z-rays)

Figure 4.3: Storing a value to a Java `int` array.

the first-*N* optimization, accesses to byte positions less than or equal to *N* proceed unmodified, using a standard indexed load or store (line 7 of Figure 4.3(c)). Otherwise, basic arithmetic (shown in lines 5–9 of Figure 4.3(b)) identifies the relevant indirection pointer and offset within the arraylet. Lazy allocation and zero compression do not affect reads, except that the read barrier returns zero instead of loading from the zero arraylet. Copy-on-write requires read barriers that traverse indirection pointers to mask out the lowest bit in case the pointer is tainted. If the write barrier finds an arraylet indirection pointer tainted by COW, it lazily allocates an arraylet, copies the original, and atomically installs the indirection pointer in the spine. If the write barrier intercepts a non-zero write to the zero arraylet, it lazily allocates an arraylet filled with zeros and installs the indirection pointer atomically. Both of these write barriers then proceed with the write. Figures 4.3(c) and 4.3(d) show pseudocode for the fast and slow paths of a z-ray store with the first-*N* optimization, lazy allocation, zero compression, and copy-on-write.

Adding complexity to barriers does increase the code size; we found on average we added 20% extra code space to our benchmarks for our z-ray implementation. To measure the extra code, we did experiments using Jikes RVM's compilation replay mechanism to avoid the problem of non-determinism from adaptive optimization, instead picking a fixed optimization plan per benchmark via profiling [13].

With a generational collector, an object's age is often defined by the heap space in which it is currently located. To find mature-to-nursery pointers, a typical generational write barrier tests the location of the *source reference* against the location of the destination object [15]. Since the source reference in our case could reside in the arraylet space, which does not indicate age, our generational array write barrier instead tests the location of the *source spine*, which defines the arraylets' age, against the destination object.

Further Barrier Optimization. Prior work notes that classic compiler optimizations have the potential to reduce the overhead of discontiguous arrays [8]. Although they do not implement it, Bacon et al. advocated loop strip-mining, which hoists loop invariant barrier code when arrays access elements sequentially. Instead of performing *n* indirection loads for *n* sequential arraylet element accesses, where *n* is the number of elements in an arraylet, this optimization performs only *one* indirection load for *n* consecutive accesses. Our fast array-copy performs this optimization, and it is very effective for benchmarks that make heavy use of `arraycopy` (see Section 4.4.1). Although we do not implement this optimization more generally in the compiler, we performed a microbenchmark study to determine its potential benefit. For a simple test application sequentially iterating over a large array, a custom-coded strip-mining implementation showed zero overhead and actually ran slightly faster than the original system compared to the implementation without strip-mining, which demonstrated a 37% slowdown on this microbenchmark. Strip-mining has the potential to reduce the overhead of discontiguous arrays further, particularly for programs that perform a large percentage of array accesses beyond the first-*N* threshold.

4.3.1 Jikes RVM-Specific Implementation

Our z-ray implementation has a few details specific to Jikes RVM. Jikes RVM is a Java-in-Java VM, and as a consequence, the VM itself is compiled ahead of time, and the resulting code and data necessary for bootstrap are stored in a *boot image*. At startup, the VM bootstraps itself by mapping the boot image into memory. The process of allocating and initializing objects in the boot image is entirely different from application allocation. Since there is no separate arraylet space at boot image building time, boot image arraylets are part of the immortal Jikes RVM boot image. For simplicity we allocate each z-ray by laying out the spine followed by each of the arraylets (which must be eagerly allocated). Indirection pointers are implemented just as for regular heap arraylets, so our runtime code can be oblivious as to whether an arraylet resides in the boot image or the regular heap.

4.3.2 Implementation Lessons

The abstraction of contiguous arrays provided by high-level languages enables the implementation of discontiguous arrays. Although the language guarantees that user code will observe these abstractions, unfortunately, under the hood, modern high performance VMs routinely subvert them in three scenarios. (1) User-provided native code accesses Java objects via the Java Native Interface. (2) The VM accesses Java objects via its own high-performance native interfaces, for example, for performance critical native VM operations such as IO. (3) The VM interacts with internals of Java objects, for example, the VM may directly access various metadata which is ostensibly pure Java. Note that none of these issues are particular to Jikes RVM; they are issues for all JVMs. Implementing discontiguous arrays is a substantial engineering challenge because the implementer has to identify every instance where the VM subverts the contiguous array abstraction and then engineer an efficient alternative.

We found all explicit calls to native interfaces (scenarios 1 and 2). At each call, we marshal array data into and out of discontiguous form. In general, marshaling incurs overhead but it is relatively small because VMs *already* copy such data out of the regular Java heap to prevent the garbage collector from moving it while native code is accessing it. Another alternative is excluding certain arrays from arrayletization entirely, and pinning them in the heap. We chose to arrayletize all Java arrays.

A more insidious problem is when the VM subverts the array abstraction by directly accessing metadata, such as compiled machine code, stacks, and dispatch tables (3). The problem arises because Jikes RVM accesses this metadata both as raw bytes of memory and as Java arrays. We establish an invariant that forbids the implementation from alternating between treating discontiguous arrays as both raw bytes and Java arrays. Instead, this metadata now is implemented as a *magic* array type that is not arrayletizable, so accesses can proceed normally [33]. We thus exploit strong typing to statically enforce the differentiation of Java arrays from low-level, non-arrayletized objects, and access each properly.

To debug our discontiguous array implementation, we implemented a tool based on Valgrind [56] that performs fine-grained memory protection, cooperating with the VM to find illegal array accesses. Jikes RVM runs on top of Valgrind, which we modified to protect memory at the byte-granularity. We used Valgrind to 'protect' each array and implement a thread-safe barrier that permits reads and writes to protected arrays. Accesses to protected arrays that do not go through the barrier caused an immediate segmentation fault (instead of corrupting the heap and manifesting much later), and generated an exception that we used to track down offending array accesses. We plan to make this valuable debugging tool available with our z-ray implementation.

4.3.3 Benchmarks and Methodology

This section describes our experimental methodology.

Benchmarks. We use the DaCapo benchmark suite, version 2006-10-MR2 [13], the SPECjvm98 suite, and pjbb2005, which is a variant of SPECjbb2005 [27] that holds the workload, instead of time, constant. We configure pjbb2005 with 8 warehouses and 10,000 transactions per warehouse. Of these 19 benchmarks, pjbb2005, hsqldb, lusearch, xalan, and mtrt are multi-threaded.

Experimental Platforms. Our primary experimental machine is a 2.4 GHz Core 2 Duo with 4 MB of L2 cache and 2 GB of memory. To ensure our approach is applicable across architectures, we also measure it on a 1.6 GHz Intel Atom two-way SMT in-order processor with 512 KB of L2 cache and 2 GB of memory. The Intel Atom is a cheap, low power in-order processor targeted at portable devices, and so more closely approximates architectures found in embedded processors. All machines run Ubuntu 8.10 with a 2.6.24 Linux kernel. All experiments were conducted using two processors. We use two hardware threads for the Atom.

JVM Configurations and Experimental Design. We made our z-ray changes to the 3.0.1 release of the Jikes Research Virtual Machine. All results on z-rays are presented as a percentage overhead over the vanilla Jikes RVM 3.0.1 that uses a contiguous array implementation. We use the Jikes RVM's default high-performance configuration ('production'), which uses adaptive optimizing compilation and a generational mark-sweep garbage collector. To maximize performance, we use *profiled* Jikes RVM builds, where the build system gathers a profile of only the VM, not the application, and uses it to build a highly-optimized Jikes RVM boot image. We use a heap size of 2× the minimum required for each individual benchmark as our default. This heap size reflects moderate heap pressure, providing a reasonable garbage collector load on most benchmarks. We also perform experiments with z-rays over a range of heap sizes.

As recommended by Blackburn et al., we use the adaptive experimental compilation methodology [13]. Our z-ray implementation changes the barriers in the application code, and therefore interacts with the adaptive optimizer. We run each benchmark 20 times to account for non-determinism introduced through adaptive optimization, and in each of the 20 executions, we measure the 10th iteration to sufficiently warm up the JVM. We calculate and plot 95% confidence intervals. Despite this methodology, some results remain noisy. For total time, only hsqldb is noisy. Garbage collection time is chaotic because of varying allocation load under the adaptive methodology, even without z-rays. Many of the garbage collection results are therefore too noisy to be relied upon for detailed analysis. We gray out noisy results in Table 4.3 and exclude them from the reported minimums, maximums, and geometric means.

	Allocation			**Heap**		**Accesses**					**Array Copy**	
	MB/	**Array %**		**Composition**		**per**	**write %**		**read %**		**byte**	**%**
Benchmark	**μsec**	**all**	**prim.**	**MB**	**%**	**μsec**	**fast**	**slow**	**fast**	**slow**	**μsec**	**$>N$**
antlr	72	83	80	12	52	157	9.3	7.6	73.5	9.6	52	23
bloat	77	65	60	18	51	264	1.0	0.4	97.8	0.8	52	0
chart	23	49	48	18	49	320	5.3	7.1	49.8	37.8	44	76
eclipse	57	75	55	38	57	373	4.6	1.4	89.4	4.7	30	25
fop	11	34	26	19	47	94	1.7	0.1	97.3	0.9	5	0
hsqldb	29	38	21	67	31	463	0.7	0.3	98.1	0.9	5	16
jython	125	77	66	24	51	584	1.2	0.3	98.0	0.6	132	3
luindex	32	40	36	12	52	186	28.6	0.2	70.7	0.5	21	0
lusearch	201	87	82	15	57	699	14.5	0.5	84.1	1.0	31	8
pmd	156	33	1	23	45	419	0.9	1.01	96.2	1.9	7	69
xalan	766	88	52	31	73	342	7.5	0.24	91.5	0.7	41	0
compress	24	100	100	4	57	191	12.9	22.5	25.3	39.3	0	0
db	4	64	9	11	56	48	0.8	8.9	65.8	24.4	15	99
jack	28	32	26	6	51	92	4.8	0.2	94.3	0.7	49	0
javac	22	49	42	12	41	106	7.3	0.4	90.9	1.4	6	4
jess	75	47	0	7	54	197	1.9	0.2	97.1	0.8	66	0
mpegaudio	0.2	15	6	3	52	669	14.3	0.1	85.5	0.1	35	0
mtrt	30	25	18	9	42	267	4.3	0.2	95.2	0.3	0	0
pjbb2005	70	63	42	193	64	1109	2.4	0.3	96.5	0.8	271	0
min	**0.2**	**15**	**0**	**3**	**31**	**48**	**0.7**	**0.1**	**25.3**	**0.1**	**0**	**0**
max	**766**	**100**	**100**	**193**	**73**	**1109**	**28.6**	**22.5**	**98.1**	**39.3**	**271**	**99**
mean	**47**	**56**	**40**	**-**	**52**	**338**	**6.4**	**2.6**	**84.6**	**6.4**	**45**	**17**

Table 4.1: Allocation, heap composition, and array access characteristics of each benchmark.

Benchmark Characterization. Table 4.1 characterizes the allocation, heap composition, array access, and array copy patterns for each of the benchmarks. This table shows the intensity of array operations for our benchmarks. Note that array accesses, and not allocation, primarily determine discontiguous array performance. The table shows allocation rate (total MB per μsec allocated), the percent of allocation due to all arrays and to just primitive (non-reference) arrays. On average, 56% of all allocation in these standard benchmarks is due to arrays, and 40% of all allocation is primitive arrays, which motivates optimizing arrays. By contrast, columns five and six measure heap composition by sampling the heap every 1MB of allocation, then averaging over those samples. For example, chart has 18MB live in the heap on average, of which 49% is arrays. Column 7 shows array access rate, measured in accesses per μsec. For instance, compress is a simple benchmark that iterates over arrays and might even be considered an array microbenchmark, but it has a much lower array access rate than many of the more complex benchmarks, such as pjbb2005. In summary, arrays constitute a large portion of the heap and are frequently accessed.

Columns 8 through 12 show the distribution of array read and write accesses over the *fast* and *slow* paths of barriers (recall Figures 4.3(c) and 4.3(d)). Fast path accesses are to elements within first-*N*, which we set to 2^{12} bytes. These statistics show the potential of first-*N* to reduce overhead. The vast majority of array accesses (84.6% on average) are reads and only exercise the fast path. There are a few outliers: chart, compress, and db exercise slow paths frequently and luindex, lusearch, and compress have a large percentage of write accesses. Note that although lusearch, mpegaudio, and pjbb2005 are the most array-intensive (699, 669, and 1,109 accesses per μsec respectively), they rarely exercise the slow paths. Overall 91% of all accesses go through the fast path, thereby enabling the first-*N* optimization to greatly reduce overhead by avoiding indirection on each of those accesses. The last two columns measure `arraycopy()`: (1) the number of bytes array copied per unit execution, measured in bytes copied per μsec, and (2) the percentage of array bytes copied that correspond to array indices beyond first-N. Some benchmarks use array copy intensively, including jython, jess and pjbb2005, but they rarely copy past first-N. Other benchmarks, such as chart copy a moderate amount *and* the majority of bytes are beyond first-N.

4.4 Z-ray Evaluation

This section explores the effect of z-rays with respect to time efficiency and space consumption.

4.4.1 Efficiency

We first show that z-rays perform well in comparison to previously described optimizations for discontiguous arrays. We break down performance into key contributing factors. We tease apart the extent to which individual optimizations contribute to overall performance, showing that first-N is the most effective optimization, and that first-N improves the effect of other optimizations. We go into detail about certain outlier results and describe performance models we create to explain them. We then show that z-ray performance is robust to variation in key configuration parameters.

Z-ray Summary Performance Results

This section summarizes the performance overhead of z-rays and compares to previously published optimizations. Table 4.2 shows the optimizations and key parameters used in each of the five systems we compare. The Naive configuration includes no optimizations and a 2^{10} byte arraylet size. The Naive A and Naive B configurations are based on Naive, but reflect the configurations and optimizations described

	Naive	**Naive A** [21]	**Naive B** [7]	**Z-ray**	**Perf Z-ray**
Arraylet Bytes	2^{10}	2^{10}	2^{11}	2^{10}	2^{10}
First-N				2^{12}	2^{12}
Lazy Alloc		✓		✓	✓
Zero Compress				✓	✓
Array Copy				✓	✓
Copy-on-Write				✓	
Overhead	27.4%	31.9%	27.5%	14.5%	12.7%

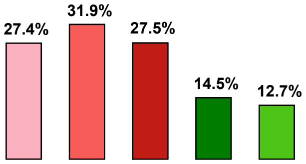


Table 4.2: Overview of arraylet configurations and their overhead.

by Chen et al. [21] and Bacon et al. [7] respectively. These configurations are not a direct comparison to prior work, because, for example, we do not implement the same compression scheme as Chen et al. However, this comparison does allow us to directly compare the efficacy of previously described optimizations for discontiguous arrays within a single system. The Naive A configuration adds lazy allocation [21] while Naive B raises the arraylet size [7]. The Z-ray configuration includes all optimizations. The Perf Z-ray configuration is the best performing configuration, and differs from the Z-ray configuration only by its omission of the copy-on-write (COW) optimization.

Table 4.2 summarizes our results in terms of average time overheads relative to an unmodified Jikes RVM 3.0.1 system. These numbers demonstrate that both Z-ray and Perf Z-ray comprehensively outperform prior work. The configurations based on the optimizations used by Chen et al. [21] (Naive A) and Bacon et al. [7] (Naive B) have average overheads of 32% and 27% respectively on the Core 2 Duo whereas Perf Z-ray reduces overhead to 12.7%. Notice that Naive (27%) performs better than Naive A (Naive with lazy allocation), showing that lazy allocation by itself slows programs down. Our Z-ray configuration, with all optimizations turned on including COW, has an average overhead of 14.5%, slightly slower than our best-performing system, Perf Z-ray at 12.7%.

Per-benchmark Configuration Comparison. Figure 4.4 compares the performance of Z-ray and Perf Z-ray against previously published optimizations for all benchmarks. Perf Z-ray outperforms prior work (Naive A and Naive B) on every benchmark. The configurations Naive A and Naive B at best have overheads of 7% and 10% respectively, while Perf Z-ray at best *improves* performance by 5.5%. While our system sees a worst case overhead of 57% on chart, Naive A and Naive B slow down chart by 74% and 62%, and suffer worst case slowdowns across all benchmarks of 107% and 76% respectively. On jython, Naive A and Naive B suffer overheads of 88% and 76% respectively, which we reduce to just 5.7%. In general, Naive, our system without any optimizations, matches the performance of Naive B, although it uses a smaller arraylet size. In 17 of 19 benchmarks excluding antlr and fop, the Z-ray configuration improves over prior work optimizations, but as mentioned, the copy-on-write optimization, while improving space usage, does add time overhead as compared with Perf Z-ray. In summary, compared to previously published optimizations, Perf Z-ray improves every benchmark, some enormously, and reduces the average total overhead by more than half.

Performance Breakdown and Architecture Variations

We now examine the z-ray overheads in more detail. We break down contributions to the overhead from the collector, mutator, reference arrays, and primitive arrays.

	Total Overhead (%)		C2D Overhead Breakdown (%)			
Benchmark	**C2D**	**Atom**	**Ref.**	**Prim.**	**Mutator**	**GC**
antlr	22.0 ±8.2	37.7 ±12.3	-3.2	14.4	17.9	98.2
bloat	15.9 ±2.0	28.7 ±8.6	4.3	11.4	14.2	73.9
chart	57.2 ±0.4	54.9 ±0.3	0.2	57.0	61.4	-6.9
eclipse	14.2 ±1.2	24.9 ±7.3	1.9	10.3	15.7	-28.1
fop	5.1 ±3.7	19.0 ±9.0	8.9	14.2	4.4	33.6
hsqldb	23.8 ±24.5	7.5 ±1.8	2.2	33.9	26.9	12.9
jython	5.7 ±1.1	12.6 ±3.2	2.6	2.8	5.0	60.9
lusearch	22.4 ±1.3	24.0 ±0.9	4.2	23.9	22.6	18.3
luindex	10.1 ±0.9	14.9 ±1.0	1.3	10.4	9.6	26.8
pmd	6.0 ±1.3	7.2 ±1.2	5.5	0.8	7.9	-19.4
xalan	-5.5 ±1.3	11.1 ±2.7	-4.8	-0.7	2.0	-56.0
compress	20.2 ±0.3	51.2 ±0.4	0.4	20.3	21.9	-82.9
db	3.7 ±0.1	14.0 ±0.1	3.4	-0.4	3.8	-4.0
jack	5.9 ±1.6	7.6 ±1.1	0.3	4.7	6.6	-15.6
javac	8.0 ±0.6	11.5 ±1.2	2.2	5.9	8.3	4.2
jess	12.2 ±1.0	17.0 ±2.8	10.3	1.4	12.0	29.0
mpegaudio	31.4 ±0.4	44.1 ±0.6	2.3	14.4	31.2	358.0
mtrt	4.2 ±1.7	6.8 ±1.6	1.4	3.4	4.4	1.7
pjbb2005	3.4 ±0.5	5.1 ±2.5	-0.1	0.6	3.6	0.6
min	**-5.5**	**5.1**	**-4.8**	**-0.7**	**2.0**	**-56.0**
max	**57.2**	**54.9**	**10.3**	**57.0**	**61.4**	**4.2**
geomean	**12.7**	**20.2**	**2.2**	**10.1**	**13.3**	**-11.3**

Table 4.3: Time overhead of Perf Z-ray compared to base system on the Core 2 Duo and Atom. 95% confidence intervals are in small type. Breakdown of overheads on Core 2 Duo for reference, primitive, mutator, and garbage collector are shown at right. Noisy results are in gray and are excluded from min, max, and geomean.

We also assess sensitivity to heap size variation and different micro-architectures.

Throughout the remainder of our performance evaluation, unless otherwise specified, our primary point of comparison is the best-performing z-ray configuration, Perf Z-ray, which disables copy-on-write. Table 4.3 shows total time overheads for the Perf Z-ray configuration for the Core 2 Duo and Atom processors relative to an unmodified Jikes RVM 3.0.1. The table includes 95% confidence intervals in small type next to each total overhead percentage. The confidence intervals are calculated using the student's t-test, and each reflects the interval for which there is a 95% probability that the true ‘result’ (the mean performance of the system being measured) is within that interval. Noisy results, which are those with a 95% confidence interval greater than 20% of the mean performance (±10%), are grayed out and excluded from geometric means. The total overhead on the Core 2 Duo is 12.7% on average.

Many benchmarks have low overhead, with xalan as the best, speeding up execution by 5.5% due to greatly reduced collection time. Despite some high overheads, z-rays perform well on xalan, db, mtrt, and pjbb2005. Because eclipse, xalan, and compress have many arrays larger than first-*N*, lazy allocation is particularly effective at reducing space consumption which, in turn, improves garbage collection time. The benchmarks antlr, chart, lusearch, compress, and mpegaudio use primitive arrays intensively which is the main source of their overheads. Table 4.3 shows that benchmark overhead comes primarily from primitive (‘Prim.’) discontiguous arrays, and we find in particular that byte and char arrays are the main contributors to overhead, each adding on average over 3%, because they are used extensively for I/O and file processing using numerous large arrays. By contrast, when arraylets are selectively applied only to reference arrays, both average and worst case overheads are reduced by about a factor of six to just 2.2% and 10.3% respectively.

Mutator Performance. Following standard garbage collection terminology, we use the terms *mutator* to refer to application activity, and *collector* to refer to garbage collection (GC) activity. The ‘Mutator’ column of Table 4.3 shows that most of the overhead of the Perf Z-ray configuration is due to the mutator. Mutator performance is directly affected through the allocation of discontiguous arrays and the execution of array access barriers. We see that chart—which according to our earlier analysis performs a large number of array accesses beyond the inlined first *N* bytes—suffers a significant mutator performance hit of 61.4%. On the other hand, xalan suffers only 2% mutator overhead.

Collector Performance. Z-rays affect the collector performance both directly, through the cost of processing spines and arraylets during collection, and indirectly, by changing how often the VM requires garbage collection due to changes in space efficiency. The 'GC' column of Table 4.3 shows that collector performance for our Perf Z-ray configuration varies significantly. Note that garbage collection exhibits chaotic performance characteristics because perturbations in the mutator can affect the volume of data allocated and the timing of collections, inducing large fluctuations in collector performance [13]. Many of the garbage collection results are consequently noisy. Among the more significant results, xalan improves collection time by 56% and javac degrades by 4%. Across those benchmarks reporting reliable garbage collection results, z-rays showed an average reduction in collection time of 11.3%.

Heap Sizes. We evaluated z-rays against a range of heap sizes to measure time-space trade-offs of garbage collection. Our collector and mutator overheads are robust across heap sizes, tracking the performance of unmodified Jikes RVM from very tight to large heaps. Because most program time and overheads are in the mutator, and collector improvements are modest, this result is not unexpected. Because the collector dominates at small heap sizes, our total overhead relative to the base system is lowest in small heaps. These results show that our good overall performance is derived from (1) a modest overall mutator overhead, and (2) an overall improvement in collector performance.

Architectural Sensitivity. To assess the architectural sensitivity of our approach, we performed experiments on two very different Intel x86 architectures: Core 2 Duo and Atom. On the Atom, the Perf Z-ray overhead increases to 20.2% from Core 2 Duo's 12.7%. In comparison, average overheads for previous designs, Naive A and Naive B, on Atom increase to 39% and 33% respectively (not shown in the table). The Atom is an in-order processor, so it is less able to mask overheads with instruction level parallelism.

In summary, z-ray performance varies significantly across benchmarks; overheads are overwhelmingly due to the mutator; primitive array types account for almost all of the z-ray overhead; and arraylet overheads are more exposed on an in-order processor.

Efficacy of Individual Optimizations

Figure 4.5 explores the effect of each of the optimizations. In this graph, overheads are expressed with respect to Z-ray, our configuration with all optimizations enabled. Arraylet size and the number of inlined first *N* bytes are held constant. We evaluate the effect of removing from Z-ray each of: the first-*N* optimization (Z-ray – FirstN), lazy arraylet allocation (Z-ray – Lazy), zero compression (Z-ray – Zero), fast array copy (Z-ray – Fast AC), and copy-on-write (Z-ray – COW $\equiv$ Perf Z-ray). When taking away an optimization, a slowdown, or positive overhead in Figure 4.5, indicates the utility of that optimization.

Omitting first-*N* comes at the most significant performance cost across the board, as expected, increasing the overhead by up to 71% in the worst case and 10% on average. Inlining the first-*N* bytes is key to reducing the overhead of discontiguous arrays and central to our approach. For mpegaudio in particular, as well as luindex, lusearch, jython, and jess, the first-N optimization significantly reduces the overhead of discontiguous arrays, particularly for primitive arrays. Furthermore, because first-N moves arraylet accesses off the critical path (Figures 4.3(c) and 4.3(d)), other optimizations such as lazy allocation that add overhead to each arraylet access become profitable. As already noted, lazy allocation adds an additional 4% of overhead to a naive system on average; compare Naive A and Naive in Figure 4.4. By contrast, lazy allocation adds no overhead to z-rays, on average: see Z-ray – Lazy, Figure 4.5.

Omitting lazy allocation (Z-ray – Lazy) has slightly more of an impact on efficiency than omitting zero compression (Z-ray – Zero), but on average both achieve performance very similar to the Z-ray configuration. Some benchmarks, in particular xalan, perform significantly better when enabling these optimizations.

Omitting fast array copy degrades performance of z-rays by on average 2.8%. Fast array copy significantly benefits chart, jython and jess, all of which frequently copy arrays. Since our array copy optimization strip-mines both the first-N test and the arraylet access logic, benchmarks that perform a lot of array copies benefit even when copying many small arrays. Improvements from strip-mining the first-N test explain the substantial benefit to jython, which copies a lot, though only 3% of copied bytes are beyond first-N, as shown in Table 4.1.

Figure 4.5 shows that copy-on-write adds a small amount of overhead, on average 1.8%, due to extra checks in barriers for tainted arraylet pointers. However, Section 4.4.2 shows that copy-on-write is effective at reducing space in the heap.

In summary, first-N is by far the most important optimization overall, and a fast array copy implementation is critical to a number of benchmarks.

Understanding and Modeling Performance Overhead

We now discuss our use of microbenchmarks and a simple analytical model to further understand z-ray performance overheads.

Table 4.3 shows that a small number of benchmarks suffer significant overheads, and Figure 4.4 shows that z-rays only improve modestly over prior work on chart. Since most of our improvement over previous designs comes from first-*N*, our difficulty in improving chart is unsurprising given the array access statistics seen in Figure 4.2 and Table 4.1, which show that chart is an outlier, with 80% of array accesses indexing beyond the first 2^{12} bytes, and 45% of array accesses taking the slow path. Figure 4.5 confirms that chart is one of the only benchmarks that does not significantly benefit from the first-*N* optimization.

To better understand the nature of this overhead, we construct a simple analytical model using a set of microbenchmarks. We wrote microbenchmarks to measure the performance of a tight loop of array access operations under controlled circumstances, generating results across the following dimensions:

- fast vs. slow-path access
- read vs. write
- array element type
- random vs. sequential access

By measuring performance for each microbenchmark in both the z-ray-modified and base VMs, we calculate an approximate overhead in terms of *milliseconds per million array operations* for each point in the cross product of the dimensions above. We then use these overheads to model and estimate the overhead incurred by z-rays for each benchmark using access statistics from Table 4.1. We found analyzing the machine code of each simple microbenchmark more tenable than inspecting all benchmark machine code to explain results. For chart, the model estimates an overhead of between 51% and 100%, with sequential and random access patterns, respectively, which explains our measured overhead of 57%. These results confirm that it would be necessary to leverage additional optimization techniques such as strip-mining to further reduce the overhead of chart.

Table 4.3 shows that compress suffers an overhead of 20%, which is in part because 99.9% of array bytes allocated are in arrays that are larger than first-*N*. Similarly, we see in Table 4.1 that compress has a high percentage of both reads and writes on the slow path, which are to primitive arrays. Even so, Figure 4.4 shows that z-rays outperform prior work on compress and Figure 4.5 shows that it is lazy allocation and the first-*N* optimization that help z-ray performance on compress. Section 4.4.2 shows that compress has significant space savings which explains its modest overhead.

These experiments serve to validate our observed performance overheads and suggest that strip-mining might be particularly effective in reducing our most significant overheads.

Sensitivity to Configuration Parameters

We now explore how performance is affected by varying our two key configuration parameters: the number of first *N* bytes inlined and the arraylet size.

Figure 4.7 shows the effect of altering the number of bytes inlined with the first-*N* optimization across the range 2^6 to 2^{18} with the arraylet size held constant at 2^{10} bytes. While extremely large values of *N*, such as those greater than 2^{12}, deliver slightly better performance on average, such high values for *N* may be unrealistic, especially for real-time. In the case of chart, setting first-*N* to 2^{18} roughly halves the overhead. For such large values of *N*, the system approaches a contiguous array system since very few arrays are large enough to have an arrayletized component, eroding any utility offered by arraylets, including the ability to bound collection time and space. Setting *N* to 2^{12} provides good performance, while also providing reasonable bounds. It is worth noting that the smaller values for *N* still deliver configurations with reasonably low performance overheads, and may be good choices for some system designs.

Figure 4.6 shows the effect of varying arraylet size from 2^8 to 2^{12} bytes, with the number of inlined first *N* bytes constant at 2^{12}. We see that changing the arraylet size overall does not affect performance much. However, in terms of space, initial tests show that when the arraylet size is lowered from 2^{10} to 2^8, our zero compression, lazy allocation, and COW optimizations are more effective, reducing the heap size further (see Section 4.4.2).

These results show that it is possible to significantly vary both the number of inlined first *N* bytes and the arraylet size while maintaining overheads at reasonable levels. While the values used in our Z-ray configuration are a good choice in our setting, language implementers should tune these parameters to satisfy their particular design criteria.

4.4.2 Flexibility

Previous work has demonstrated the flexibility of discontiguous arrays [7, 21]. While we primarily target improving the running-time performance of a general-purpose system in our evaluation, we show here how z-ray optimizations improve space efficiency. Chapter 5 discusses the impact of discontiguous arrays on heap fragmentation.

Space Efficiency

Benchmark	% Alloc Large	% Savings			Total Heap Footprint	
		Lazy	Zero	COW	% Arrayletizable	% Saved
antlr	55.4	26.0	17.6	10.5	6.4	3.4
bloat	1.2	17.2	16.1	32.2	4.7	1.6
chart	39.4	16.1	14.3	22.9	7.1	2.6
eclipse	37.9	30.7	15.6	11.4	6.9	2.9
fop	6.5	8.9	15.0	60.1	0.9	-1.7
hsqldb	10.0	0.7	4.2	100.0	5.0	0.9
jython	4.2	1.2	13.2	5.7	3.8	1.0
luindex	1.6	1.9	18.2	30.7	5.5	2.5
lusearch	1.0	0.1	7.2	16.8	10.8	0.0
pmd	70.9	24.1	4.9	48.6	10.6	4.0
xalan	64.5	75.6	1.4	7.1	27.0	25.0
compress	99.9	26.2	3.3	0.0	60.4	49.1
db	87.5	0.1	12.6	0.2	8.1	4.1
jack	1.5	78.3	23.4	0.0	9.4	6.2
javac	1.4	1.9	20.2	0.8	5.4	2.9
jess	0.0	20.7	22.3	0.0	8.1	5.0
mpegaudio	2.7	8.6	39.2	0.0	1.8	-1.3
mtrt	0.7	5.1	18.2	0.0	8.7	4.6
pjbb2005	0.3	39.7	3.5	0.0	1.4	0.6
min	**0.0**	**78.3**	**39.2**	**0.0**	**0.9**	**-1.7**
max	**99.9**	**0.1**	**1.4**	**100.0**	**60.4**	**49.1**
mean	**25.6**	**20.1**	**14.2**	**18.2**	**10.1**	**5.9**

Table 4.4: Effect of space saving optimizations.

One motivation for discontiguous arrays is that they offer additional flexibility for implementing space saving optimizations such as lazy allocation, zero compression, and arraylet copy-on-write. Table 4.4 presents space savings statistics gathered using the Z-ray configuration, showing the effect of each of the space saving optimizations. While Chen et al. [21] explore byte-grained compression, each of the optimizations we evaluate here operate at the granularity of entire arraylets: 2^{10} bytes.

The '% Alloc Large' column in Table 4.4 shows the fraction of allocated bytes that are due to large arrays, where large is $> 2^{12}$. The benchmarks with high overhead (antlr, chart, eclipse, and compress), all have a high percentage of large arrays. For example, 99.9% of compress's allocated bytes are due to large arrays.

The three '% Savings' columns demonstrate the efficacy of each of the individual space savings optimizations: lazy allocation, zero compression, and arraylet copy-on-write. The 'Lazy' column shows that on average, lazy allocation avoids allocating 20% of the space consumed by large arrays, meaning that 20% of large array bytes fall within arraylets to which the program never writes. Lazy allocation saves memory in all benchmarks, and in many cases yields substantial savings, e.g., 75.6% in xalan. The 'Zero' column gives the proportion of allocated arraylets that hold only zero values, as measured by taking snapshots of the heap after every 1MB of allocation. These results demonstrate that, on average, zero compression may reduce the volume of live arraylets in the heap by 14.2%. The 'COW' column shows that by sharing arraylets, copy-on-write avoids actually copying 18.2% of those bytes beyond first-*N* that are copied via `arraycopy`, and is extremely effective for hsqldb. Copy-on-write offers a trade-off: it costs around 1–2% in total performance but saves space.

The final two columns of Table 4.4 show the total reduction in heap footprint, also measured by taking heap snapshots after every 1MB of allocation. The '% Arrayletizable' column shows the percentage of the live heap consumed by arrayletizable bytes, beyond first-*N*, when *no* space saving optimizations are employed, which is on average 10.1%. The '% Saved' shows the combined effect of our three optimizations, and is expressed as a percentage of total heap footprint. In two benchmarks, the optimized system takes up slightly more heap space. However, xalan and compress save 25% and 49% of the heap, respectively, due to compression and sharing, which is 92% and 81% of arrayletized bytes. Results for compress agree with prior work [5]: about 50% of compress's heap is zero. The rest of the benchmarks save space modestly. Overall z-rays save about 6% of the heap, which as column 6 indicates, is about 60% of arrayletized bytes.

In summary, we find that each of our coarse-grained space saving optimizations yields savings, and that for some benchmarks, notably xalan and compress, these savings are substantial. Compression at a finer granularity could realize even more space savings, as shown in Chapter 3.

4.5 Discontiguous Array Conclusion

We introduce z-rays, a new time-efficient and flexible design of discontiguous arrays. Z-rays use a spine with indirection pointers to fixed-sized arraylets, and five tunable optimizations: a novel first-*N* optimization, lazy allocation, zero compression, fast array copy, and copy-on-write. This paper introduces inlining the first *N* bytes of the array into the spine so that they can be directly accessed, greatly contributing to efficient z-ray performance. We show that fast array copy, lazy allocation, and zero compression each help reduce discontiguous array overhead significantly. Our space savings optimizations, including the novel copy-on-write optimization, reduce the heap size on average by 6%. The experimental results show that z-rays perform within 12.7% on average compared with contiguous arrays on 19 Java benchmarks. Z-rays decrease the overhead as compared to previous discontiguous designs by a factor of two to three. We perform a microbenchmark study that indicates strip-mining and hoisting invariant indirection references out of loops could reduce overhead further for sequentially accessed arrays. Previous work uses arraylets to meet space and predictability demands of real-time and embedded systems, but suffers high overheads. Z-rays bridge this performance gap with an efficient, configurable, and flexible array optimization framework.

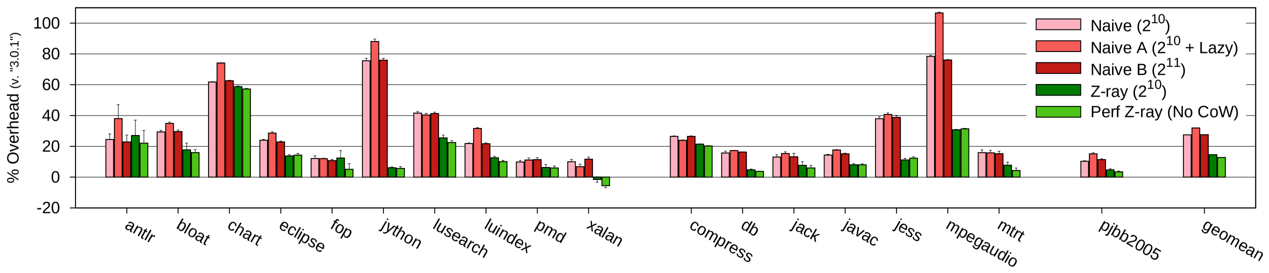


Figure 4.4: Percentage overhead of Z-ray and Perf Z-ray configurations over a JVM with contiguous arrays, compared to previous optimizations.

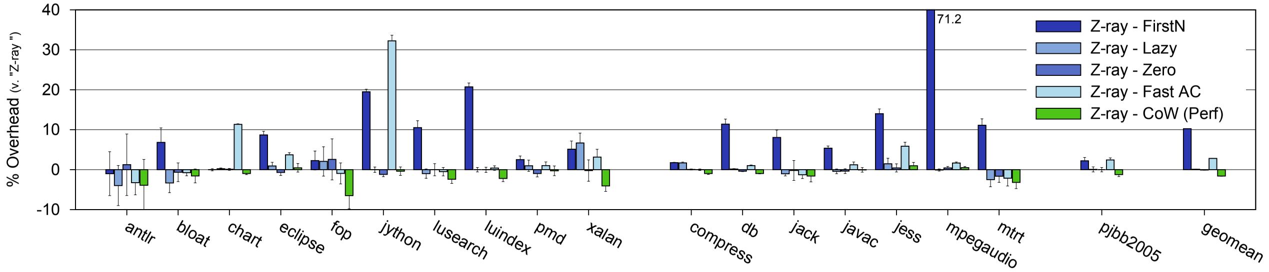


Figure 4.5: Overhead taking away each optimization from Z-ray configuration.

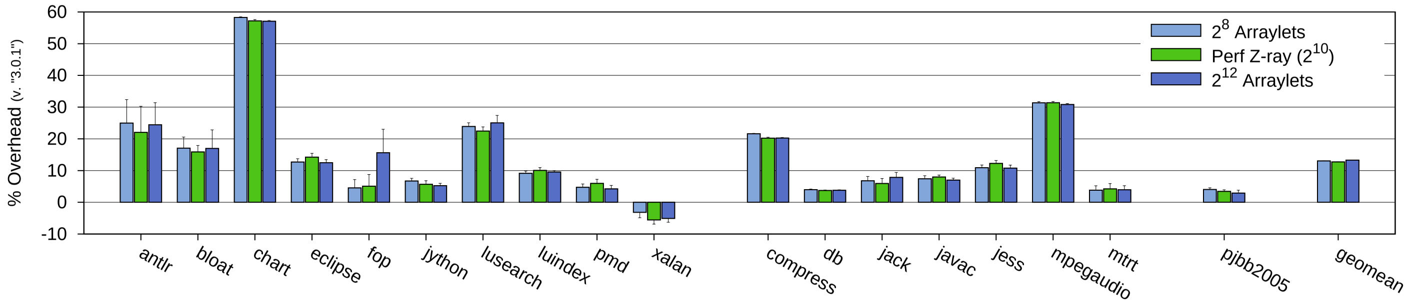


Figure 4.6: Overhead of Perf Z-ray, varying number of arraylet bytes.

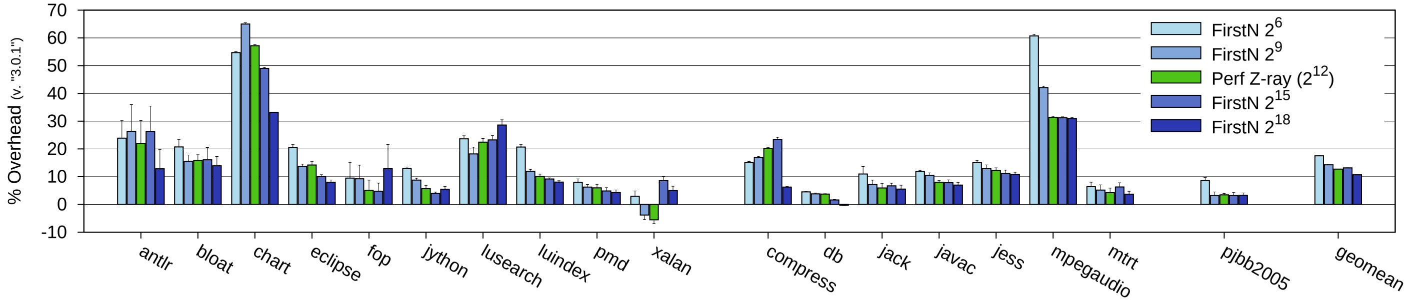


Figure 4.7: Overhead of Perf Z-ray, varying number of inlined first *N* bytes.

Chapter 5

Fragmentation

Fragmentation is an important consideration for space efficiency because it helps to define worst-case bounds on space usage, which is critical for real-time predictability. Fragmentation is defined as memory that will be wasted because it is not available for arbitrary allocation. Prior work notes that quantifying fragmentation is 'problematic' [7], because it is a function not only of live data at a given point in time, but also of what memory can and will be used next by the application. Because garbage collection is periodic, there is only a precise measurement of live data and fragmentation after a whole heap collection, whereas in languages with explicit memory management, such as C, fragmentation can be measured instantaneously on every allocation. Because our overall goal is space efficiency, in this chapter, we explore how our reorganization of arrays changes heap fragmentation. First we offer a qualitative discussion of how fragmentation is changed with our z-ray implementation. Then we present how to measure fragmentation, both at its worst-case limit and in practice.

5.1 Z-ray's Qualitative Fragmentation

Discontiguous arrays in general have benefits for fragmentation, which are well understood in the literature [7, 57]. Fragmentation is in part a function of the largest object size. With contiguous arrays, the largest object is bounded by size of the largest array. With discontiguous arrays it is bounded by the largest spine or arraylet. By reducing the size of the largest object, discontiguous arrays increase the likelihood of finding a chunk of memory large enough to satisfy an allocation request, and hence the system is less likely to suffer from fragmentation and premature out-of-memory errors. The literature also discusses fragmentation ramifications on generational mark-sweep heaps which we use in our experiments [7, 12, 60].

Our z-ray implementation's arraylet space and first-*N* optimization affect fragmentation differently than previous implementations.

Effect of arraylet space. All arraylets are fixed-size, thus, there is no internal fragmentation within the arraylet space, because the allocator can fill any open slot for any arraylet allocation request. The arraylet space eliminates the need for the 'large object space' (discussed in Section 2 and 4.2.2), which is otherwise common in garbage-collected systems. The arraylets may be spread over multiple blocks, inducing external fragmentation if many arraylets are allocated and then many die. However, because arraylets are fixed-sized and accessed via indirection, the collector can compact them if needed. There might be external fragmentation between different heap spaces, but our page manager prevents this case by returning whole free pages to a global pool.

Effect of first-*N* optimization. The first-*N* optimization increases the maximum object size compared to naive discontiguous arrays, because the inlined first-*N* elements increase the spine size. We did not observe any problems caused by the larger spine size, but if it is a concern, the system can disable the optimization or reduce *N*. Our set of optimizations offer flexibility, because the developer can tune them to trade between overhead and fragmentation bounds.

To summarize, one of the primary motivations for discontiguous arrays is that they can help control memory fragmentation by bounding the largest unit of allocation. Z-rays include these benefits, although the first-N optimization has the effect of increasing the bound on the largest unit of allocation by N.

5.2 Measuring Fragmentation

To perform a theoretical study of measuring fragmentation for different array layouts, we first have to define terms and set-up specific to our memory organization in the JVM. In this context, we then describe two types of fragmentation: internal and external, and equations to measure them in Sections 5.2.2 and 5.2.3. Section 5.2.4 explores the theoretical maximal bounds of fragmentation caused by our base JVM versus our z-ray implementation, showing we improve maximal wasted space. Because there is usually a large gap between theoretical maximums and experimentally measured fragmentation, we will describe how we could use the internal and external equations to measure actual fragmentation in our systems in Section 5.2.5. We have demonstrated in Section 4.4.2 that our z-rays can save heap space. We quantitatively measure external fragmentation for our benchmarks with contiguous arrays versus z-rays, showing a great improvement in practice as well.

5.2.1 JVM-specific Details

We would like to compare maximal fragmentation of both our base JVM with contiguous arrays and our modified JVM with the discontiguous array layout. In order to precisely compare fragmentation for the purposes of this study, we will modify our heap organization slightly. Since we only change the layout of large arrays, array sizes greater than first-N, we will not consider here the previously-studied fragmentation of the rest of the heap, which includes small arrays and other objects. To accurately compare just the changed parts of the heap, we have to look at the affected heap spaces in both the base and modified systems. In the base system, large arrays are put into a large object space (LOS) if they are greater than a $LOS\_THRESHOLD$, which is 8KB in our system. In our modified system, we will set N to be equal to $LOS\_THRESHOLD$ so that the definition of "large" is consistent for comparison. In our discontiguous array implementation, we alleviate the need for the LOS, instead putting large array spines in the generational spaces and the arraylet space. For the purposes of apples-to-apples comparison of fragmentation, we instead will isolate large arrays in our discontiguous JVM. We will consider spines as part of the arraylet space so that the only heap spaces that need to be compared for this study are the LOS and arraylet spaces. This isolation changes the property that memory is divided into fixed-sized chunks in the arraylet space, but facilitates direct measurement of differences in fragmentation between the two systems.

To understand the effect on fragmentation when we change the layout of arrays, we need to know precisely how arrays are allocated in each system. The base system's large array space allocates on the granularity of pages, requesting a number of contiguous pages that are a multiple of $PAGE\_BYTES$. Our discontiguous array space allocates on the granularity of a *block* which is $BLOCK\_BYTES$ in size, which is set to the size of an arraylet, or $ARRAYLET\_BYTES$ [1]. With our new organization that includes variable-sized spines in the arraylet space, there could be some block-level fragmentation. We will describe two different kinds of fragmentation: internal to the granularity at which we allocate (blocks or pages) and external, which includes wasted space over all of virtual memory. We will then discuss the theoretical maximal bounds of both kinds of fragmentation, as well as

---

[1] Our arraylet space allocator takes ideas from the Immix allocator that divides space into coarse-grained blocks and fine-grained lines [18]

how to measure them in practice.

5.2.2 Internal Fragmentation

The first kind of fragmentation is internal and is based on our heap's *space budget*. Conceptually this fragmentation is the amount of pages or blocks, depending on the granularity, requested by the memory manager's allocator versus the actual number of bytes taken up by live application objects. This fragmentation is internal in that it emcompasses unusable space inside our allocation granularity not taken up by live objects[2].

To formally define an equation to calculate internal fragmentation, we first need to define some terms. To abstract the allocation granularity away, let us say that the allocator allocates chunks of memory of size $GRANULAR\_BYTES$. Therefore, for an allocator requesting contiguous pages, $GRANULAR\_BYTES = PAGE\_BYTES$, and for a block allocator, $GRANULAR\_BYTES = BLOCK\_BYTES$. We must also define liveness as *Live(obj)* = 1 if the object is considered live, 0 otherwise. We will say *Data(obj)* is the number of bytes in object *obj*.

We can now define a quantitative internal fragmentation equation to be

$$F_{INTERNAL} = \sum_{obj \in space} ((M \times GRANULAR\_BYTES) - Data(obj)) * Live(obj)$$

where M above is the number of allocation-units requested to allocate *obj* which satisfies $(M \times GRANULAR\_BYTES) > Data(obj)$.

This practical allocation-unit-level fragmentation measures unusable memory at the end of allocated objects. If a live object encompasses a whole or an even

---

[2] We were inspired by the instantaneous fragmentation equations presented by Bacon et al. for the Metronome collector [7]

multiple of an allocation-unit, there is no fragmentation, and if an allocation-unit has no live objects on it, it can be reused and is not considered fragmented.

Given that $F_{INTERNAL}$ is measure of instantaneous fragmentation in bytes, we can measure fragmentation as a percentage by dividing this amount by the total live allocated memory in the space. We define the total memory in the space as the sum of bytes on all live allocation-units. First, we define a live allocation unit (m_i) as

$Live(m_i)$ = 1 if $(\exists\ obj \in m_i\ s.t.\ Live(obj) == 1)$, 0 otherwise.

We now define the total space memory in bytes to be

$$space\_bytes = \sum_{i=1}^{TOTAL\_M} GRANULAR\_BYTES * Live(m_i)$$

Therefore, the percentage of internal fragmentation in a space is

$$f_{INTERNAL} = \frac{F_{INTERNAL}}{space\_bytes}$$

5.2.3 External Fragmentation

External fragmentation occurs at the granularity of all of virtual memory. External fragmentation occurs when live, non-moving objects are spread out, pinning down memory and preventing future allocation. There could be enough space for a particular allocation, yet allocation fails because there is not a contiguous chunk that is large enough to satisfy it. This is a problem when virtual memory is constrained, as often is the case in embedded systems. If we run all applications on a 64-bit machine, virtual memory is not constrained and there is no need to worry about this kind of fragmentation. However, since we are studying space efficiency, we are going to assume the common 32-bit system where external fragmentation does exist.

Consider an example where we assume the allocator granularity is a page.

We have 16 contiguous pages of virtual memory. Our largest object size takes 9 pages, and say the smallest object takes a page. If the smallest object is allocated on the 9th page of our 16-page virtual memory (an 8-page object was allocated before it and then freed), there are 8 free pages before it, and 7 after. However, if we then try to allocate one of our largest objects, though we have enough virtual space for it (8+7 = 15 free pages), there are not 9 contiguous pages that can satisfy this allocation. The heap is fragmented and the allocator must either move objects around or request more virtual memory to satisfy this large object allocation.

External fragmentation is worst when memory is pinned down by small objects, that prevent allocation of the largest object. Let's say OBJ_MIN_BYTES is the minimum size of an object in bytes that can go into a space. Similarly, OBJ_MAX_BYTES is the size of the largest object in bytes. When there is a small object allocated in virtual memory every $OBJ\_MAX\_BYTES - GRANULAR\_BYTES$ bytes, there will be no room for the maximum-sized object to be allocated. Thus, virtual memory utilization will be OBJ_MIN_BYTES out of every $OBJ\_MAX\_BYTES - GRANULAR\_BYTES + OBJ\_MIN\_BYTES$. Similarly, fragmentation will be the unused amount of memory, $OBJ\_MAX\_BYTES - GRANULAR\_BYTES$, in $OBJ\_MAX\_BYTES - GRANULAR\_BYTES + OBJ\_MIN\_BYTES$ of memory. Therefore we define a quantitative external fragmentation equation as

$$f_{EXTERNAL} = \frac{OBJ\_MAX\_BYTES - GRANULAR\_BYTES}{OBJ\_MAX\_BYTES - GRANULAR\_BYTES + OBJ\_MIN\_BYTES}$$

5.2.4 Theoretical Maximum Fragmentation

We would like to understand the effect on the theoretical upper bound of both internal and external fragmentation when we change the layout of arrays. For the theoretical worst-case limit of fragmentation, we assume the most adversarial program or allocation sequence.

We see that though our base JVM allocates large arrays at the granularity of a page and our discontiguous array implementation allocations by blocks, their internal fragmentations are of the same order. The worst case internal fragmentation would occur when objects are just larger than a multiple of the allocation-unit. The most adversarial program would thus waste $GRANULAR\_BYTES - 1$ bytes per allocated large array. For the arraylet space, an adversarial case would create a spine for each array allocated that is just one byte over a multiple of the block size, and each array would have remainder elements, but no fixed-sized arraylets, because they improve fragmentation. Therefore the theoretical maximal internal fragmentation for both contiguous and discontiguous arrays is $O(k \times GRANULAR\_BYTES)$ where k is the number of large arrays allocated. Although internal fragmentation is on the same order for both contiguous and discontiguous layouts, in our configurations $GRANULAR\_BYTES$ are set to different values. Our contiguous layout sets $GRANULAR\_BYTES$ to $PAGE\_BYTES$ or 4KB, while our Z-ray configuration sets $GRANULAR\_BYTES$ to $BLOCK\_BYTES$ or $ARRAYLET\_BYTES$ which is 1KB. Since our large object space and arraylet space have the same number of large arrays, k, the contiguous internal fragmentation is $O(4k)$ in comparison with z-rays $O(k)$. Only a constant factor is different, and thus internal fragmentation, while expected to be reduced in practice, is on the same order for both array layouts. We see that overall our dividing up of arrays does not reduce worst-case internal fragmentation.

External fragmentation depends entirely on the minimum and maximum object sizes allowed in our spaces. In Java, there is no upper bound on the size of objects that can be allocated, but based on an adversarial program and our setup assumptions, the largest object is an array with $Integer.MAX\_VALUE$ ($2^{31} - 1$) elements. We define OBJ_MAX_BYTES to be equal to the largest contiguous chunk of memory in Java that can be allocated, and *OBJ_MIN_BYTES* to be the smallest. For Jikes with contiguous arrays, in our system set up to evaluate fragmentation, *OBJ_MIN_BYTES* is equal to our *LOS_THRESHOLD*, and *OBJ_MAX_BYTES* is close to 8GB, assuming 4 byte array elements. As mentioned above, *GRANULAR_BYTES* for the large object space is *PAGE_BYTES* or 4KB. For discontiguous arrays, the minimum object size stays the same, as we have set up to study only large arrays in these two contexts. However, *OBJ_MAX_BYTES* is reduced to the number of bytes required by the spine of the largest possible array which is around 8MB. Similarly, *GRANULAR_BYTES* is the size of an arraylet, or 1KB as mentioned above.

Therefore, the worst-case external fragmentation for the base JVM, using our equation from Section 5.2.3, is O((8GB - 4KB) / (8GB - 4KB + 8KB)) or (8589930492 / 8589938684) which is 0.999999046, about 1. Whereas, our arraylet space has worst-case external fragmentation on the order of O ((8MB - 1KB) / (8MB - 1KB + 8KB)) or (8387584 / 8395776) which is around 0.9990242. Although this amount still approaches 1 in the limit, by making large arrays discontiguous we have reduced *OBJ_MAX_BYTES* by 1000 times and *GRANULAR_BYTES* by 4 times, making these two extremes closer together. This reduction improves the chance that the memory manager can satisfy an allocation request and lowers the likelihood of premature out-of-bounds errors. Thus, our discontiguous array layout slightly reduces theoretical maximal external fragmentation. Although this worst-case bound looks very high, in practice JVMs do not incur this large amount of fragmentation. Because the memory implementation is abstracted away from the programmer, the collector can move objects if virtual memory fragmentation severely degrades performance, whereas native languages cannot move objects to reduce fragmentation.

5.2.5 Fragmentation in Practice

The above section investigates what the maximal bounds of fragmentation are given the most adversarial allocation sequences. Real programs rarely reach fragmentation equal to their theoretical maximums. Thus, in this section we explore measuring the actual fragmentation at particular points in program execution which we expect to be much lower. Since external fragmentation has a larger effect on program predictability and can cause dire premature out-of-memory errors, we use these equations to quantify external fragmentation in practice for our benchmarks. We compare the large-object space layout to our modified arraylet space design, showing that we reduce practical fragmentation as well as theoretical.

Practical fragmentation is roughly the total amount of memory allocated over the total memory actually in use by the application at a particular point in time, measuring the wasted space. There is some question as to when practical fragmentation in the heap should be measured. Optimally, we could measure fragmentation after every object allocation, or after every large array allocation since this is what we are analyzing here. However, this measurement is not necessarily tractable, thus we propose measuring fragmentation right before and after full heap garbage collections that trace all large objects. We believe heap fragmentation could be poor right before a collection, as collection is triggered by an unsatisfiable allocation. However, our instantaneous fragmentation equations require knowledge of the exact number of live bytes in the heap, which is known right after a garbage collection. To calculate object liveness right before a collection, we can take the previous collection's live bytes and add all bytes allocated in our spaces in between the collections, regardless of whether the subsequent collection will count them as live. We ensure that collections occur at precisely the same time in our base JVM and our optimized JVM by forcing collections at even intervals. Therefore we can measure internal fragmentation at representative program points, before and after collection, to compare the efficiency of contiguous versus discontiguous arrays.

We can use the equations for internal fragmentation defined in Section 5.2.2 to measure fragmentation in practice for our benchmarks. Notice that $Live(obj)$ is defined by whether the object is *considered* live, which will change with our fragmentation measurements before and after a garbage collection. Although the order of internal fragmentation for the large object space and arraylet space are the same, the granularity in our implementation is different and we expect more heap fragmentation for contiguous arrays. In our discontiguous system, arraylets will still take up exactly a block and do not cause block-level internal fragmentation. While spines will create fragmentation, there should in general be fewer spines than arraylets for large arrays, and we expect spines to be less adversarial than the worst case.

To measure external fragmentation defined in Section 5.2.3 in practice, we have to empirically find OBJ_MIN_BYTES and OBJ_MAX_BYTES for our benchmarks. Our contiguous array space in our base Jikes and our z-ray implementation both already define OBJ_MIN_BYTES statically. What actually determines external fragmentation is OBJ_MAX_BYTES, or the large object size over the course of a program run. Below, Table 5.1 shows the byte-size of largest array allocated for each of our DaCapo benchmarks. These numbers were collected using the same experimental framework as in Section 4.3.3, using Jikes RVM 3.0.1 with a GenMS heap. Table 5.1 shows not only the size in bytes for the contiguous representation of the largest array, but also the corresponding size in bytes for the spine, the largest contiguous chunk, using a z-ray layout. The chart benchmark has the largest array of all benchmarks, 4MB. This 4MB array is broken up with z-rays, the largest contiguous chunk being the spine which is only about 20KB, a reduction of 205

Benchmark	Max Contiguous Bytes	Max Z-ray Bytes	Reduction in External Fragmentation
antlr	296640	5940	50 times
bloat	296640	5940	50 times
chart	4194304	20464	205 times
eclipse	593280	6780	88 times
fop	296640	5940	50 times
hsqldb	296640	5940	50 times
jython	296640	5940	50 times
luindex	296640	5940	50 times
lusearch	296640	5940	50 times
pmd	786428	8168	96 times
xalan	296640	5940	50 times

Table 5.1: Largest allocated array per benchmark, with corresponding size of contiguous Z-ray bytes, and the reduced external fragmentation between them.

times. Hence, although the theoretical worst-case external fragmentation was on the same order, we see between 50 and 205 times reduction in external fragmentation in practice for our benchmarks. This great reduction in our discontiguous array design leads to more efficient space usage.

5.3 Fragmentation Conclusion

We have offered a qualitative discussion of the changes to fragmentation incurred as we move from a contiguous array layout to our z-ray layout, taking into account previous work's research on discontiguous array fragmentation. We then detailed slight changes to our memory management space organization to quantitatively study apples-to-apples fragmentation of both our base and modified JVMs. We presented definitions and equations to measure both internal and external fragmentation. We have compared the theoretical maximal fragmentation with our two array layouts, one using the large object space and one dividing up arrays in the arraylet space. Although internal fragmentation is on the same order for both designs, we have shown that external fragmentation is reduced by 1000 times because we reduced the largest object size but still approaches one in the arraylet space. We have also described how to evaluate actual fragmentation at particular points during program runs for both the original and our z-ray designs. This quantifies the amount of wasted heap space with internal fragmentation at various intervals, whereas external fragmentation requires finding the largest chunk of memory allocated over a whole run. We performed an experiment to measure the reduction in external fragmentation, which is a function of the largest object, with our z-ray layout, showing that in practice we reduce the fragmentation by large orders of magnitude. This thesis has thus studied in depth the changes to heap fragmentation as we move arrays from a contiguous layout to a discontiguous one, making memory not just time, but also space efficient.

Chapter 6

Saving Memory Traffic and Bandwidth

This chapter investigates memory inefficiencies at the hardware level, within the cache memory hierarchy. Chapter 2 showed that 88% of our benchmarks' write backs to main memory from the lowest level cache are useless, i.e., the data is never read again by the program. This problem is partially due to the rapid allocation of short-lived data in managed languages that pollutes the caches. This chapter examines how to get rid of this useless traffic in the face of emerging hardware memory scalability issues to improve performance into the future. This chapter describes how to harness the abstraction of memory management at the software level to inform hardware of dead regions of data to optimize traffic, bandwidth, and application performance.

We take advantage of the heap organization of managed languages to pass down to hardware a range of memory that is used for rapid allocation and then dies en masse. We call this range of memory addresses the *candidate region*. We design three software-hardware optimizations that exploit this information. 1) The *invalidation* optimization invalidates resident cache lines that are in the candidate region passed down by software. This optimization improves cache replacement by eagerly evicting dead lines and avoiding write-backs of dead, dirty lines. 2) We design a uniprocessor version of *in-cache zeroing* of data at a cache line granularity in the candidate region when it is first accessed and live to avoid fetches from memory. Finally, 3) we perform *priority biasing* of lines within the cache set of the candidate region, changing their placement so that short-lived data is evicted faster. This approach attempts to reduce cache pollution, but is not as effective as invalidation. Our optimizations require modest hardware support in the form of an extra byte per cache line and a few control registers. These optimizations tolerate the rapid allocation of short-lived data by reducing cache displacement and reducing read and write traffic to main memory, thereby improving application performance.

6.1 Cooperative Invalidation

This section explains the design of cooperative software-hardware invalidation. We focus on the nursery in a generational collector and assume that the VM uses the standard contiguous region of memory for the nursery. The software is responsible for communicating to the hardware the *candidate region* and when the candidate region is invalid. To exploit this information, the hardware must keep track of the upper and lower bound of the region. To eliminate useless write-backs or perform priority-biasing, the hardware must track cache-resident lines in the region. For in-cache zeroing, the hardware keeps track of the high-water mark in the candidate region. Lines below the mark are valid, and lines above the mark are invalid, but will be initialized to zero in-cache upon initial access. The software communicates periodically to hardware—at least at the beginning of the program and after collections that invalidate the range. The hardware modifications operate in response to the software and on cache accesses. The next sections describe the software and hardware responsibilities for each optimization in more detail.

6.1.1 Software Responsibilities

The software periodically communicates with the hardware. At the conclusion of each nursery collection, the garbage collector communicates that the region is invalid. We use a stop-the-world generational collector and a synchronous invalidation operation. With synchronous invalidation, the write-back optimization requires only a lower and upper region bound. *Asynchronous* invalidation would reduce latency and is necessary to support concurrent collector, but adds some hardware complexity. Asynchronous invalidation requires tracking the allocation point in the region in a third control register. As the program allocates, it writes valid objects in cache lines. The hardware must track the high-water mark of initializing writes into the candidate region to keep from invalidating them with asynchronous invalidation. The only required JVM changes are an extra call from software to hardware to communicate the initial candidate region and the invalidation of that region at the conclusion of each nursery collection.

As discussed in 1.2, our optimization framework, while focused here on the nursery region, is applicable to broader contexts and other heap organizations. We can use the same infrastructure to communicate candidate regions that are identified as dead by full-heap garbage collectors or region allocators. Instead of passing one large address range to hardware, we could identify several discontiguous regions of dead data, more similar to the design in ESKIMO [42], discussed below in related work in Section 6.3. Furthermore, to pass information about mature data to hardware right when it dies so it is more likely to be in cache and be invalidated, we could use a generational heap with a reference-counted mature space. Because reference counted heap spaces identify data as dead more immediately than traced spaces, we could retain the benefits of frequent invalidation and cache-resident dead data to reduce traffic effectively.

6.1.2 Hardware Modifications

This section details the hardware modifications that are necessary for our software-hardware cooperative optimizations, shown in Figure 6.1. There are five main hardware modifications. First, we require two extra control registers, the *invalidation*

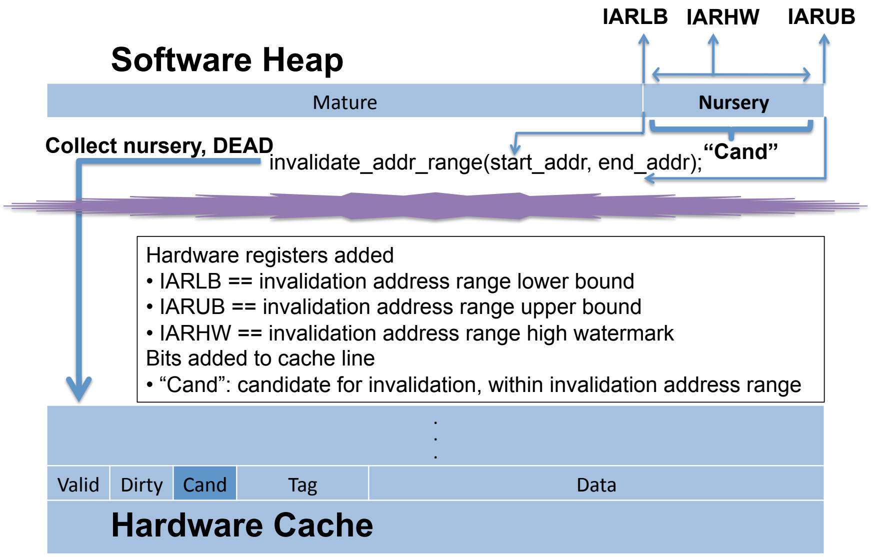


Figure 6.1: Software heap layout collaborating with hardware caches including hardware modifications.

address region lower bound (IARLB) and the *invalidation address region upper bound* (IARUB). Second, we add a byte to each cache line called the *candidate byte*. If the candidate byte is non-zero it indicates that this line is within a candidate region. When the byte is non-zero, it identifies the process owning the line. The byte is zero by default. Third, we require logic which sets this byte when a new cache line is installed if the memory access address is between the IARLB and IARUB. Fourth, we add an extra candidate byte to each miss status holding register or MSHR, to indicate whether the missed cache line holds data within the candidate region. Lastly, when we receive the invalidation signal from software, we modify the invalidation logic to zero each cache line's valid bit if its candidate byte is set for this process.

The software will first communicate the candidate region, so that hardware can set the IARLB to the nursery starting address, and the IARUB to the ending address.

Comparison logic is responsible for checking whether each cache line holds the data from within the candidate region when the line is instantiated. Although the candidate region is contiguous in virtual address space, it might not be the case in the physical address space. We choose to compare the access address with the IARLB and IARUB within the virtual address space to simplify the comparison. The comparison is done in parallel with the TLB access to generate the candidate byte which stores the process identification. If the access hits the L1 cache, the candidate byte will be discarded. On the other hand, if the access misses L1, the candidate byte will be attached to corresponding MSHR entry. If the cache line is in another level of cache, when the data is fetched back, the candidate byte value within MSHR entry will be put into the L1 cache line. If the access misses the whole cache hierarchy, when the data comes back from main memory, the candidate byte of the corresponding cache line at each cache level needs to be updated with the candidate byte stored in the MSHR entry at each level. This design limits the number of processes that can simultaneously use the invalidation optimization to 256. This design also requires that the candidate region upper and lower bounds be cache line aligned. Both are reasonable requirements; the latter is common in modern JVMs. Because the operating system maintains a limited list of handles that are given out to processes, we can use this list and disable our optimization if the process has not acquired one of these handles.

The lower half of Figure 6.1 shows modifications to the cache memory hierarchy including the control registers we add. We also show the added candidate byte per cache line as the darkened box in Figure 6.1.

We estimate that this hardware support adds less than 1.5% to the area of the last level cache. It uses three control registers, one byte per cache line, and logic to compare virtual addresses to control registers. The most area-intensive part is the byte per cache line. The logic of invalidating many candidate cache lines when we receive the software signal can be done efficiently in parallel by hardware, comparing the invalidation process' ID with the candidate byte. Although the operation is expensive, it is only called once per garbage collection and its cost is minute compared to a garbage collection cycle. As we show below, these minimal changes allow us to save a lot of memory bandwidth.

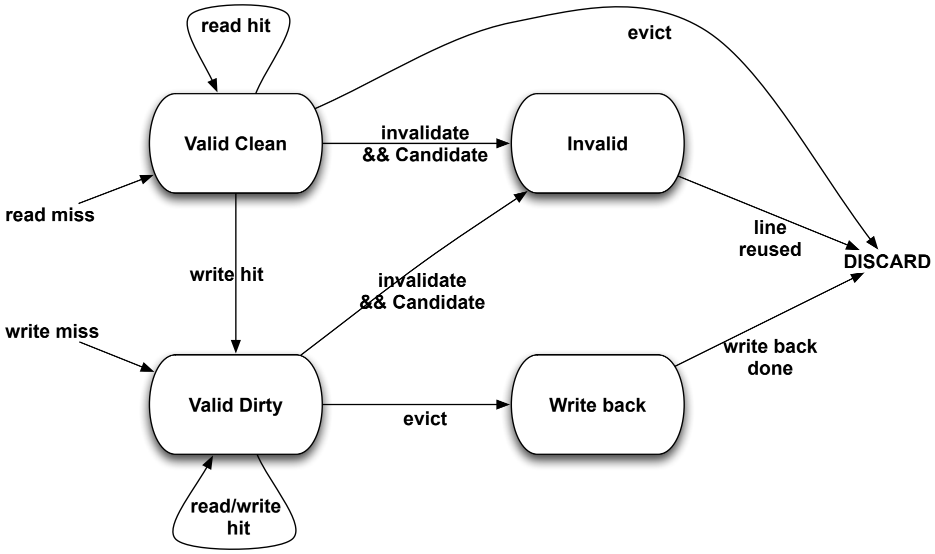


Figure 6.2: Diagram of cache line states.

6.1.3 Eliminating useless write-backs with invalidation

The software for this optimization first communicates the candidate region, which is initially invalid. The application then executes as usual. Figure 6.2 shows a diagram of the cache line states during program execution. When the hardware processes a read or write cache miss during application execution, it sets the candidate byte if the virtual address is between the lower and upper bound. If the miss is a read, the line is clean and valid. If it is a write, it is dirty and valid. We assume that the entire line is fetched on a write miss, i.e., a *write allocate* policy [45]. At the conclusion of a garbage collection, the software makes the invalidate region down call, indicating all objects and data on the cache lines in the region are dead. The hardware invalidates all cache lines with the matching candidate byte in parallel, shown as the transition to the “Invalid” state in Figure 6.2. On subsequent misses, the cache replaces invalid lines first, without writing back their contents, as usual for invalid cache lines. Figure 6.2’s transition from the “Invalid” state to the “DISCARD” state shows lines that we evict with our optimization. Invalidation wins twice—it improves cache replacement and it eliminates useless write-backs.

6.1.4 In-cache zeroing

Current hardware and prior research use special cache-line zero instructions to avoid line fetches [42, 45, 49, 68]. For example, the PowerPC data cache block zero (dcbz) instruction was created such that software can efficiently zero a full cache line without reading it from memory [68]. In-cache zeroing differs from these approaches because it also *eliminates the zeroing instructions, zeroing logic, and their cache effects from the software*.

Java semantics and many other programming languages require zero initialization of all fresh allocation. Implementations of these semantics can choose between zeroing at the point of object allocation or eager zeroing allocation regions en masse. Although zeroing at allocation time is better for locality since the application is about to use this memory anyway, it adds complexity to the hot path of allocation. Because this code is inserted at every allocation point, it bloats the application code. Because of this bloat and the complexity of zeroing variable sized objects, no production or research VMs of which we are aware use this approach. Instead, they all choose to perform en masse zeroing. For example, Jikes RVM zeros 32 KB chunks of the nursery at a time as it acquires chunks from a global resource pool. Jikes RVM uses a highly optimized zeroing sequence that works well with hardware prefetching. Although this approach performs better than allocation-time zeroing, it does touch all 32 KB of data quickly. This zeroing marches through the lowest level cache, and these lines are unlikely to remain resident. In-cache zeroing eliminates these instructions altogether from the JVM.

Here we explain the design of in-cache zeroing in a uniprocessor setting. We have not yet fully designed an efficient in-cache zeroing optimization for the multiprocessor setting. To implement in-cache zeroing, the JVM requires the same changes as cooperative invalidation–it must communicate the candidate region to the hardware and this region must be initially invalid. The cache hardware requires a extra control register to track the *invalidation address region high-water mark* (IARHW). The high-water mark is the upper bound of valid data in the candidate region. Initially, the hardware sets the IARHW to the lower bound of the invalidation region (IARLB) when the JVM sets the candidate region. The cache hardware then tracks all writes to the region. If the virtual address of a write is in the range and it exceeds the current high-water mark, in-cache zeroing initializes all cache lines between the current mark and the write virtual address. It then sets the high-water mark to this address. It thus instantiates these lines in cache just prior to their use. This optimization eliminates all writes, reads, and fetches for zero initialization. It furthermore eliminates all the software instructions that perform them and their cache effects.

6.1.5 Priority biasing

We explore a *priority biasing* approach, which is a less invasive approach than invalidation, but also less powerful. Priority biasing targets only cache replacement decisions in the candidate region. This approach seeks to minimize the cache displacement effect of the nursery by evicting this data sooner than usual. For addresses in the candidate range, priority biasing separately modifies the initial miss placement bias and the hit placement bias in the recency queue that implements the least-recently-used set-associativity replacement algorithm. We experiment with fractional bias values based on the size of the recency queue, i.e., k in a k-way set-associative cache. The default miss and hit placement bias is one, the MRU position. For example, a bias of 1/2, sets the recency to k/2, the middle of the queue. We experiment with a variety of bias values to find if biasing nursery lines closer to the LRU position will evict them faster and reduce the nursery's cache displacement effect. This approach only requires two control registers. Only policies that changes placement bias on cache hit requires additional cache line bits. However, the results show that priority biasing is not consistently effective and is much less effective than invalidation.

6.1.6 Invalidation Design Options

The design of a software-hardware cooperative invalidation optimization can be performed at many different granularities, which tradeoff the invalidation work between software and hardware. The design presented in Section 6.1.3 has software invalidating at the granularity of regions, putting more work on hardware to maintain information about what can be invalidated. This section offers alternative designs for invalidation, describing a spectrum of granularities and how they trade off work in the system. We sketch possible designs, but do not build implementations.

Our current software-hardware cooperative design invalidates *regions* of memory at a time. This design leverages software's memory organization including software's short-lived nursery that is frequently identified as dead. Software communicates once every time a collection identifies a dead region. This design puts the burden on hardware to know which cache-resident lines are in this region and can be invalidated. Because the region passed down is a virtual address range, we require the candidate byte per cache line to do atomic invalidation in parallel cheaply in physically-indexed low-level caches. Hardware modifications are necessary in order to do invalidation in parallel; however, new hardware support is not required with options that loop over the invalidation region to check which addresses are resident in cache, as we shall see.

We can also invalidate at the granularity of a *cache line*, the cache hierarchy's organization unit. For example, the PowerPC architecture has a cache line invalidate instruction (*cli*) [39] that invalidates a line in cache so that the next access requires fetching from main memory. Because *cli* is a privileged instruction, we require operating system (OS) support to use it for invalidation. The JVM could pass one large dead region down to the OS with a system call, and the kernel could loop over the virtual address range, calling the *cli* instruction once per aligned cache line in the region. This software approach puts the burden on the OS to loop over addresses to perform invalidation, requiring no hardware modifications when the ISA contains this instruction. Cache line invalidation, however, does introduce more displacement in the instruction cache (i-cache) and cannot efficiently invalidate in parallel as hardware can. We believe this cache-line granularity invalidation design would not displace the i-cache much. If desired, this design could move the loop over dead addresses to hardware to achieve atomicity with a simple state machine, reducing the i-cache effects. It is not clear that the hardware modifications justify the modest gains. This approach lifts the burden of invalidation up to software, and may still realize all the benefit of saving traffic and improving application performance.

Another possible cooperative invalidation design is at the *page* granularity. Instead of invalidating regions or cache lines, a middle ground is to invalidate a page at a time. This granularity matches the address translation and protection granularities that exist both in the TLB in hardware, and in the OS. One benefit of this approach is that it requires less frequent software communication than the cache line design, thereby polluting the i-cache less. Another benefit of this approach is that it does not require additional cache line bits since all lines within a page have the same virtual-to-physical address mapping. However, the burden of knowing which lines are candidates for invalidation falls back to the hardware.

There are three ways hardware could keep track of cache lines to invalidate: 1) As mentioned above for the cache line design, hardware could walk through the cache, looping over invalid addresses to check if lines are resident using a finite state machine. 2) Hardware could maintain a bitmap for each page that identifies which lines are in cache. This option is not an obvious option for the region approach, but is facilitated here because all lines within a page have the same virtual-to-physical mapping. 3) Finally, hardware could use content addressable memory (CAM) that keeps a map from pages to cache lines. This option allows efficient parallel cache invalidation by comparing the page identification bits of cache tags to the invalid page passed down by software.

When comparing the three options, the first option takes many cycles upon receiving the software invalidation, the second requires more space overhead, and the third requires that specific expensive hardware already be present. The second option, keeping a bitmap per page, requires 8 bytes per page for a 4KB page size, which adds 8 bytes to each TLB entry. If 4MB superpages are used instead, each TLB entry would require an extra 8KB of storage, which would be hard to justify. In addition to the extra storage, option 2) requires logic to invalidate lines marked as resident. In the absence of hardware with CAM, option 1) seems the most feasible. Although the page-granularity invalidation design offers an appealing middle-ground, it still requires hardware support. However, the hardware additions are more modest than the current design and more in-line with the existing cache and TLB organization, and could also reduce traffic to memory extensively.

In summary, many invalidation designs are possible that work at the region, page and cache line granularities. These designs offer a spectrum of alternatives, varying the amount of work that the language runtime, the operating system, and the memory subsystem have to perform. All of our invalidation designs, while offering implementation flexibility, achieve our objective of software-hardware cooperation to reduce program traffic and bandwidth.

6.2 Evaluation

This section explores the effects of our proposed software-hardware optimizations on the cache hierarchy and application performance. We show that we can eliminate almost half of all write-backs to memory with our useless write-back optimization. This drastic reduction in write-backs translates to less traffic, and improved application performance. We present experiments varying both the L2 size and nursery size, showing the effect on cache miss rates and write-backs. We present initial results of improvements to L2 references and misses in Valgrind for our in-cache zero optimization designed for a uniprocessor system. We present results on our priority biasing experiments on Valgrind, showing that it is difficult to improve the miss rate solely by more eagerly evicting short-lived data.

6.2.1 Methodology

Benchmarks. We perform experiments on the DaCapo benchmark suite, version 2006-10-MR2 [13]. We exclude chart due to its dependence on external libraries that would not work with our simulation setup. For our cycle-accurate simulator, we could only get 7 of the remaining 10 benchmarks successfully running to completion. Benchmarks hsqldb, lusearch, xalan are multi-threaded.

JVM configurations and experimental design. We modified the 3.1.0 release of the Jikes Research Virtual Machine. We use Jikes RVM's best performing garbage collector: a generational heap with an immix mature space [18]. We use a heap size of 2× the minimum required for each individual benchmark, reflecting moderate heap pressure. We use Jikes RVM's *boundedNursery* configuration to set the maximum nursery size in our experiments. The JVM can shrink the nursery size when the mature space is too limited to accommodate all nursery survivors.

As recommended by previous research, we eliminate the non-determinism of the adaptive compiler during the measured application run [34]. The just-in-time compiler performs replay compilation, applying a fixed compilation plan when it first compiles each method [38]. This fixed plan is calculated offline by having the adaptive compiler record the set of sampling and optimization decisions per method in a set of runs that result in the best performance. We run each application twice using the replay compilation technique, measuring the second run to reduce non-determinism and measure steady-state behavior.

Experimental platforms. We use Valgrind 3.5.0 [56], configuring the L1 and L2 cache sizes. Valgrind is a binary re-writer and an instrumentation framework which we modified to evaluate our optimizations. We did most of our implementation modifications to the CacheGrind tool, which models and profiles the cache hierarchy and behavior. We added functionality to receive the software downcall.

For our Valgrind experiments, we compare an unmodified system with our optimized one. Side-by-side with an unmodified cache hierarchy, we simulate in a matching cache hierarchy the effect of the invalidation, in-cache zeroing plus invalidation, and moving candidate lines into the LRU positions. Valgrind does not simulate a "valid" bit per cache line, so moving them to the LRU position has the effect of forcing lines to be evicted next. We count the number of total write-backs that the invalidation optimization eliminates, and the number of misses or fetches we can save with in-cache zeroing. We perform side-by-side comparison instead of comparing cache performance with separate runs to gather more accurate results.

We also include experimental results with the cycle-accurate PTLsim [76] simulator. The default PTLsim supports a write-though cache, does not accurately model the memory controller structure, and has no bandwidth limits. We use version `2009-03-14 svn rev 229` of PTLsim. We briefly detail additions to PTLsim to enable our optimizations and to allow for meaningful evaluation.

We modified the simulator to support a write-back cache. In order to exploit the performance impact with variable bandwidth settings, we model a simple memory controller, which consists of a first-come-first-served queue. Both read and write request transactions are pushed into the queue in order, and are served in-order from the top of the queue.

PTLsim relies on a load-fill request queue to hold information about outstanding loads which miss in the cache. It also tracks all outstanding cache line

	Total Size	Line Size	Associativity
L1 Data	32KB	64 bytes	8-way
L1 Instruction	32KB	64 bytes	8-way
L2	4MB	64 bytes	16-way

Table 6.1: Cache configurations we model.

	Generous	Moderate	Constrained
Bandwidth	1400MB/s	100MB/s	50MB/s

Table 6.2: Evaluated bandwidth settings.

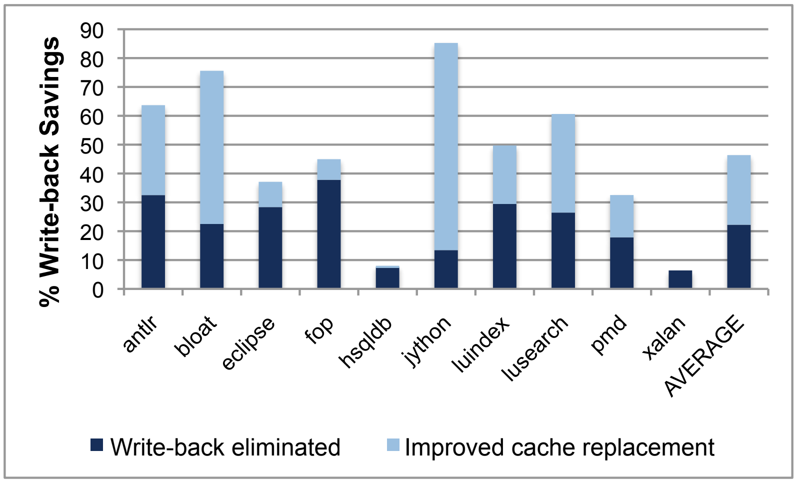
misses in a missing buffer. We add the candidate byte to each entry of the two structures to indicate that the missed cache line consists of data whose virtual address falls into the candidate region. When a load instruction is issued or a store instruction is committed, we add logic that compares the address to update the candidate byte in the load-fill request queue or the missing buffer, respectively. When hardware serves the request and the data comes back from main memory, the cache updates the line's candidate byte.

Cache and memory configuration. We model our cache hierarchy after the machines used in recent related work, both of which use two Intel Xeon Clovertown four-core processors [61, 77]. Although the two previous papers differ on the Clovertown models they use, we chose to model the cache configuration of the E5320. We simulate write-back, allocate caches. Because PTLSim does not simulate hardware parallelism, we simulate using the cache sizes of one core. As shown in Table 6.1, we use a 32KB L1 data cache and a 32KB L1 instruction cache, both with a line size of 64 bytes and 8-way set associativity. Our baseline experimental setup uses a combined 4MB L2 cache with 64-byte line size and 16-way set associativity, however, we vary the L2 capacity in experiments below.

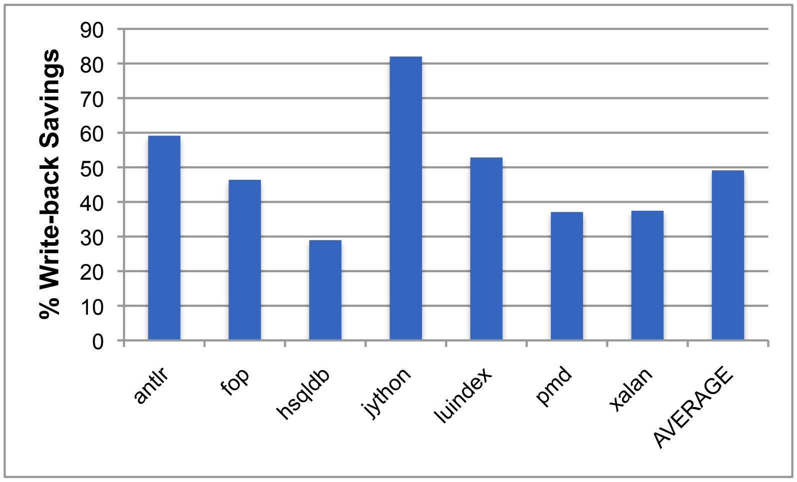
We make a number of changes in light of the inability of our simulation infrastructure to simulate multiple cores. First, we experiment with smaller L2 sizes to compensate for the fact that our L2 is never shared, unlike the L2 cache of the hardware we model. Second, much of the memory bandwidth bottleneck comes from the scaling of cores without commensurate scaling of memory bandwidth [61, 77], so we experiment with memory bandwidth settings ranging from limited to fairly unconstrained. The exploration of constrained bandwidth settings is important since current research shows that this is a problem that will continue to get worse with parallel applications running on CMPs [61, 77]. In some experiments, we throttle bandwidth by reducing the length of the memory queue that models latency and bandwidth. For our benchmarks, we found appropriate *generous, moderate*, and *constrained* bandwidth settings for PTLsim, which are listed in Table 6.2.

6.2.2 Useless Write Backs Eliminated

Figure 6.3 shows for each of our DaCapo benchmarks, the percentage of cache line write-backs that we eliminate due to our useless write-back invalidation optimization. This figure presents results for a 4MB nursery in the JVM and a 4MB L2 cache. Because we model the normal cache side-by-side with the optimized cache in Valgrind, we can break these saved write-backs down into two categories in Figure 6.3(a). The optimized cache has fewer write-backs because, first, it eagerly evicts *all* dead data, not just dirty data, reducing overall cache pollution and improving cache replacement. Secondly, we count the lines in our optimized cache that are evicted and dirty and *would have* resulted in a write-back, but do not because they are dead. For our cycle-accurate simulator, we include total write-back savings without the break-down, for our “Constrained” bandwidth setting in Figure 6.3(b). For PTLsim, write-back savings is fairly consistent across bandwidth



(a) Valgrind



(b) PTLsim

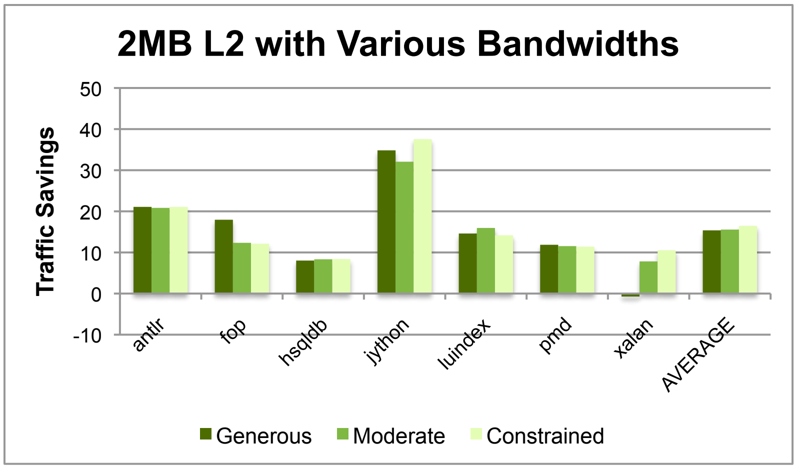
Figure 6.3: Per benchmark percentage of write-backs that we can save for both Valgrind and PTLsim using a 4MB nursery and L2 cache. For Valgrind we are able to divide savings into reducing the cache pollution and not writing back dirty dead data. For PTLsim, we show total reduction in write-backs for our constrained bandwidth setting.

settings. Overall with Valgrind, we can eliminate on average 22% of dirty write-backs, and reduce cache pollution with on average 24% of write-backs, giving us a total savings of 46% of all write-backs. For PTLsim, we achieve similar savings, on average almost *half* of all write-backs are eliminated. Results for our 2MB nursery and L2 cache follow similar trends, but are on average 9-10% less than for the 4MB configuration.

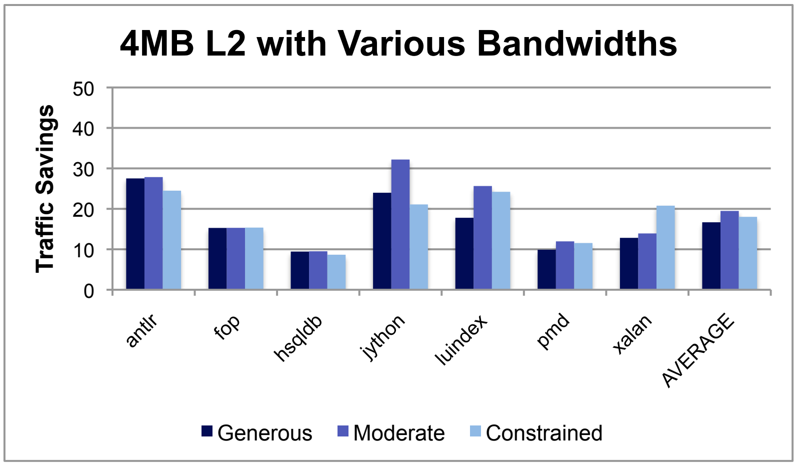
From the essential versus useless study at the word-granularity in Section 2.1, we saw that benchmarks hsqldb and xalan have a larger percentage of mature write-backs, and do not have as much opportunity to benefit from our nursery-focused optimization. Figure 6.3 matches this expectation, showing that hsqldb and xalan can only save 8% and 6% of write-backs for Valgrind and 29% and 37% on PTLsim, respectively. We surmise some of the reason for this difference is the difference in bookkeeping in the two simulators. For example, Valgrind counts only one write-back per cache miss that results in the eviction of dirty data, instead of a write-back per dirty cache line evicted. However, the other benchmarks all save significantly, with jython saving the largest percentage: 85% on Valgrind and 82% on PTLsim. Most of jython's savings comes from reduced cache pollution. However, all other benchmark savings are either divided evenly between reduced cache pollution and eliminated dirty write-backs, or are due to slightly more eliminated write-backs. Four benchmarks, antlr, bloat, jython, and lusearch, particularly benefit from our software-hardware cooperation, all eliminating 59% or more of write backs to memory, significantly reducing traffic. On average, passing the short-lived dead data range can save almost half of write-backs between L2 cache and main memory.

6.2.3 Traffic and Cycle Savings

Figures 6.4 and 6.5 shows for both 2MB and 4MB nursery-L2 size pairings, the improvement in traffic and performance that our invalidation optimization achieves on PTLsim over the unmodified system. For each of our DaCapo benchmarks, we



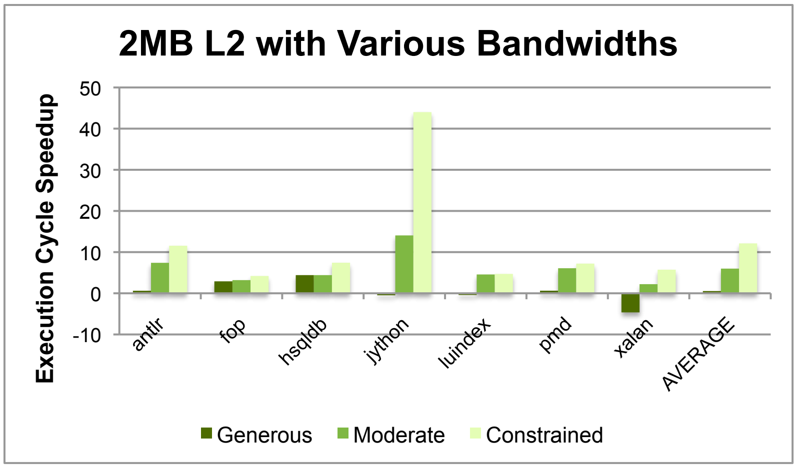
(a) 2MB Traffic



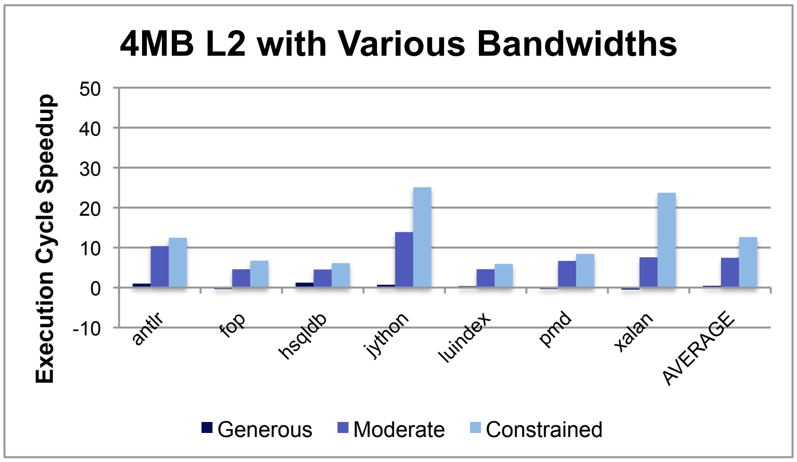
(b) 4MB Traffic

Figure 6.4: Per benchmark savings in read and write traffic for PTLsim. We show improvement for both a 2MB and 4MB nursery and L2 pairings at three bandwidth settings.

show the savings at our three bandwidth settings: generous, moderate, and constrained. Overall, we see larger speedups as we throttle bandwidth more, but traffic reductions are more stable across bandwidth settings. This result is expected because constrained bandwidth limits performance, and write-back savings has a



(a) 2MB Cycles



(b) 4MB Cycles

Figure 6.5: Per benchmark savings in execution cycles for PTLsim. We show improvement for both a 2MB and 4MB nursery and L2 pairings at three bandwidth settings.

larger cycle-count impact in a bandwidth-limited environment. We achieve on average 16 and 18% improvements in traffic at a constrained bandwidth for 2MB and 4MB sizes, respectively, as compared with 15 and 17% traffic reduction at a generous bandwidth, as shown in Figures 6.4(a) and 6.4(b). For execution cycles, only xalan with the generous bandwidth setting degrades performance slightly by 4.6%. Otherwise, we very modestly improve performance at generous bandwidth settings, but in constrained settings improve performance on average 12–13% as shown in Figures 6.5(a) and 6.5(b). Across the board, jython seems to save the most traffic and execution time, saving 38% and 21% of traffic at 2MB and 4MB at our constrained bandwidth setting, respectively, and improving the number of cycles by 44% and 25% at 2MB and 4MB, respectively. As related work discusses, the memory wall will only worsen as modern applications and CMPs grow more sophisticated, making our invalidation optimization more effective and necessary as bandwidth grows more limited.

6.2.4 Varying L2 size

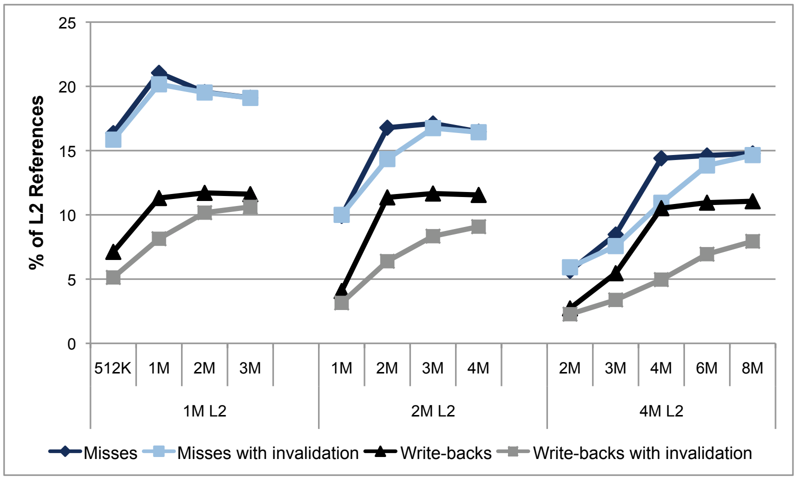


Figure 6.6: Varying L2 sizes with nursery sizes, showing misses and write-backs as a percentage of L2 references for both unmodified and optimized cache.

This section explores miss and write-back behavior at various L2 and nursery sizes, as measured with our Valgrind simulator. While the Clovertown machine we are modeling has a 4M L2 per die, many of our benchmarks are not multi-threaded and they do not accurately reflect the bandwidth pressure of many processes running on a CMP. To simulate more constrained and realistic settings, we perform experiments reducing the L2 size to model reduced capacity due to other threads. We experiment with 1M, 2M, and 4M L2 caches, while holding the L1 size constant at 32KB.

Misses and write-backs Figure 6.6 shows the misses and write-backs on average for our benchmarks using various nursery sizes from half to two-times each of the three L2 sizes listed above. The graph presents misses and write-backs as a percentage of total L2 references in both our baseline system, and in our optimized system that invalidates and eagerly evicts cache lines. The write-backs for our optimized system include reductions due to both types of savings mentioned above. The exact number of misses and write-backs is less important than trends as we vary the nursery size in relation to the L2 size. In general for our smallest nurseries, those that are half the size of the L2, a larger amount of the L2 is taken up by long-lived mature data. We expect less cache displacement from our rapidly allocating applications, and thus fewer misses and write-backs in this case. As we increase the size of the nursery above the L2 size, we expect a high percentage of misses and write backs as the nursery data touches and evicts the entire L2 cache.

As expected, keeping the nursery size constant and increasing the L2 cache size reduces misses and write-backs. Figure 6.6 shows this trend with the 2M and 3M nursery sizes. Our experimental data shows a trend that the largest number of misses and write-backs for our benchmarks with unmodified caches occurs when the nursery size is close to the L2 size. Smaller nurseries lead to fewer misses and therefore write-backs, and nursery sizes larger than the L2 reach a plateau since they can at most displace the entire L2. We see write back trends match miss trends, but overall they are fewer in number, which is expected since only misses that evict dirty cache lines result in write-backs.

We compare trends for the unoptimized cache configuration versus our optimized configuration in Figure 6.6. Our cache with invalidation reduces or matches the miss rate at all cache-nursery size pairings, and always reduces, sometimes by a large amount, the percentage of write-backs. The invalidation configuration moves the peak misses or write-backs from being at a nursery size equal to the L2 size, to a nursery size greater than the L2. At nursery sizes that match the L2 size, particularly for 2M and 4M L2 sizes, we see largely reduced miss and write-back percentages. With both the L2 and nursery at 2M, we see a 2.4% drop in the miss rate, whereas with 4M, we see at 3.5% drop. Similarly, at the 2M pairing, our optimizations reduce write backs from 11 to 6%, and at 4M, we reduce write-backs from 10.5 to 5%. As we increase the nursery size, the invalidation configuration also has more misses and write-backs, reaching the same miss rate plateau. The optimization reduces write-backs because of reduced cache pollution, even at large nursery sizes. Because we are evicting programmatically dead data eagerly, we are virtually increasing the capacity of our L2 and saving a lot of L2 misses and bandwidth which translates into improved performance, as shown above.

Write-backs savings Figure 6.7 shows the percentage of write-backs that can be eliminated averaged over all benchmarks, similar to Figure 6.3, but at many L2 and nursery size pairings, as in Figure 6.6. We break write-back savings due to our optimization into two categories, as in Section 6.2.2. We have those that are due to reducing cache pollution and improving cache replacement by eagerly evicting dead data, and we have those lines that are both dead and dirty and are saved write-backs by our optimization.

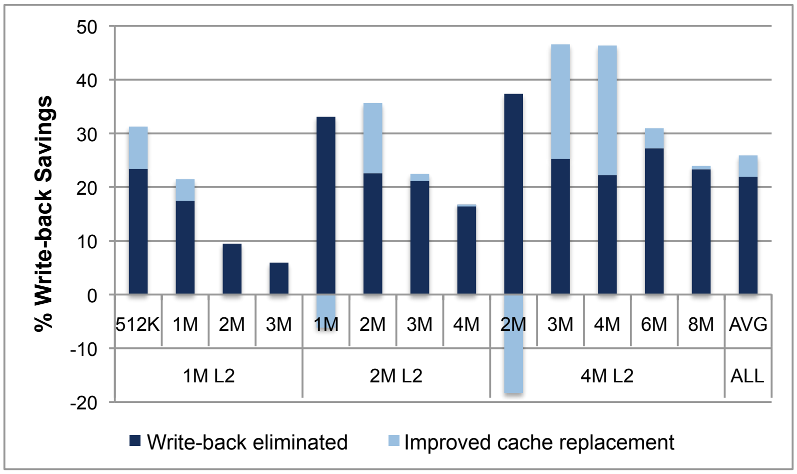


Figure 6.7: Varying L2 sizes with nursery sizes, showing write-back savings divided into those that reduce the cache pollution and those dirty dead lines that are not written back.

As expected, the number of write-backs we can eliminate with our useless invalidation optimization is highest when the nursery size matches the L2 size. We can save a total of 35% of write-backs with a 2M nursery and L2 cache. With a 4M L2 and either a 3M or 4M nursery, we save our largest average percentage of write-backs at 46%, as shown in Figure 6.3.

In general, as nursery sizes grow with respect to the L2 size, we eliminate fewer write-backs. Because the nursery size grows, collection and thus invalidation is less frequent, and we are less able to reduce the cache pressure of the nursery that overruns the entire L2 cache. Also, not all nursery data can fit into the L2 cache, and therefore older parts of the nursery are naturally evicted as newer parts are allocated and accessed. Therefore, there is a reduced number of write-backs we can avoid. At small nursery sizes, the total savings due to our optimization is lower because there is less nursery cache pollution in general, and more mature data resident in cache. The nursery fits entirely in the L2 cache, so our invalidation can remove large percentages of dirty write-backs of dead data. Looking at the smallest nursery size with 2M and 4M L2 caches, we see that our optimized cache writes back more data, unable to improve cache replacement with fewer write-backs. We cannot reduce the cache pressure when the nursery entirely fits in the L2 cache, and we eagerly invalidate it. In the unoptimized system, this data would remain in cache and never get evicted or invalidated, so our optimization is not as helpful in this case. However, as we have pointed out, most production JVMs use larger nursery sizes. In general we find the write-back savings due to reducing cache pollution are very dependent on the nursery size, whereas the number of write-backs that we can eliminate because they contain dead data is more stable across nursery sizes.

We surmise that the high percentage of write-backs saved when using a very small 512KB nursery with a 1M L2 size has to do with the large number of L2 misses, as seen in Figure 6.6. The small nursery size means that objects are not given as long a chance to die, and are promoted to the mature space sometimes prematurely. The very small L2 size means that we have a large number of misses because most application footprints cannot fit in the reduced space. At this data point, we save a large percentage of write-backs because the whole of the nursery is resident in the L2 cache. Our optimization can still save some cache pollution; however, we surmise that if we went to even smaller (unrealistic) nursery sizes, the percentage of write-backs saved would drop again, similar to the 2M and 4M L2 trends.

This elimination of a large percentage of write-backs corresponds to a great reduction in traffic and bandwidth between lower levels of cache and memory, and improved application performance as we have shown in Section 6.2.3.

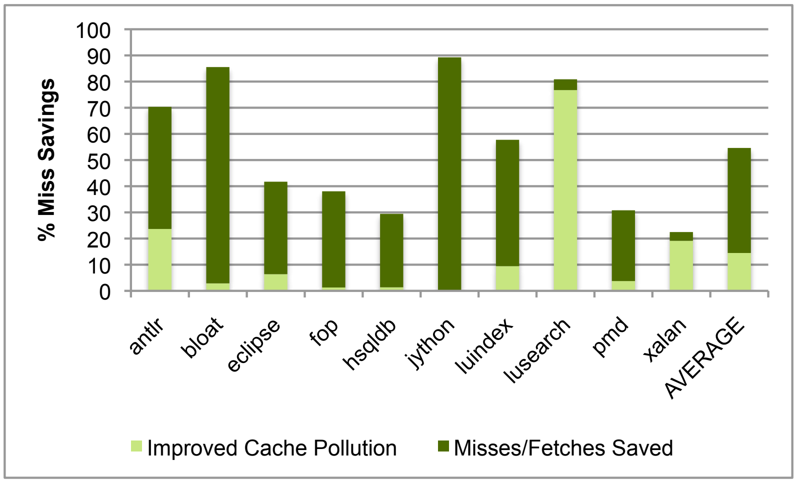


Figure 6.8: Per benchmark percentage of misses that we can save with in-cache zeroing using a 4MB nursery and L2 cache. We divide savings into reducing cache pollution and avoiding misses that result in fetches of data that we instead in-cache zero.

6.2.5 In-Cache Zeroing

We perform experiments in Valgrind to test the potential of our design of in-cache zeroing to reduce cache pollution, and read and write traffic to memory. We experiment with a 4M nursery and 4M L2 cache. Figure 6.8 shows the misses saved using only the in-cache zero optimization as a percentage of the baseline with no optimizations. We divide up the misses saved into two categories: those saved by avoiding the cache displacement of software's zeroing instructions that eagerly zero ahead of data use ("Improved Cache Pollution"), and those that could be eliminated by not fetching data when we lazily zero, but instead zeroing in-cache ("Misses/Fetches Saved"). Together, these two ways to reduce cache misses collectively save on average 55% of misses. For all benchmarks but lusearch, the majority of savings comes from not having to fetch the invalid data from memory when we lazily zero it, with jython having the most savings at 89%. The lusearch benchmark,

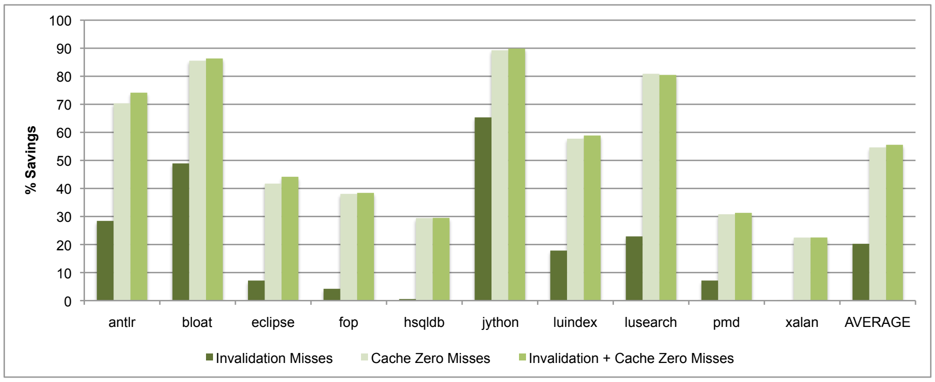


Figure 6.9: Using a 4MB nursery and L2 cache, for each benchmark we show the percentage of savings over the baseline of misses with only invalidation, with only in-cache zeroing, and with both optimizations enabled.

however, saves 77% of misses just by eliminating the software's zeroing instructions and not displacing the cache by eager zeroing. The general benchmark savings trends roughly match the write-back savings we can achieve with our invalidation optimization. These results suggest that avoiding the fetch of invalid data is very effective at reducing traffic, while the cache effects of eager zeroing are less imposing. Our in-cache zero optimization is able to save on average 11% of references to the L2 cache (not shown in graphical form) solely by eliminating software's zeroing instructions. By having hardware automatically maintain a high-water mark and zeroing new memory in cache without fetching from memory, we can save over half of L2 cache misses.

We compare the savings in L2 misses and write-backs for each of our optimization configurations and combinations. Figures 6.9 and 6.10 shos for the 4M nursery and L2 cache setting, the percentage savings of both misses and write-backs over the baseline. We show savings per benchmark with only the invalidation optimization, only the in-cache zero optimization, and both enabled together. In general, the in-cache zeroing targets reducing the miss rate, which corresponds

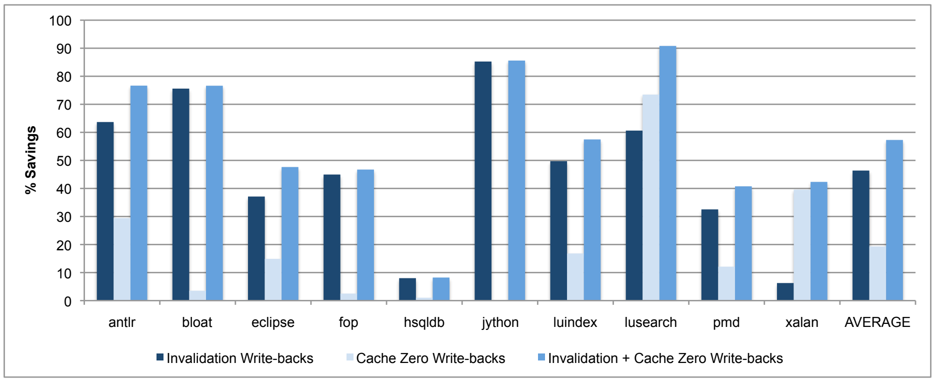


Figure 6.10: Using a 4MB nursery and L2 cache, for each benchmark we show the percentage of savings over the baseline of write-backs with only invalidation, with only in-cache zeroing, and with both optimizations enabled.

to the number of fetches from main memory, while the invalidation optimization targets reducing the number of write-backs. In Figure 6.9, we see this trend: invalidation saves fewer misses than in-cache zeroing, but when combined we can save the largest percentage of L2 misses at over 50%. For write-backs, Figure 6.10 shows that in-cache zeroing can save modestly by reducing the cache pollution because it does not include software's zeroing instructions, nor does it have to write back eagerly zeroed data before it is used. Invalidation saves more write-backs to main memory, but together the optimizations can save close to 60% of write-backs. The two outliers in this trend are lusearch and xalan that save a larger percentage of write-backs with in-cache zeroing than invalidation. Figure 6.8 shows that these benchmarks both have a high percentage of misses saved with in-cache zeroing due to reduced cache pollution. It is possible these benchmarks are more sensitive to cache pollution from software's zeroing instructions that eagerly zero data. Their usage patterns could access mature data after software's eager zeroing, thus causing a lot of unnecessary write-backs of zero nursery data. Lazy hardware zeroing then avoids many write-backs of this displacing data. Overall, we see on average that in-cache zeroing raises the miss savings to 55% from invalidation's 20%. Hardware zeroing can save on average 19% of write-backs, but invalidation is more effective at saving write-backs at 46%. Together, our optimizations can save 56% of L2 misses and 57% of write-backs on average, leading to improved cache efficiency, and a large reduction in both read and write traffic to main memory.

6.2.6 Priority Biasing

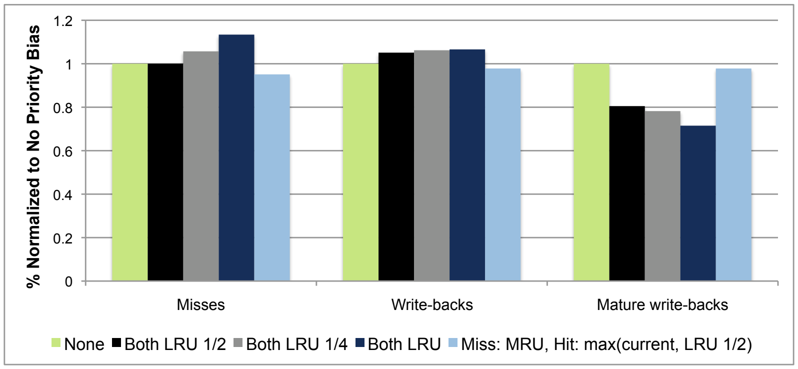


Figure 6.11: Comparing cache statistics for various set priority biasing techniques, using a 4MB nursery and L2.

We performed detailed experiments to test the hypothesis that rapidly allocated, short-lived objects that are cache-displacing could benefit from being more eagerly evicted from the cache. Upon a cache access, either for a line that is newly installed or that hits in the cache, the line is usually put in the most-recently-used position of its set. For our priority biasing experiments, we modify this position for addresses in the short-lived address range, placing them closer to the LRU position. We experiment with four different priority biasing policies: a) Both upon installation and hit, the line is put half way between MRU and LRU ("Both LRU 1/2"); b) Both upon installation and hit, the line is put half way between the middle set position and LRU ("Both LRU 1/4"); c) Both upon installation and hit, the line is put at the LRU position and is very eagerly evicted ("Both LRU"); and finally d) Upon installation, the line is put in the MRU position as normal, and upon hit, the line is placed at the maximum (closest to MRU) between its current set position and the middle set position ("Miss: MRU, Hit: max(current, LRU 1/2)"). This final policy assumes the line will be reused most right after it is brought into cache, hence placing it in the MRU position. We then prefer not to demote the cache line if it is hit, instead having it remain in its position if it is between MRU and the mid-point, or boosting the cache line to the mid-point if it is closer to LRU. This technique harnesses data's temporal locality, letting the data naturally degrade its set position, more closely matching the default MRU policy. We compare each of these four early-eviction policies to no priority biasing, which places all cache accesses in the MRU position ("None").

It should be noted that actual hardware is not as precise with cache set ordering as knowing the exact index of each line between MRU and LRU. Many processors use the pseudo-LRU or tree-LRU organization of cache sets to roughly approximate the most to least-recently-used spectrum [1]. However, for our experiments, we wanted an accurate study of priority biasing techniques, and are able to simulate full precision in Valgrind.

Figure 6.11 shows how cache behavior changes on average across benchmarks as we vary line placement using a 4MB nursery and L2 size. We compare misses and write-backs for a cache enabled with only the priority biasing optimization, not including the invalidation optimization. All percentages are normalized to the cache with no priority biasing enabled. Lower bars show that priority biasing is effective in reducing the number of misses or write-backs. When comparing miss percentages with a 16-way set associative L2, the priority bias technique "Miss: MRU, Hit: max(current, LRU 1/2)" slightly lowers the cache miss rate by about

5%, while all other techniques raise the number of misses. We see a similar pattern for write-backs, as the same technique saves 2% and other priority biasing policies generate more writes to memory. All four priority biasing techniques reduce mature write-backs. This reduction is expected because as we eagerly evict nursery data, we expect more long-lived mature data to be resident in cache and avoid being written back. The "Both LRU" policy reduces the number of mature write-backs the most by 28% because it aggressively evicts more nursery data; however, it does this at the expense of raising the misses by 13% and write-backs by 6%. The better performing, more conservative policy, "Miss: MRU, Hit: max(current, LRU 1/2)", decreases mature write-backs by only 2%.

In summary, we tried many different priority biasing techniques, and found that on the whole, they too aggressively evict nursery data, leading to increased miss rates and write-backs. However, these eager-eviction policies are effective at keeping more long-lived data in cache. Our best performing policy was able to slightly reduce the percentage of misses and write-backs by more closely matching the MRU default. The fact that this technique performed well suggests that nursery data reuse varies within benchmark runs, and lines should not be too eagerly demoted in the cache set so that they are too quickly evicted. We see evidence of object streams, finding that nursery data has irregular reuse and although most data is short-lived, some data has unpredictable lifetimes.

6.3 Related Work

This section surveys research studying the bandwidth and allocation walls for modern programs running on modern CMP hardware. We detail related work on optimizing the caches based on identifying dead cache blocks [1]. We conclude with

---

[1] A cache block and cache line are synonymous here and in the literature.

work on changing cache set replacement policies to improve cache behavior.

Motivation. Many researchers have shown that bandwidth is an increasing performance bottleneck as chip multiprocessor machines scale up the number of cores, both in hardware and software [41, 55, 61, 77]. Rogers et al. showed that the *bandwidth wall* between off-chip memory and on-chip cores is a performance bottleneck, and can severely limit core scaling [61]. Molka et al. perform a detailed study of the memory performance of the Nehalem microarchitecture [55]. They find that on the four core Intel X5570 i7 processor, memory read bandwidth does not scale beyond two cores, while write bandwidth does not scale beyond one core.

On the software side, Zhao et al. show that allocation intensive Java programs pose an *allocation wall* which limits application scaling and performance [77]. Studying "partially scalable" benchmarks, they found a strong correlation between object allocation rates and memory bus write traffic. As allocation quickly saturates the limited write bandwidth, Java applications' ability to scale to effectively use more cores degrades dramatically, running into this problematic wall. Inoue et al. explore web-based server workloads, and found that the memory manager has to take into account the bus traffic it creates, or it will limit application scaling as the number of cores increases [41]. Languages with region allocators and garbage collection encourage rapid allocation, which degrades performance in the memory subsystem. Blackburn and McKinley call this rapidly allocated, short-lived data *object streams* [17], which displace data in the cache, like traditional streams, but is different in two key ways: 1) most data has short reuse and some data is long-lived, and 2) access is irregular. Our work addresses these allocation characteristics in prolific high-level language applications that interact poorly with hardware memory.

Memory latency has always lagged behind processor latency, but recent research shows that the memory bottleneck is exacerbated by frequently allocating applications and the increase in cores per chip that has not been accompanied by commensurate increases in memory bandwidth. More communication between cores and memory puts pressure on bandwidth, and degrades scalability and therefore performance.

Software's identification of invalid data. The ESKIMO system is most closely related to our work [42]. ESKIMO passes to hardware the software's semantic knowledge about allocated and freed objects for explicitly-managed C programs to reduce energy and power requirements for DRAM. Like our work, they identify invalid memory, memory that has been allocated but not initialized by the program and memory that has been freed, and communicate this information to the cache. Their mechanisms operate at the cache line granularity. They perform various DRAM optimizations, including eliminating write-backs from cache to main memory if the fine-grained regions identified by software as "inconsequent" occupy one or more entire cache lines. To optimize the memory system further, ESKIMO re-implemented previous work by Lewis et al. [42, 49]. When a cache write miss occurs to an inconsequent address, the data need not be fetched from main memory. They add one bit to every cache line to identify inconsequent lines. In their system, the hardware must store a map of allocations to cache lines. This map requires fine-grained tracking of objects and their alignment in cache lines. In contrast to a map proportional to the number of allocations, our approach adds only a few registers to hardware. We exploit contiguous region allocators, used exclusively in high performance virtual machines for managed languages and popular in demanding C programs. For example, Apache uses a region allocator. We exploit the generational heap organization to cheaply pass down to hardware a large region of memory that is dead instead of frequently communicating small regions after each program call to *free*. We leverage Java semantics that zero-initialize all objects to zero cache lines that are being touched for the first time, avoiding reads to main memory. Both approaches reduce bandwidth. Whereas their approach reduces power, and is performance-neutral, our optimizations improve application performance as well.

Dead cache blocks and prefetching. Many researchers have focused on predicting which blocks in cache can be evicted based on usage patterns, mainly in conjunction with prefetching [2, 37, 47, 48, 51, 52, 64, 73, 74]. This work uses the cache hierarchy more efficiently by predicting which data is "dead", i.e. will not be used again, in hardware, but does not necessarily eliminate traffic to memory or bandwidth and in fact often increases it. Lai et al. predict dead cache blocks based on traces of memory operations, and use this information to predict the next address to prefetch into the cache after dead block eviction [48]. Other researchers predict dead blocks using a counter of accesses per L2 cache line, dynamically learning from the program [47]. Hu et al. and Abella et al. both predict dead blocks with time-keeping techniques correlated with cache blocks [2, 37]. Hu et al. use identified dead blocks to improving performance with a victim cache and prefetching [37]. Abella et al. turns off dead L2 blocks dynamically to save power [2]. Liu et al. introduce a new technique to identify dead cache blocks by tracking cache bursts to a particular block, calling a block dead after it leaves the MRU position [52]. Scheduled region prefetching tries to tolerate the copious amounts of time the processor stalls while moving data between memory and the lowest level cache [51]. All of this work is done at the level of hardware only, which can suffer mispredictions and lacks higher-level semantic information from software, and thus must always write modified data to lower levels of the memory hierarchy.

Wang et al. use a cooperative software-hardware approach to achieve more effective prefetching, and perform static compiler analysis on programs to identify data that will not be reused again soon [73, 74]. They pass this information down to hardware as a hint attached to memory instructions, and modify hardware to have one extra bit per cache line. The hardware uses this information to optimize cache evictions as well as to assist hardware prefetching on C and Fortran benchmarks. Sartor et al. extend this work to give hints to hardware about data that will be reused again soon and should be kept in cache [64]. While this research focuses on software-hardware cooperation, the hints are based on static program information and do not guarantee that the data that should be evicted is dead or will not be used again by the program.

Previous work is limited by hardware's view of memory accesses or by static program information about reuse, and sometimes increases bandwidth needs due to mispredictions and prefetching. In comparison, we use the managed runtime's dynamic view of program data that is actually dead to inform the hardware what it can safely invalidate in cache to avoid traffic to memory. It is possible our techniques could be combined with hardware prefetchers to reduce cache pollution, however, we leave this exploration to future work.

Line placement within a cache set. Previous work on cache design motivates our optimizations that try to limit cache displacement and traffic. Jouppi investigated many different cache policies and their effect on performance [45]. In particular, his *write-validate* policy combines no-fetch-on-write and write-allocate for good performance. He suggests not going to memory upon a cache write, and storing sub-block dirty bits to keep track of modified data in cache. His best policy motivates zeroing cache lines without reading from memory, and limiting write back traffic to memory, both of which we target with our optimizations. The PowerPC instruction set architectures (ISAs) include a data cache block zero (dcbz) instruction to forego fetching from memory and zero a cache line directly for better cache performance [68].

Previous hardware policies change the order of replacement of lines within a cache set to improve performance [46, 59, 75]. Kampe et al.'s Self-Correcting LRU algorithm is designed to correct LRU replacement techniques, including evicting dead lines that are not accessed again after they leave the MRU position [46]. Wong and Baer choose to evict L2 cache lines that did not exhibit temporal locality [75]. Qureshi et al. dynamically place cache lines either in the LRU position to reduce capacity misses or in the MRU position to account for working set changes [59]. This work is similar to our priority cache set biasing for cache-polluting data, but is not based on semantic information passed down from software as ours is.

6.4 Useless Write Back Conclusions

We introduce a software-hardware cooperative approach to save memory traffic and bandwidth. Detailed analysis shows that a surprising 88% majority of write-backs to memory for the Java DaCapo benchmarks are useless and never used again. Managed programs tend to rapidly allocate short-lived data, marching through the cache linearly, displacing other usefully cache lines, and increasing bandwidth usage. This bandwidth is not essential to program execution. Furthermore it aggravates the memory bottleneck, which CMPs are already exacerbating by adding more cores without a commensurate amount of memory and memory bandwidth. We show how software-hardware cooperation alleviates this problem. The VM communicates candidate address ranges of dead data to hardware, and the cache hardware improves its replacement decisions and eliminates the write-back of dirty, but useless data. We show our invalidation optimization reduces write-backs on average by

48% on a variety of nursery and memory system configurations, leading to a 13% application performance improvement.

We investigate the affect of cache-line zeroing and show it reduces useless initializing writes and fetches from memory. With only software zeroing instructions, VMs typically perform zeroing en masse for performance, much earlier than strictly required, which negatively affects cache efficiency. Although our optimizations currently target the nursery region, they are applicable more broadly and can be incorporated into full-heap garbage collectors and region allocators.

This work shows how to harness the memory management abstraction of ubiquitous high-level languages to communicate semantics efficiently with hardware, reducing their memory costs, climbing the memory and bandwidth walls. Such software-hardware cooperation is likely to become more important, especially for memory system performance, as chip multiprocessors add processors and grow in complexity.

Chapter 7

Conclusions

This thesis has analyzed the sources of and solutions to memory inefficiencies for managed language applications running on emerging architectures. While raising the level of abstraction in managed languages has benefited programmers by easing development, the extra layers of memory management have created inefficiencies. Managed language program complexity and object organization have encouraged over-provisioning and redundancy in heap data structures, in particular arrays. Our heap data compression study found that over half of the heap can be compressed to save space. Similarly, rapid allocation of short-lived data made conducive by garbage collectors has created excessive cache-to-memory traffic, much of which is useless. Our memory subsystem study found that 88% of writes to memory from cache are never read again by the program. These memory inefficiencies degrade application performance. This degradation is worsened as the real-time and embedded domains have become more prevalent, and hardware has transitioned to chip multiprocessor machines that are bandwidth-limited.

Although these inefficiencies are the costs of using a high-level language, the managed runtime's virtualization of memory also offers an opportunity to dynamically optimize memory while the program is running without assistance from the programmer. We take advantage of this abstraction to introduce z-rays, a more time and space-efficient design of arrays, that makes compression of parts of arrays tractable and effective. Z-rays' flexibility bridges the gap between fast contiguous array layouts, and space-efficient and space-predictable discontiguous layouts. To combat useless memory traffic, we introduce a software-hardware cooperative optimization to reduce bandwidth in the memory subsystem. We take advantage of the garbage collector's identification of ranges of dead data, communicating program semantics down to the cache hierarchy to invalidate dead data in order to help cache displacement, and eliminate reads and writes to memory. We show software-hardware cooperation reduces traffic to memory by about half, improving application performance substantially as bandwidth gets more limited in future architectures.

We show that the memory abstraction of managed languages is not just a cost to be borne, but an opportunity to overcome the memory wall. Memory efficiency has become a prime concern as hardware hits physical limits and transitions to chip multiprocessor machines which exacerbate the memory bottleneck. As both languages and architectures have evolved, further stressing the memory system, we have developed new dynamic optimizations to accommodate the distinct challenges that have arisen at many levels. We have not only made heap layout more efficient, but have advocated a software-hardware cooperative approach for communicating semantic information to low-level hardware to optimize the memory subsystem's bandwidth. This thesis improves memory efficiency and performance of sophisticated modern programs running on current and future architectures, creating a more cooperative optimization framework.

Bibliography

[1] *Intel Pentium 4 and Intel Xeon Processor Optimization. Reference Manual.* June 2002.

[2] Jaume Abella, Antonio Gonzalez, Xavier Vera, and Michael F.P. O'Boyle. IATAC: A smart predictor to turn-off L2 cache lines. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2(1):55–77, 2005.

[3] AICAS. Jamaica VM. `http://www.aicas.com/`.

[4] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[5] C. Scott Ananian and Martin Rinard. Data size optimizations for Java programs. In *ACM Languages, Compiler, and Tool Support for Embedded Systems (LCTES)*, pages 59–68, 2003.

[6] Andrew W. Appel and Marcelo J. R. Gonçalves. Hash-consing garbage collection. Technical Report CS-TR-412-93, Princeton University, 1993.

[7] David Bacon, Perry Cheng, and V. T. Rajan. Controlling fragmentation and space consumption in the Metronome, a real-time garbage collector for Java. In *ACM Languages, Compiler, and Tool Support for Embedded Systems (LCTES)*, pages 81–92, 2003.

[8] David Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *ACM Principles of Programming Languages (POPL)*, pages 285–298, 2003.

[9] David F. Bacon, Perry Cheng, David Grove, and Martin T. Vechev. Syncopation: Generational real-time garbage collection in the Metronome. In *ACM Languages, Compiler, and Tool Support for Embedded Systems (LCTES)*, pages 183–192, 2005.

[10] F.L. Bauer and H. Wössner. The Plankalkül of Konrad Zuse: a forerunner of today's programming languages. *Communications of the ACM (CACM)*, 15(7):678–685, 1972.

[11] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *ACM Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1–12, Seattle, WA, November 2002.

[12] E.D. Berger, K.S. McKinley, R.D. Blumofe, and P.R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 117–128, 2000.

[13] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, Frampton D., S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 169–190, 2006.

[14] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Myths and realities: The performance impact of garbage collection. In *Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 25–36, 2004.

[15] Stephen M. Blackburn and Antony L. Hosking. Barriers: Friend or foe? In *International Symposium on Memory Management (ISMM)*, pages 143–151, 2004.

[16] Stephen M. Blackburn and Kathryn S. McKinley. In or out? Putting write barriers in their place. In *International Symposium on Memory Management (ISMM)*, pages 175–184, 2002.

[17] Stephen M. Blackburn and Kathryn S. McKinley. Transient caches and object streams. Technical Report TR-CS-06-03, Australian National University, Department of Computer Science, October 2006.

[18] Stephen M. Blackburn and Kathryn S. McKinley. Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *ACM Programming Language Design and Implementation (PLDI)*, pages 22–32, 2008.

[19] R. Bodík, R. Gupta, and V. Sarkar. ABCD: eliminating array bounds checks on demand. In *ACM Programming Language Design and Implementation (PLDI)*, pages 321–333, 2000.

[20] Mark Bohr. A 30 year retrospective on dennard's mosfet scaling paper. *IEEE Solid-State Circuits Society News*, 12(1):11–13, 2007.

[21] G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, B. Mathiske, and M. Wolczko. Heap compression for memory-constrained Java environments.

In *ACM Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 282–301, 2003.

[22] Guangyu Chen, Mahmut Kandemir, and Mary J. Irwin. Exploiting frequent field values in Java objects for reducing heap memory reqirements. In *ACM/USENIX Virtual Execution Environments (VEE)*, pages 68–78, 2005.

[23] Guangyu Chen, Mahmut Kandemir, Narayanan Vijaykrishnan, and Mary Jane Irwin. Field level analysis for heap space optimization in embedded Java environments. In *International Symposium on Memory Management (ISMM)*, pages 131–142, 2004.

[24] Perry Cheng and Guy E. Blelloch. A parallel, real-time garbage collector. In *ACM Programming Language Design and Implementation (PLDI)*, pages 125–136, 2001.

[25] C. Click, G. Tene, and M. Wolf. The pauseless GC algorithm. In *ACM/USENIX Virtual Execution Environments (VEE)*, pages 46–56, 2005.

[26] Nathan Dean Cooprider and John David Regehr. Offline compression for on-chip RAM. In *ACM Programming Language Design and Implementation (PLDI)*, pages 363–372, 2007.

[27] SPEC corporation. SPECjbb2005 Java server benchmark, 2005. ftp://ftp.-spec.org/jbb2005/.

[28] Sylvia Dieckmann and Urs Hölzle. A study of allocation behavior of the SPECjvm98 Java benchmarks. In *European Conference for Object-Oriented Programming (ECOOP)*, pages 92–115, 1999.

[29] Jens Ernst, William Evans, Christopher W. Fraser, Steven Lucco, and Todd A. Proebsting. Code compression. In *ACM Programming Language Design and Implementation (PLDI)*, pages 358–365, 1997.

[30] William S. Evans and Christopher W. Fraser. Bytecode compression via profiled grammar rewriting. In *ACM Programming Language Design and Implementation (PLDI)*, pages 148–155, 2001.

[31] Fiji Systems LLC. Fiji VM. `http://www.fiji-systems.com/`.

[32] Robert Fitzgerald and David Tarditi. The case for profile-directed selection of garbage collectors. In *International Symposium on Memory Management (ISMM)*, pages 111–120, 2000.

[33] D. Frampton, S. M. Blackburn, P. Cheng, R. J. Garner, D. Grove, J. E. B. Moss, and S. I. Salishev. Demystifying magic: High-level low-level programming. In *Virtual Execution Environments (VEE)*, pages 81–90, 2009.

[34] Jungwoo Ha, Magnus Gustafsson, Stephen M. Blackburn, and Kathryn S. McKinley. Microarchitectural characterization of production JVMs and Java workloads. In *IBM CAS Workshop*, 2008.

[35] T. Harris, S. Tomic, A. Cristal, and O. Unsal. Dynamic filtering: Multi-purpose architecture support for language runtime systems. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 39–52, 2010.

[36] Antony L. Hosking, J. Eliot B. Moss, and Darko Stefanovic. A comparative performance evaluation of write barrier implementations. In *ACM Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 92–109, 1992.

[37] Zhigang Hu, Stefanos Kaxiras, and Margaret Martonosi. Timekeeping in the memory system: Predicting and optimizing memory behavior. In *International Symposium on Computer Architecture (ISCA)*, pages 209–220, 2002.

[38] X. Huang, Z. Wang, S.M. Blackburn, K.S. McKinley, J.E.B. Moss, and P. Cheng. The garbage collection advantage: Improving mutator locality. In *ACM Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 69–80, 2004.

[39] IBM. Cache line invalidate instruction. `http://publib.boulder.ibm.com/infocenter/aix/v6r1/index.jsp?topic=/com.%ibm.aix.aixassem/doc/alangref/cli.htm`.

[40] IBM. Websphere real time. `http://www-01.ibm.com/software/web-servers/realtime/`.

[41] Hiroshi Inoue, Hideaki Komatsu, and Toshio Nakatani. A study of memory management for web-based applications on multicore processors. In *ACM Programming Language Design and Implementation (PLDI)*, pages 386–396, 2009.

[42] Ciji Isen and Lizy John. ESKIMO: Energy savings using semantic knowledge of inconsequential memory occupancy for DRAM subsystem. In *ACM/IEEE International Symposium on Microarchitecture*, pages 337–346, 2009.

[43] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Mikio Takeuchi, Takeshi Ogasawara, Toshio Suganuma, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Design, implementation, and evaluation of optimizations in a just-in-time compiler. In *ACM Java Grande*, pages 119–128, 1999.

[44] Richard Jones and Rafael Lins. *Garbage collection: Algorithms for automatic dynamic memory management*. John Wiley & Son Ltd., 1996.

[45] Norman P. Jouppi. Cache write policies and performance. In *International Symposium on Computer Architecture (ISCA)*, pages 191–201, 1993.

[46] Martin Kampe, Per Stenstrom, and Michel Dubois. Self-correcting LRU replacement policies. In *Conference on Computing Frontiers*, pages 181–191, 2004.

[47] Mazen Kharbutli and Yan Solihin. Counter-based cache replacement and bypassing algorithms. *IEEE Transactions on Computers*, 57(4):433–447, 2008.

[48] An-Chow Lai, Cem Fide, and Babak Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *International Symposium on Computer Architecture (ISCA)*, pages 144–154, 2001.

[49] Jarrod A. Lewis, Bryan Black, and Mikko H. Lipasti. Avoiding initialization misses to the heap. In *International Symposium on Computer Architecture (ISCA)*, pages 183–194, 2002.

[50] Henry Lieberman and Carl E. Hewitt. A real time garbage collector based on the lifetimes of objects. *Communications of the ACM (CACM)*, 26(6):419–429, 1983.

[51] Wei-Fen Lin, Steven K. Reinhardt, and Doug Burger. Designing a modern memory hierarchy with hardware prefetching. 50(11):1202–1218, 2001.

[52] Haiming Liu, Michael Ferdman, Jaehyuk Huh, and Doug Burger. Cache bursts: A new approach for eliminating dead blocks and increasing cache effi-

ciency. In *ACM/IEEE International Symposium on Microarchitecture*, pages 222–233, 2008.

[53] Darko Marinov and Robert O'Callahan. Object equality profiling. In *ACM Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 313–325, 2003.

[54] Nick Mitchell and Gary Sevitsky. The causes of bloat, the limits of health. In *ACM Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 245–260, 2007.

[55] D. Molka, D. Hackenberg, R. Schone, and M.S. Muller. Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system. In *Parallel Architectures and Compilation Techniques (PACT)*, pages 261 –270, 2009.

[56] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *ACM Programming Language Design and Implementation (PLDI)*, pages 89–100, 2007.

[57] Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony Hosking, Ethan Blanton, and Jan Vitek. Schism: Fragmentaton-tolerant real-time garbage collection. In *Programming Language Design and Implementation (PLDI)*, pages 146–159, 2010.

[58] J. S. Quarterman, A. Silberschatz, and J. L. Peterson. 4.2BSD and 4.3BSD as examples of the UNIX system. *ACM Computing Surveys*, 17(4):379–418, 1985.

[59] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive insertion policies for high performance caching. In *International Symposium on Computer Architecture (ISCA)*, pages 381–391, 2007.

[60] J.M. Robson. Worst case fragmentation of first fit and best fit storage allocation strategies. *ACM Computer Journal*, 20(3):242–244, 1977.

[61] Brian Rogers, Anil Krishna, Gordon Bell, Ken Vu, Xiaowei Jiang, and Yan Solihin. Scaling the bandwidth wall: Challenges in and avenues for cmp scaling. In *International Symposium on Computer Architecture (ISCA)*, pages 371–382, 2009.

[62] Jennifer B. Sartor, Stephen M. Blackburn, Daniel Frampton, Martin Hirzel, and Kathryn S. McKinley. Z-rays: Divide arrays and conquer speed and flexibility. In *Programming Language Design and Implementation (PLDI)*, pages 471–482, 2010.

[63] Jennifer B. Sartor, Martin Hirzel, and Kathryn S. McKinley. No bit left behind: The limits of heap data compression. In *International Symposium on Memory Management (ISMM)*, pages 111–120, 2008.

[64] Jennifer B. Sartor, Subramaniam Venkiteswaran, Kathryn S. McKinley, and Zhenlin Wang. Cooperative caching with keep-me and evict-me. In *Workshop on Interaction between Compilers and Computer Architectures*, pages 46–57, 2005.

[65] Semiconductor Industry Association. SIA world semiconductor forcast 2007–2010, November 2007. http://www.sia-online.org/pre_release.cfm?ID=455.

[66] Ran Shaham, Elliot K. Kolodner, and Mooly Sagiv. Heap profiling for space-efficient Java. In *ACM Programming Language Design and Implementation (PLDI)*, pages 104–113, 2001.

[67] Fridtjof Siebert. Eliminating external fragmentation in a non-moving garbage collector for Java. In *ACM Compilers, Architectures, and Synthesis for Embedded Systems (CASES)*, pages 9–17, 2000.

[68] Ed Sikha, Rick Simpson, Cathy May, and Hank Warren. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufmann Publishers, 1994.

[69] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bitwidth analysis with application to silicon compilation. In *ACM Programming Language Design and Implementation (PLDI)*, pages 108–120, 2000.

[70] TIOBE Software. TIOBE programming community index, 2007. http://-tiobe.com.tpci.html.

[71] Ben Titzer, Joshua S. Auerbach, David F. Bacon, and Jens Palsberg. The ExoVM system for automatic VM application reduction. In *ACM Programming Language Design and Implementation (PLDI)*, pages 352–362, 2007.

[72] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Software Engineering Symposium on Practical Software Development Environments (SESPSDE)*, pages 157–167, 1984.

[73] Zhenlin Wang, Doug Burger, Kathryn S. McKinley, Steven K. Reinhardt, and Charles C. Weems. Guided region prefetching: A cooperative hardware/software approach. In *International Symposium on Computer Architecture (ISCA)*, pages 388–398, 2003.

[74] Zhenlin Wang, Kathryn S. McKinley, Arnold L. Rosenberg, and Charles C. Weems. Using the compiler to improve cache replacement decisions. In *Parallel Architectures and Compilation Techniques (PACT)*, pages 199–210, 2002.

[75] Wayne A. Wong and Jean-Loup Baer. Modified LRU policies for improving second-level cache behavior. In *High-Performance Computer Architecture (HPCA)*, pages 49–60, 2000.

[76] Matt T. Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 23–34, 2007.

[77] Yi Zhao, Jin Shi, Kai Zheng, Haichuan Wang, Haibo Lin, and Ling Shao. Allocation wall: A limiting factor of Java applications on emerging multi-core platforms. In *ACM Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 361–376, 2009.

[78] Craig Zilles. Accordion arrays: Selective compression of unicode arrays in Java. In *International Symposium on Memory Management (ISMM)*, pages 55–66, 2007.

Index

Vita

Jennifer Bedke Sartor was born in Salt Lake City, Utah to Suzanne Bedke and Lynn Sartor. She attended Horizon High School in Scottsdale, Arizona. In December of 2001, she graduated from The University of Arizona with a Bachelors of Science degree in Mathematics and honors Computer Science, minoring in Spanish. In the fall of 2002, she started her graduate studies at The University of Texas at Austin. Jennifer received the degree of Master of Science in the field of Computer Science in December of 2004.

Permanent address: 810 Crestwood Place
Brookings, OR 97415

This dissertation was typeset with LaTeX[†] by the author.

[†] LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.