Z-Rays: Divide Arrays and Conquer Speed and Flexibility

Jennifer B. Sartor, Stephen M. Blackburn, Daniel Frampton, Martin Hirzel, Kathryn S. McKinley

jbsartor at cs.utexas.edu

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.

We propose *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.

Introduction

Konrad Zuse invented arrays in 1946 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.

Related Work. 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* to achieve real-time guarantees[5,25]. Chen et al. contemporaneously invented arraylets to aggressively compress arrays during collection and decompress on demand for memory-constrained embedded systems[12]. 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]. Applications that use these JVMs include control systems and high frequency stock trading. Thus, although discontiguous arrays are needed for their *flexibility*, which achieves space and time predictability, so far they have sacrificed throughput and time *efficiency*.

We present *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. The collector performs zero-compression by eliminating whole 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. We believe we are the first to implement a copy-on-write optimization, which initially shares whole arraylets that are copied and only copies later upon a write. This study is the first to rigorously evaluate and report Java array properties and their impact 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 by reducing memory write traffic with our compression optimizations. Z-rays performance and flexibility thus broaden their usability to general-purpose systems, making them an attractive optimization tool for language implementation on current and future architectures.

Background

This section briefly discusses our implementation context for discontiguous arrays in Java. Discontiguous array representations are feasible in managed languages because the organization of memory is abstracted away from the programmer. We implement z-rays in Jikes RVM[2], a high performance Java-in-Java virtual machine, but this is not integral to our approach.

Garbage Collection. 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, which are zero-initialized in Java, 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. Our heap uses a standard free-list mature space[9].

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. Java has barriers for array bounds checks, for cast checks on reference array stores, and for the generational collector described above. Java optimizing compilers eliminate provably redundant checks[11,20].

Z-rays: Efficient Discontiguous Arrays

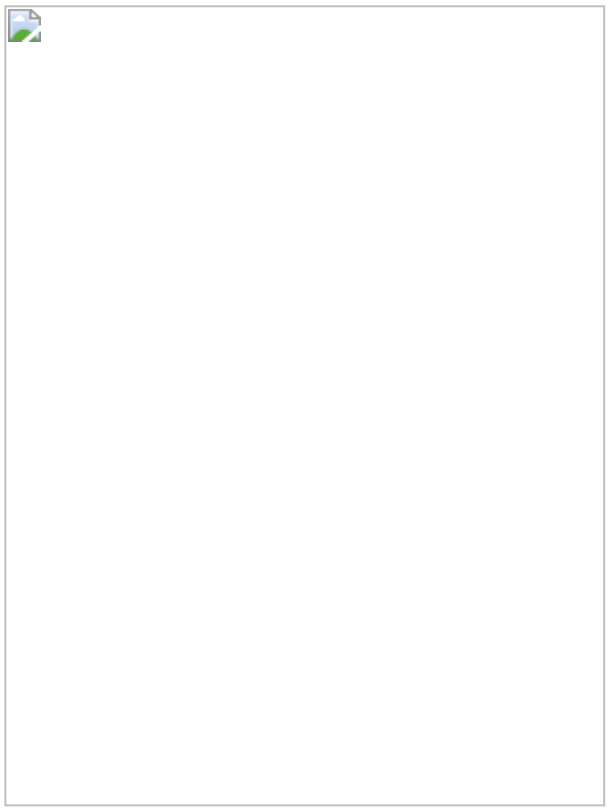
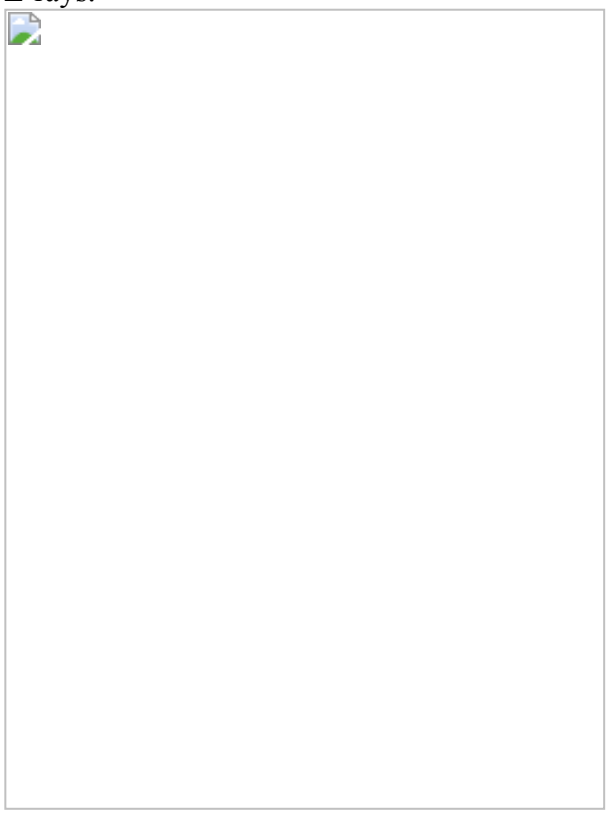


Figure 1: Discontiguous reference arrays divided into a spine pointing to arraylets for prior work and optimized

Z-rays.



This section first describes a basic discontiguous array design which 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.

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. The spine has three parts: (1) A header that encapsulates the object's identity. (2) Indirection pointers to arraylets which store actual elements of the array. (3) Possibly inlined data elements. Spines vary in size. 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, 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. 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). The arraylet size is a global constant, and we explore different values in our results.

Memory Management of Z-rays

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[18]. The base version of Jikes RVM we use has a non-moving large-object space for objects 8KB and larger. We adopt a modified version of (c) for z-rays, reducing fragmentation and avoiding untimely reclamation and expensive copying. We allocate spines, which define array age, into the nursery and arraylets into their own non-moving space. Figure 1 shows the *arraylet space* which is managed efficiently with fixed-sized blocks. Nursery collections trace and promote spines to the old space if they survive, like other objects. The collector requires one liveness bit per arraylet that we maintain in a side data structure. If a spine dies, its corresponding arraylets' liveness bits are cleared, which the allocator inspects to find arraylets available for immediate reuse. This approach limits the memory cost of short-lived and sparsely-populated arrays.

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. Figure 2 shows the cumulative distribution plots for all array accesses for 12 (DaCapo and pseudojbb) 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 access patterns bias towards the beginning. 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.

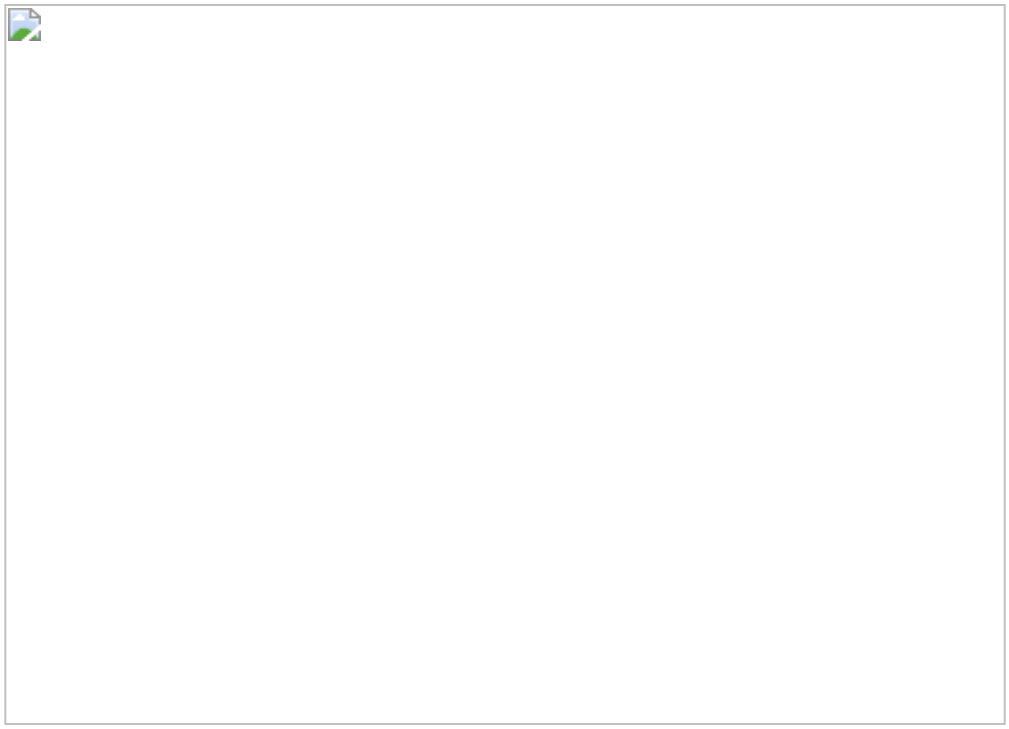


Figure 2: Cumulative distribution of array access positions, faint lines show 12 representative benchmarks and

solid line is overall average.

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. The program directly accesses the first `E = 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. 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 results explore 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%.

Lazy Allocation

We borrow from and modify Chen et al.'s space optimization ideas, including lazy allocation[12]. Because accesses to arraylets go through a level of indirection, the JVM can lazily allocate arraylets upon first write. Unused portions of an over-provisioned or sparsely populated array need never be backed with arraylets, saving space and time. Since Java specifies that all objects are zero-initialized, we create a single immutable global *zero arraylet* (in Figure 1), and all arraylet pointers initially point to the zero arraylet. Arraylets are only instantiated after the first non-zero write to their index range. Whereas Chen et al. do not describe a thread-safe implementation[12], we implement lazy allocation atomically to ensure safety. Results show that lazy allocation greatly improves space efficiency for some benchmarks, thereby reducing collector time and improving performance.

Zero Compression

For space efficiency, Chen et al. perform aggressive compression of arraylets at the byte granularity, forcing expensive decompression[12]. 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. Results show that this space saving optimization which also doesn't add overhead improves overall garbage collection time and thus total time.

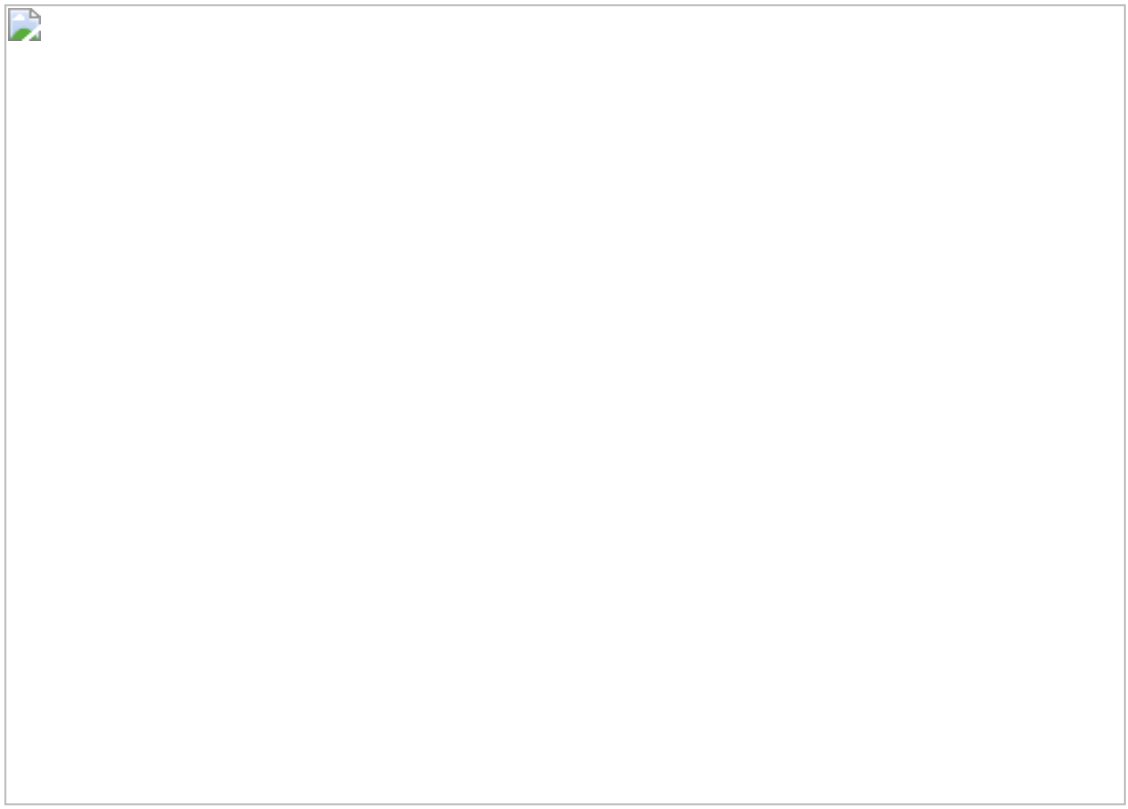
Fast Array Copy

The Java language includes an explicit `arraycopy` API to support efficient copying of arrays. The standard implementation of `arraycopy` uses fast, low-level byte copy instructions for non-overlapping ranges, and otherwise uses element-by-element assignments. Discontiguous arrays complicate the optimization of `arraycopy` because copying must respect arraylet boundaries. We 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, this optimization is beneficial. Results show that omitting fast array copy degrades performance of z-rays by on average 2.8%.

Copy-on-Write

Z-rays introduce a copy-on-write (COW) optimization for arrays. When an `arraycopy` range is 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 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 application 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 (on average 1.8%), but improves space usage.

Benchmarks and Methodology



Experimental Design

For comprehensive methodology details and benchmark characterization, see [J.B. Sartor, S.M. Blackburn, D. Frampton, M. Hirzel, and K.S. McKinley. Z-Rays: Divide Arrays and Conquer Speed and Flexibility. *ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation.* Toronto, Canada, June 2010.] We use the DaCapo benchmark suite[10], the SPECjvm98 suite, and `pjbb2005`, which is a variant of `SPEC-jbb-2005`[29]. All results are presented as a percentage overhead over the 3.0.1 release of the Jikes Research Virtual Machine with a contiguous array implementation, which we modified for our z-ray implementation. We use adaptive experimental compilation methodology[10], and a heap size of 2 times the minimum required for each individual benchmark as our default, reflecting moderate heap pressure.

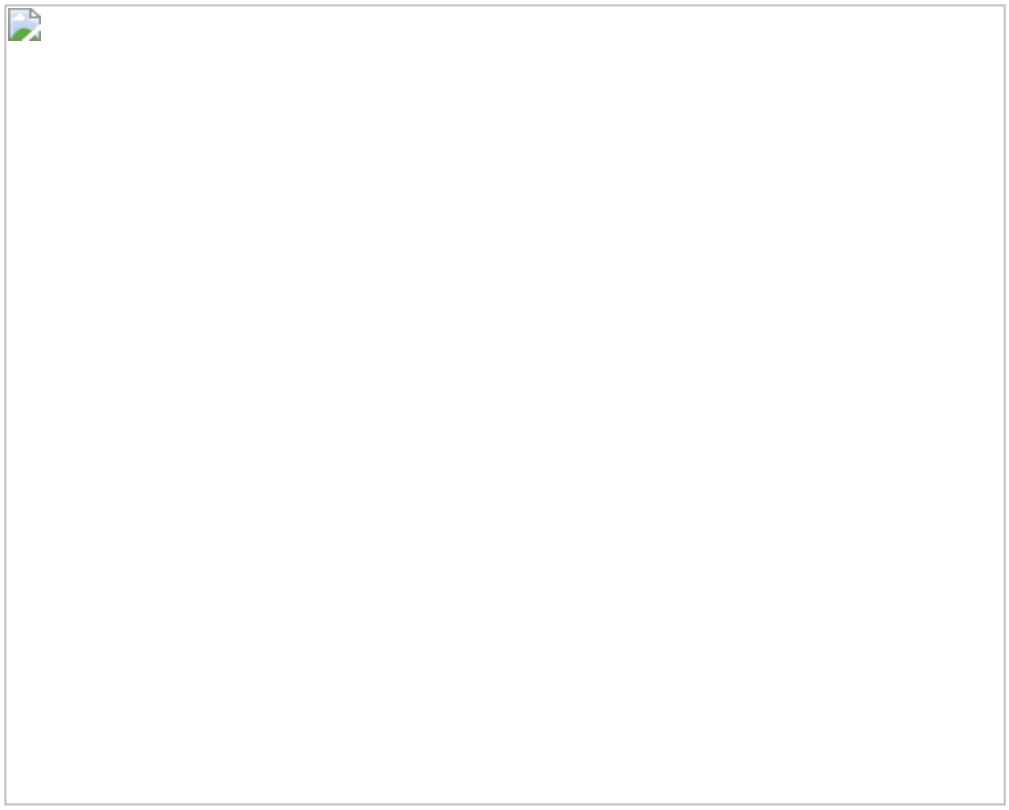
Benchmark Characterization.

We characterize our benchmarks in terms of allocation, heap composition, array access and array copy patterns in Table 1. On average, 56% of all allocation is due to arrays, 40% for primitive arrays. We measured the distribution of array read and write accesses over the *fast* (within first-N) and *slow* paths of barriers to show the potential of first-N. There are a few outliers, like `chart`, which frequently exercise slow paths. Overall, however, 91% of all accesses go through the fast path, thereby enabling first-N to avoid those indirections and greatly reduce overhead.

Results

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

Efficiency

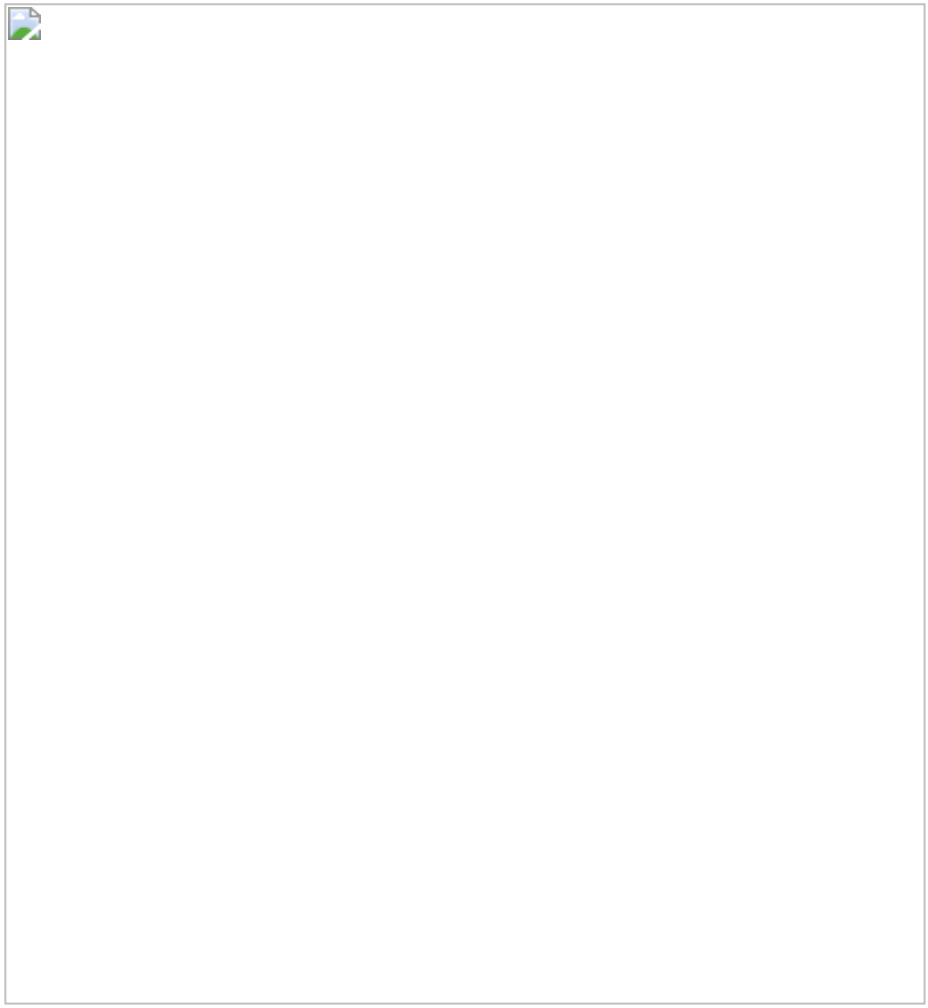


Z-ray Summary Performance Results

This section summarizes the performance overhead of z-rays, comparing with the efficacy of previously described discontiguous array optimizations, but we do not have a direct comparison to prior implementations. Table 2 shows the optimizations and key parameters used in the five configurations 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, adding lazying allocation and raising the arraylet size. The Z-ray configuration includes all optimizations, with a 2^{12} byte first-N. The Perf Z-ray configuration is the best performing configuration, disabling copy-on-write from the Z-ray configuration.

Figure 3: Percentage overhead of Z-ray and Perf Z-ray configurations over a JVM with contiguous arrays, compared to previous optimizations.

As Table 2 shows average overheads, Figure 3 compares the performance of z-rays against previously published optimizations for all benchmarks. These numbers demonstrate that both Z-ray and Perf Z-ray comprehensively outperform prior work. The configurations Naive-A and Naive-B have average overheads of 32% and 27% respectively 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, has an average overhead of 14.5%, slightly slower than our best-performing system at 12.7%.



Performance Breakdown and Architecture Variations

Table 3 summarizes our results in terms of average time overheads relative to an unmodified Jikes RVM 3.0.1 system using the Perf Z-ray configuration. We show main results on the a 2.4GHz Core2Duo in the 'C2D' column. We calculate and plot 95% confidence intervals, and statistically insignificant results are grayed out. Note that garbage collection exhibits chaotic performance characteristics in general with the adaptive compiler because perturbations can affect the volume of data allocated and the timing of collections[10].

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 `pjbb`. Because `eclipse`, `xalan`, and `compress` have many arrays larger than first-N, lazy allocation is particularly effective at reducing space consumption which improves garbage collection time. Table 3 shows that benchmark overhead comes primarily from primitive (`Prim.') discontiguous arrays.

Table 3 shows that most of the overhead of z-rays is due to the mutator (aka application). We see that `chart`---which 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. Though collector performance varies significantly, `xalan` improves collection time by 56% and `javac` degrades by 4%, on average reducing collection time by 11.3% (for statistically significant results).

To ensure our approach is applicable across architectures, we also measure on a 1.6GHz Intel Atom two-way SMT low power, in-order processor targeted at portable devices. The 'Atom' column of Table 3 shows that the Perf Z-ray overhead increases to 20.2% (from Core2Duo's 12.7%).

In summary, z-ray performance varies significantly across benchmarks and overheads are overwhelmingly due to the mutator, especially for primitive arrays.

Figure 4: Overhead taking away each optimization from our Z-ray configuration.

Figure 5: Overhead of Perf Z-ray configuration, varying the number of arraylet bytes.

Figure 6: Overhead of Perf Z-ray configuration, varying the number of inlined first N bytes.


Efficacy of Individual Optimizations

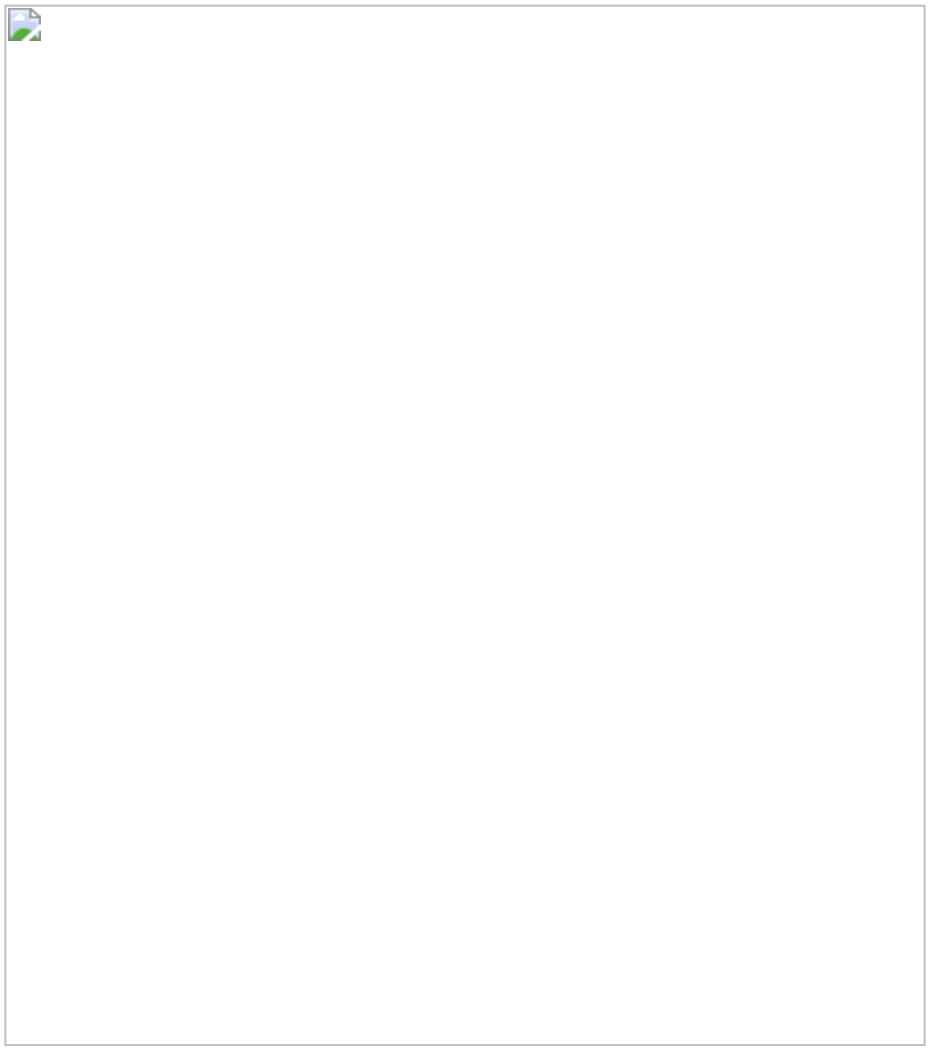
Figure 4 shows the effect of disabling each of our 5 optimizations with respect to the Z-ray configuration. A slowdown, or positive overhead indicates the utility of a given optimization (without the optimization, z-rays are slower). Omitting first-N comes at the most significant performance cost, increasing the overhead by up to 71% in the worst case and 10% on average. Inlining the first-N bytes is key to reducing overhead and central to our approach. Furthermore, because first-N moves arraylet accesses off the critical path, other optimizations (such as lazy allocation) become profitable.

Sensitivity to Configuration Parameters

We explore how performance is affected by varying our two key configuration parameters. Figure 6 shows the effect of altering the number of first-N bytes inlined with the arraylet size held constant at 2^{10} bytes. Figure 5 shows the effect of varying arraylet size, with the number of inlined first N bytes constant at 2^{12}. These results show that it is possible to significantly vary both parameters while maintaining overheads at reasonable levels.

Flexibility

Space Efficiency



Discontiguous arrays offer flexibility because they can be used for 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. First we present the fraction of allocated bytes in large arrays (greater than first-N), then show the efficacy of each space optimization. Lazy allocation ('Lazy') avoids allocating on average 20% of the space consumed by large arrays, greatly saving 75.6% in `xalan`. On average, zero compression ('Zero') reduces live heap arraylets by 14.2%. Copy-on-write ('COW') avoids copying 18.2% of bytes beyond first-N.

The final two columns of Table 4 are measured by taking heap snapshots after every 1MB of allocation (like 'Zero' above). The `% Arrayletizable' column shows that on average 10.1% of the live heap is consumed by bytes beyond first-N *without the three* space saving optimizations. `% Saved' shows savings as a percentage of total heap footprint will all three optimizations. In two benchmarks, we increase heap space slightly. However, `xalan` and `compress` save 25% and 49% of the heap, respectively, which is 92% and 81% of arrayletized bytes. Overall z-rays save about 6% of the heap. In summary, we find that each of our arraylet-granularity space saving optimizations yields savings, and that for some benchmarks, these savings are substantial.

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. We introduce 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 our optimizations reduce overhead significantly and save space, on average reducing the heap size 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. 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 for general-purpose systems.