Z-Rays: Divide Arrays and Conquer Speed and Flexibility [*]

Jennifer B. Sartor[†] Stephen M. Blackburn[‡] Daniel Frampton[‡] Martin Hirzel[§] Kathryn S. McKinley[†]

[†]University of Texas at Austin {jbsartor,mckinley}@cs.utexas.edu

[‡]Australian National University {Steve.Blackburn,Daniel.Frampton}@anu.edu.au

[§]IBM Watson Research Center hirzel@us.ibm.com

Abstract

Arrays are the ubiquitous organization for indexed data. Throughout programming language evolution, implementations have laid out arrays contiguously in memory. This layout is problematic in space and time. It causes heap fragmentation, garbage collection pauses in proportion to array size, and wasted memory for sparse and over-provisioned arrays. Because of array virtualization in managed languages, an array layout that consists of indirection pointers to fixed-size discontiguous memory blocks can mitigate these problems transparently. This design however incurs significant overhead, but is justified when real-time deadlines and space constraints trump performance.

This paper proposes *z-rays*, a discontiguous array design with flexibility and efficiency. A z-ray has a *spine* with indirection pointers to fixed-size memory blocks called *arraylets*, and uses five optimizations: (1) inlining the first *N* array bytes into the spine, (2) lazy allocation, (3) zero compression, (4) fast array copy, and (5) arraylet copy-on-write. Whereas discontiguous arrays in prior work improve responsiveness and space efficiency, z-rays combine time efficiency and flexibility. On average, the best z-ray configuration performs within 12.7% of an unmodified Java Virtual Machine on 19 benchmarks, whereas previous designs have *two to three times* higher overheads. Furthermore, language implementers can configure z-ray optimizations for various design goals. This combination of performance and flexibility creates a better building block for past and future array optimization.

Categories and Subject Descriptors D3.4 [*Programming Languages*]: Processors—Memory management (garbage collection); Optimization; Run-time environments

General Terms Performance, Measurement, Experimentation

Keywords Heap, Compression, Arrays, Arraylets, Z-rays

1. Introduction

Konrad Zuse invented arrays in 1946; Fortran first implemented arrays; and every modern language includes arrays. 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. Prior work shows that just eliminating zero bytes from arrays reduces program footprints by 41% in Java benchmarks [27]. In managed languages, garbage collection uses copying to coalesce free space and reduce fragmentation. Copying and scanning arrays incur large 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 [5, 12, 28]. Siebert's design organizes array memory in trees to reduce fragmentation, but requires an expensive tree traversal for every array access [28]. Bacon et al. and Pizlo et al. use a single level of indirection to fixed-size *arraylets* [5, 25]. Chen et al. contemporaneously invented arraylets to aggressively compress arrays during collection and decompress on demand for memory-constrained embedded systems [12]. They use lazy allocation to materialize arraylets upon the first non-zero store. All prior work introduces substantial overheads. Regardless, three production Java Virtual Machines (JVMs) already use discontiguous arrays to achieve real-time bounds: IBM WebSphere Real Time [5, 19], AICAS Jamaica VM [1, 28], and Fiji VM [14, 25]. Thus, although discontiguous arrays are needed for their *flexibility*, which achieves space and time predictability, so far they have sacrificed throughput and time *efficiency*.

This paper presents *z-rays*, a discontiguous array design and JVM implementation that combines flexibility, memory efficiency, and performance. 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 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 inline small files and the beginning of large files in *i*-nodes [26]. 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*.

The collector performs zero-compression at the granularity of arraylets by eliminating arraylets that are entirely zero. When the program copies arrays, our fast array copy implementation copies contiguous chunks of memory, instead of copying element-by-element. Our copy-on-write optimization always initially shares whole arraylets that are copied and only copies later if and when the program subsequently writes to a copied arraylet. To our knowledge, this study is the first to implement array copy-on-write, show that it is does not significantly hurt performance, and show that it saves significant amounts of space. This study is also the first to rigorously evaluate and report Java array properties and their impact

[*] This work is supported by ARC DP0666059, NSF SHF0910818, NSF CSR0917191, NSF CCF0811524, NSF CNS0719966, Intel, and Google.

on discontiguous array optimization choices. 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.

Z-rays are thus immediately applicable to discontiguous arrays in embedded and real-time systems, since they improve flexibility, space efficiency, and add time efficiency. Since the largest object size determines heap fragmentation and pause times, and first-*N* increases it by *N*, some system-specific tuning may be necessary to achieve particular space and time design goals. We believe z-rays may also help to ameliorate challenges in general-purpose multicore hardware trends. For example, multicore hardware is becoming more memory-bandwidth limited because the number of processors is growing much faster than memory size. Lazy allocation and copy-on-write eliminate unnecessary, voluminous, and bursty write traffic that would otherwise slow the entire system down. Z-rays not only make discontiguous arrays more appealing for real-time virtual machines, but also make them feasible for general-purpose systems.

Our results demonstrate that z-rays achieve both performance and flexibility, making them an attractive building block for language implementation on current and future architectures.

2. Related Work

This section surveys work on implementations of discontiguous arrays, describes work on optimizing read and write barriers, and establishes how array representations relate to space consumption.

Implementing discontiguous arrays. Siebert's tree representation for arrays limits fragmentation in a non-moving garbage collector for a real-time virtual machine [1, 28]. 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.

Discontiguous arrays provide a foundation for achieving real-time guarantees in the Metronome garbage collector [4, 5]. 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" [5]. Metronome is used in IBM's WebSphere Real Time product [19] 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 [14, 25] also uses arraylets to meet real-time system demands, but the arraylet implementation is not currently optimized for throughput [24].

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 [1, 14, 19]. 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 developing a spine-with-arraylets design [12]. 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 [18]. Optimizations and hardware features such as instruction level parallelism and out-of-order processors have reduced barrier overheads over the years [7, 8, 15]. If needed, special hardware can further reduce their overheads [13, 17]. We borrow Blackburn and McKinley's fast path barrier inlining optimization and Blackburn and Hosking's evaluation methodology. Section 5.1 discusses the potential added performance benefit of compiler optimizations such as strip-mining in barriers. In summary, we exploit recent progress in barrier optimization to make z-rays efficient.

Heap object compression. High-level languages abstract memory management and object layout to improve programmer productivity, usability, and security, but abstraction usually costs. Mitchell and Sevitsky study *bloat*, spurious memory consumption caused by careless programming [22]. A shocking fraction of the Java heap is bloat, motivating the need for space savings. Sartor et al.'s limit study of Java estimates the effect of compression, and finds that array compression is likely to yield the most benefit [27]. Ananian and Rinard propose using offline profiling and an ahead-of-time compiler to perform compression techniques such as bit-width reduction [3]. Zilles reduces the bit-width of unicode character arrays from 16 bits to 8 bits [31]. Chen et al. compress arraylets [12]. All these techniques trade time for space, incurring time overheads to reduce space consumption in embedded systems. This paper primarily studies ways to improve discontiguous array performance and is complementary to using them for compression. Z-rays offer a much better building block for compression and future array optimization needs.

3. Background

This section briefly discusses our implementation context for discontiguous arrays in Java. We implement z-rays in Jikes RVM [2], a high performance Java-in-Java virtual machine, but our use of Jikes RVM is not integral to our approach.

Java 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 such as Java.

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. 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 [21, 30]. It allocates objects into a nursery. When the nursery fills up, the collector copies surviving objects into a mature space, but most objects do not survive. To avoid scanning the mature space for a nursery collection, a generational write barrier records pointers from the mature space to the nursery space and the collector treats these pointers as roots [21, 30]. Once frequent nursery collections fill the mature space, a *full heap collection* scavenges the entire heap.

Read and Write Barriers. Java has barriers for bounds checks on every array read and write (shown in Figure 3(a)), cast checks on every reference array write, and the generational write barrier described above. Java optimizing compilers eliminate provably redundant checks [11, 20]. 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 (See Section 5.3).

4. Z-rays: Efficient Discontiguous Arrays

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 [4, 12]. Next, this section presents the z-ray memory management strategy and the five z-ray optimizations.

4.1 Basic Arraylets

Similar to previous work, we divide each array into exactly one *spine* and zero or more fixed-size *arraylets*, as shown in Figure 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 [4], we *inline* the remainder into the spine directly after indirection pointers (see Figure 1(a)), which avoids managing variable-sized arraylets or wasting any arraylet space. We include an indirection pointer to the remainder 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 3(b). The arraylet size is a global constant, and we explore different values in Section 7.

4.2 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 1(b) shows the *arraylet space*. The arraylet space uses a non-copying collector with fixed-sized blocks equal to the arraylet size.

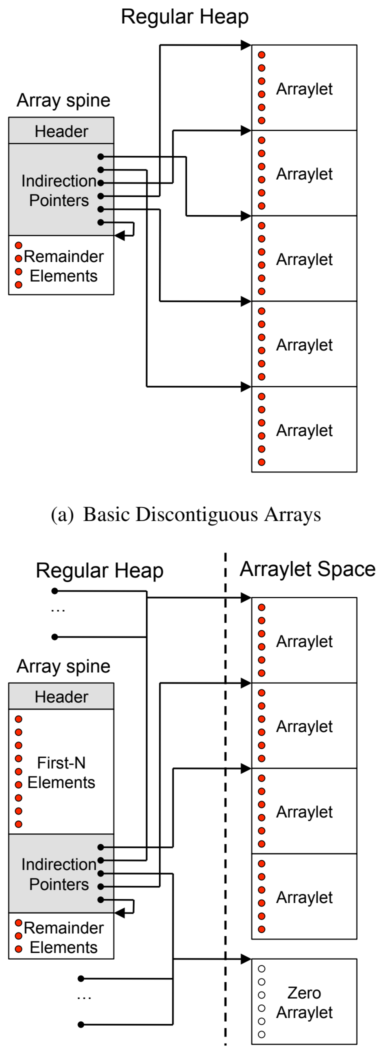


Figure 1. Discontiguous reference arrays divided into a spine pointing to arraylets for prior work and optimized Z-rays.

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 distributeed 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. In principle the arraylet space can easily be defragmented since all arraylets are the same size (see Metronome's size-class defragmentation [4]), 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

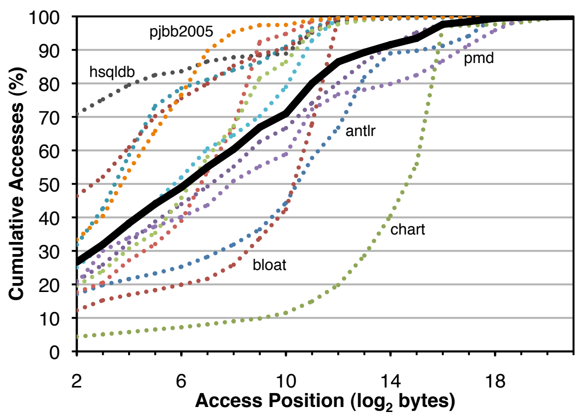


Figure 2. Cumulative distribution of array access positions, faint lines show 12 representative benchmarks (of 19) and solid line is overall average.

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 [18]. 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.

4.3 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.

We instrumented Jikes RVM to gather array size and access characteristics. Figure 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 have the same trend. 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. 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 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 in Figure 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 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 7 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%.

4.4 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 [12]. 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 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 [12], we implement lazy allocation atomically to ensure safety. Section 7.1.3 shows that lazy allocation greatly improves space efficiency for some benchmarks, thereby reducing collector time and improving performance.

4.5 Zero Compression

Chen et al. perform aggressive compression of arraylets at the byte granularity, focusing only on space efficiency [12]. 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 7.1.3 shows that this space saving optimization improves overall garbage collection time and thus total time.

4.6 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, `arraycopy` must notify the garbage collector when reference arrays are copied since the copy may generate new inter-space pointers (such as old-to-young) that the garbage collector must be aware of.

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 7.1.3.

4.7 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 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 improves space usage.

5. Implementation

We now describe key details of our efficient z-ray implementation.

5.1 Run-time Modifications

Z-rays affect 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 [9]. 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 4.5. 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.

```
1 void arrayStore(Address array, int index, int value) {
2   int len = array.length;
3   if (index >= len)
4     throw new ArrayBoundsException();
5   int offset = len * BYTES_IN_INT;
6   array.store(offset, value);
7 }
```

(a) Array store (contiguous array).

```
1  void arrayStore(Address array, int index, int value) {
2    int len = array.length;
3    if (index >= len)
4      throw new ArrayBoundsException();
5    int offset = len * BYTES_IN_INT;
6    int arrayletNum = index / INTS_IN_ARRAYLET;
7    int spineOffset = arrayletNum * BYTES_IN_ADDRESS;
8    Address arraylet = array.loadAddress(spineOffset);
9    offset = offset % ARRAYLET_BYTES;
10   arraylet.store(offset, value);
11 }
```

(b) Array store (conventional arraylet).

```
1  void arrayStore(Address array, int index, int value) {
2    int len = array.length;
3    if (index >= len)
4      throw new ArrayBoundsException();
5    int offset = len * BYTES_IN_INT;
6    if (offset < FIRST_N_BYTES)
7      array.store(offset, value);
8    else
9      arrayletStore(array, offset, value);
10 }
```

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

```
1  @NoInline          // force this code out of line
2  void arrayletStore(Address spine,int offset,int value){
3    int arrayletNum =
4      (offset - FIRST_N_BYTES) / BYTES_IN_ARRAYLET;
5    int spineOffset =
6      FIRST_N_BYTES + arrayletNum * BYTES_IN_ADDRESS;
7    Address arraylet = spine.loadAddress(spineOffset);
8    if (arraylet & SHARING_TAINT_BIT != 0)
9      ...             // atomic copy on write
10   else if (arraylet == ZERO_ARRAYLET)
11     if (value == 0)
12       return;       // nothing to do
13     else
14       ...           // lazy allocation and atomic update
15   offset = (offset - FIRST_N_BYTES) % BYTES_IN_ARRAYLET;
16   arraylet.store(offset, value);
17 }
```

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

Figure 3. Storing a value to a Java **`int`** array.

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 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 3(c)). Otherwise, basic arithmetic (shown in lines 5–9 of Figure 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 3(c) and 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-rays 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. It generates a fixed deterministic optimization plan for each benchmark via profiling [10].

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 [7]. 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 [5]. 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 7.1.3). 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.

5.2 Jikes RVM-Specific Details

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.

5.3 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 raw bytes and Java arrays on the same memory. Instead, all access to this metadata now use a *magic* array type that is not arrayletizable [16]. 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 [23] 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 use 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 cause an immediate segmentation fault (instead of corrupting the heap and manifesting much later), and generate an exception that we can use to track down offending array accesses. We plan to make this valuable debugging tool available with our z-ray implementation.

6. Benchmarks and Methodology

Benchmarks. We use the DaCapo benchmark suite, version 2006-10-MR2 [10], the SPECjvm98 suite, and pjbb2005, which is a variant of SPECjbb2005 [29] 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 2.4GHz Core 2 Duo with 4MB of L2 cache and 2GB of memory. To ensure our approach is applicable across architectures, we also measure it on a 1.6GHz Intel Atom two-way SMT in-order processor with 512KB of L2 cache and 2GB 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

	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 1. Allocation, heap composition, and array access characteristics of each benchmark.

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 [10]. 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 3 and exclude them from the reported minimums, maximums and geometric means.

Benchmark Characterization. Table 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 3(c) and 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.

7. Evaluation

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

7.1 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.

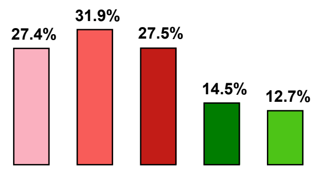
	Naive	Naive A [12]	Naive B [4]	Z-ray	Perf Z-ray
Arraylet Bytes	2^{10}	2^{10}	2^{11}	2^{10}	2^{10}
First-N				2^{12}	2^{12}
Lazy Alloc		✔		✔	✔
Zero Compress				✔	✔
Array Copy				✔	✔
Copy-on-Write				✔	
Overhead 	27.4%	31.9%	27.5%	14.5%	12.7%

Table 2. Overview of arraylet configurations and their overhead.

7.1.1 Z-ray Summary Performance Results

This section summarizes the performance overhead of z-rays and compares to previously published optimizations. Table 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 by Chen et al. [12] and Bacon et al. [4] 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 [12] while Naive B raises the arraylet size [4]. 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 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. [12] (Naive A) and Bacon et al. [4] (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 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, on the whole, the copy-on-write optimization does add 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.

	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 3. Time overhead of Perf Z-ray compared to base system on the Core 2 Duo and Atom (95% confidence intervals 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.

7.1.2 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. 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 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 3 shows that benchmark over-

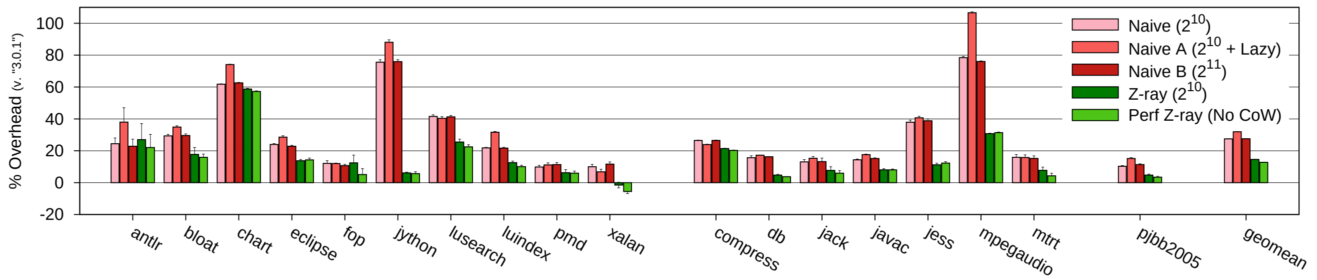


Figure 4. Percentage overhead of Z-ray and Perf Z-ray configurations over a JVM with contiguous arrays, compared to previous optimizations.

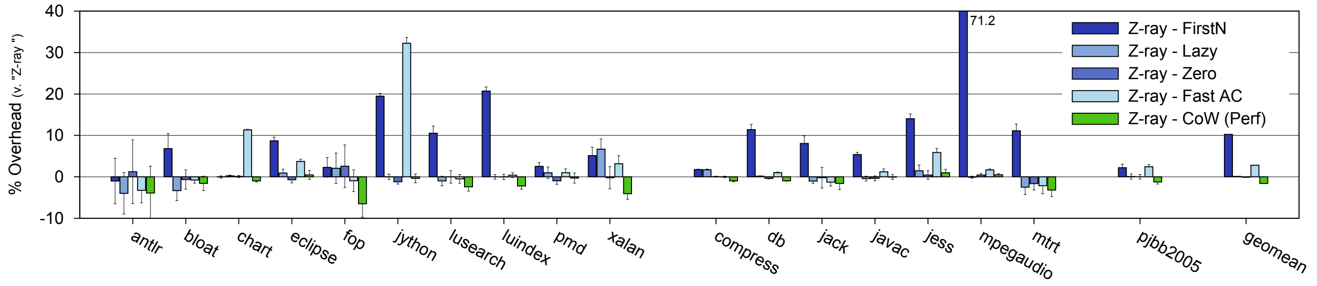


Figure 5. Overhead taking away each optimization from our Z-ray configuration.

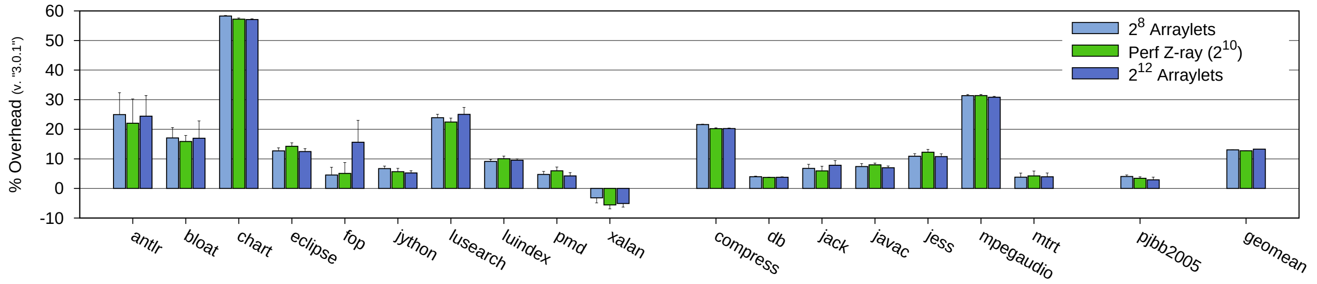


Figure 6. Overhead of Perf Z-ray configuration, varying the number of arraylet bytes.

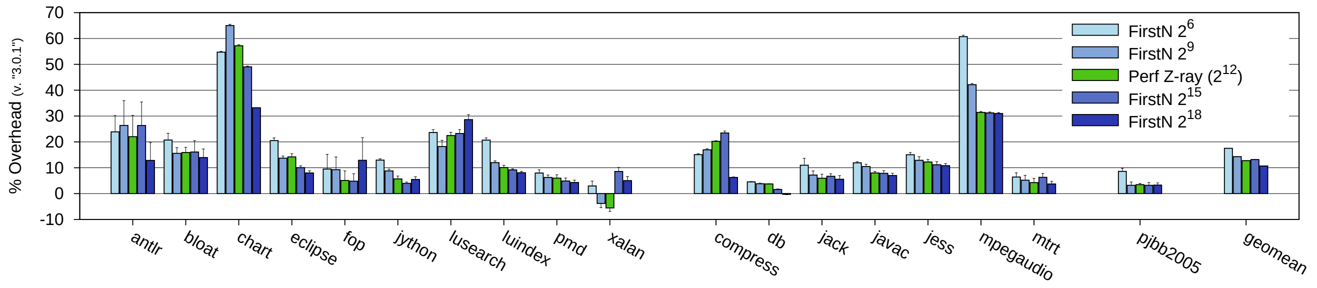


Figure 7. Overhead of Perf Z-ray configuration, varying the number of inlined first *N* bytes.

head 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 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 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 [10]. 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. We do not show the graph here, but we find that z-ray overhead is 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.

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 tabular form). 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.

7.1.3 Efficacy of Individual Optimizations

Figure 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 ≡ Perf Z-ray). A slowdown, or positive overhead, in Figure 5 indicates the utility of a given optimization (without the optimization, z-rays are slower).

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 3(c) and 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). By contrast, lazy allocation adds no overhead to z-rays, on average (see Z-ray − Lazy, Figure 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* (see Table 1).

Figure 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 7.2.1 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.

7.1.4 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 3 shows that a small number of benchmarks suffer significant overheads, and Figure 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 2 and Table 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 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 (Table 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 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 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 shows that z-rays outperform prior work on compress and Figure 5 shows that it is lazy allocation and the first-*N* optimization that help z-ray performance on compress. In Section 7.2.1 we show that compress has significant space savings.

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

7.1.5 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.

	%				Total Heap Footprint	
	Alloc	% Savings			% Array-	%
Benchmark	Large	Lazy	Zero	COW	letizable	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. Effect of space saving optimizations.

Figure 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. 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 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 7.2.1).

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.

7.2 Flexibility

Previous work has demonstrated the flexibility of discontiguous arrays. While we primarily target improving the running-time performance of a general-purpose system in our evaluation, we show here how z-ray optimizations can improve space efficiency. We then discuss the impact of discontiguous arrays on heap fragmentation.

7.2.1 Space Efficiency

One motivation for discontiguous arrays is that they offer additional flexibility that can be used to implement space saving optimizations such as lazy allocation, zero compression, and arraylet copy-on-write. Table 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. [12] 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 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 (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 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 [3]: 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.

7.2.2 Fragmentation

We now briefly discuss how discontiguous arrays and our z-ray implementation affect fragmentation. 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' [4], 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. Consequently, this section offers a qualitative discussion of fragmentation.

Discontiguous arrays in general have benefits for fragmentation, which are well understood in the literature. 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 [4, 6].

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 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 3 and 4.2), which is otherwise common in garbage-collected systems. 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*.

8. Conclusions

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.

References

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

[2] B. Alpern, D. 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. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

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

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

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

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

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

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

[9] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and Realities: The Performance Impact of Garbage Collection. In *Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 25–36, 2004.

[10] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, 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 *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 169–190, 2006.

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

[12] G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, B. Mathiske, and M. Wolczko. Heap compression for memory-constrained Java environments. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 282–301, 2003.

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

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

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

[16] 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.

[17] 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.

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

[19] IBM. Websphere real time. `http://www-01.ibm.com/software/webservers/realtime/`.

[20] K. Ishizaki, M. Kawahito, T. Yasue, M. Takeuchi, T. Ogasawara, T. Suganuma, T. Onodera, H. Komatsu, and T. Nakatani. Design, implementation, and evaluation of optimizations in a just-in-time compiler. In *Java Grande*, pages 119–128, 1999.

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

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

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

[24] F. Pizlo. Private communication, 2010.

[25] F. Pizlo, L. Ziarek, P. Maj, A. Hosking, E. Blanton, and J. Vitek. Schism: Fragmentaton-tolerant real-time garbage collection. In *Programming Language Design and Implementation (PLDI)*, 2010.

[26] 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.

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

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

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

[30] D. 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.

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