Accordion Arrays: Selective Compression of Unicode Arrays in Java

Craig Zilles
Department of Computer Science, University of Illinois at Urbana-Champaign
zilles@cs.uiuc.edu

Abstract

In this work, we present **accordion arrays**, *a straightforward and effective memory compression technique targeting Unicode-based character arrays. In many non-numeric Java programs, character arrays represent a significant fraction (30-40% on average) of the heap memory allocated. In many locales, most, but not all, of those arrays consist entirely of characters whose top bytes are zeros, and, hence, can be stored as byte arrays without loss of information.*

In order to get the almost factor of two compression rate for character vectors, two challenges must be overcome: 1) all code that reads and writes character vectors must dynamically determine which kind of array is being accessed and perform byte or character loads/stores as appropriate, and 2) compressed vectors must be dynamically inflated when an incompressible character is written. We demonstrate how these challenges can be overcome with minimal space and execution time overhead, resulting in an average speedup of 2% across our benchmark suite, with individual speedups as high as 8%.

Categories and Subject Descriptors D.3 [*PROGRAMMING LANGUAGES*]: Code generation

General Terms Performance

Keywords Array, Character, Compression, Java, Memory Management, Polymorphism, Unicode

1. Introduction

With memory costs representing a significant fraction of server costs, it is important to identify optimizations that target memory efficiency. Using less memory has a number of processor-observable advantages, including smaller data working sets, which will fit better in caches and translation look-aside buffers (TLBs), as well as lower memory system bandwidth requirements. In addition, in garbage collected languages, a reduction in the rate at which memory is allocated reduces the frequency at which garbage collection needs to be performed.

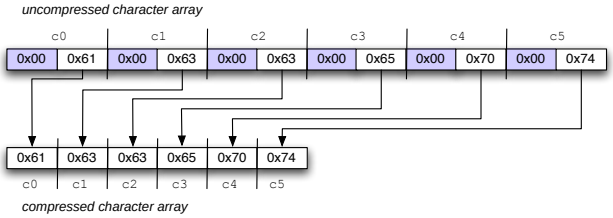

Figure 1. Accordion array compression. We can halve the memory usage of Unicode character arrays that consist entirely of characters whose top bytes are all 0's by storing only the characters' bottom bytes in a byte array.

In this work, we focus on the memory usage resulting from character arrays in managed languages; specifically, we focus on Java. Character arrays represent an important target for optimization because they can account for a significant fraction of a program's heap memory allocation; we find in a suite of non-numeric Java programs character arrays are responsible for 30-40% of allocated heap memory on average. One of the reasons for this large fraction is the use of 16-bit characters. The use of 16-bit Unicode encodings (UCS-2) for characters in modern languages facilitates internationalization, because this standard encoding enables representing most of the character sets in widespread use in the world. In most of five continents, however, an 8-bit subset of Unicode is sufficient to represent most text. In all of the Java programs we studied, we find that over 99% of character arrays consist entirely of 8-bit characters.

While the benefits of using Unicode are unquestionable, it generally represents a significant space inefficiency, one that can be easily eliminated by compression. The compression technique must however serve the usage model of how character arrays are accessed. Given that Java provides random access to characters within a character array (through the character load (`caload`) and store (`castore`) bytecodes), it is important to provide the ability to efficiently index into the middle of a compressed array and read/write a character. This is not a property generally provided by a variable-length encoding scheme (*e.g.*, Huffman or UTF-8 encoding).

ISMM'07, October 21–22, 2007, Montréal, Québec, Canada.

In this paper, we present the idea of *accordion arrays*, a straight-forward and effective memory compression technique targeting Unicode character arrays in managed languages. In principle, the technique is quite simple: character arrays consisting entirely of 8-bit characters are squeezed (much like bellows of an accordion) so that they can be stored in byte (8-bit) arrays (as shown in Figure 1). This encoding achieves a halving of memory allocated for the characters and retains the ability to efficiently read and write arbitrary characters in the vector: the nth character is stored at an n byte offset from the beginning of the string and can be read or written with *byte* load or store instructions, respectively.

Because most character array objects are short lived, to get a significant benefit from this compression, we need to directly allocate arrays in compressed form. In general, however, we do not know until after allocation whether an array will need to hold an incompressible (>255) character. Thus, we must *speculatively* allocate arrays in compressed form. If, after allocation, we discover that we need to write an incompressible character to a compressed array, we must first *inflate*—in the same sense that a thin lock [4] is inflated—the compressed array. Frequently this inflation can be performed in place, as the array is frequently the most recently allocated object, but, in rare circumstances, inflation must be accomplished by allocating a new uncompressed array. When this is necessary, we must install within the original array a *forwarding pointer* to the new array, to permit future accesses to the original array to find the new copy

To benefit from compressible character arrays while maintaining the ability to store arbitrary Unicode characters in any character array, all types of character arrays (`compressed`, `uncompressed`, and `inflated`) must peacefully co-exist within the managed runtime. Because each type of array requires a different sequence of instructions to index into the array and load or store an element, any code that accesses a character array has to dynamically dispatch to the appropriate routine based on the type of the array encountered. In effect, we are making the character array primitive polymorphic.

This dynamic dispatch has the potential to add overhead that would exceed any benefits resulting from improved memory efficiency, but we find that efficient implementations—those with negligible overhead on modern dynamically-scheduled superscalar processors—of accordion arrays are possible. There are three main factors that contribute to this result:

1. Accesses to character arrays are disproportionately uncommon relative to the amount of heap memory allocated to character arrays: while character arrays represent 30-40% of heap memory allocated, access to them only account for only 1-3% of the bytecodes executed.
2. Dynamic dispatch branches are quite predictable and inflations are extremely rare. The branches are predictable because they are highly biased—the overwhelming majority (>99%) of arrays are compressed—and there is locality (both in code location and time) in the accesses to non-compressed arrays. In practice the number of inflations is much less than the number of uncompressed arrays, because most of the uncompressed arrays are allocated on behalf of Strings, whose content is known at allocation time.
3. The extra instructions that dynamic dispatch adds are almost entirely predictable branches and ALU operations that are off the program's critical path. On modern dynamically-scheduled superscalar machines, these instructions can often be executed on what would otherwise be idle functional units, generally resulting in negligible overhead.

With small execution overhead, the benefits of improved memory efficiency and reduced garbage collection frequency translate into a reduction of overall execution time or an improvement of throughput. We find on a collection of DaCapo benchmarks and SPECjbb2005, that accordion arrays achieve an average speedup of 2%, with speedups of 5-8% for the most memory intensive programs.

This paper makes three contributions, each of which makes up a major section in the paper:

1. We extensively characterize the usage of Unicode character arrays in representative non-numeric Java programs (Section 2).
2. We describe the design and an implementation of accordion arrays that is efficient in both space and time, detailing its components in the code generator, run-time, and garbage collector (Section 3).
3. We characterize and analyze the performance of accordion arrays implemented in the Harmony DRLVM JVM (Section 4).

In Sections 5 and 6, we discuss related work and conclude, respectively.

2. Motivation

In this section, we demonstrate that, in a number of Java programs, character (char) arrays are a significant fraction of heap usage and very few of the Unicode characters in those arrays require more than the (8-bit) ISO-8859-1 representation. We find that those arrays that do include full (16-bit) Unicode characters are often allocated by a Java String constructor, but character arrays allocated by String constructors represent only a small fraction of all char arrays allocated. In addition, we find that a small number of non-String allocated char arrays include 16-bit characters, but most of those arrays are short lived.

The Heap is Dominated by Char Vectors: For many non-numerical applications of the Java language, arrays of char-

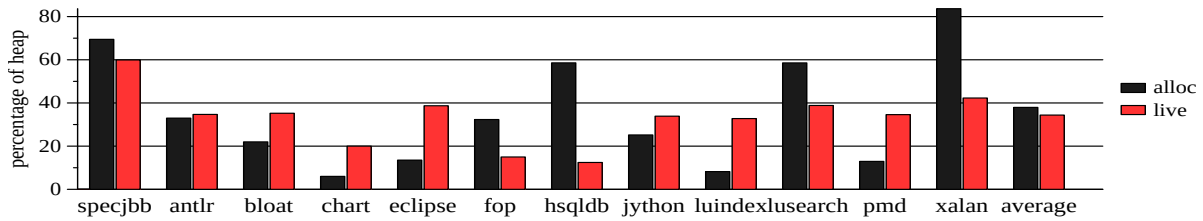

Figure 2. Fraction of Java heap usage attributed to character arrays. Data shown both for the average fraction of live heap at garbage collections (**live**), and of all heap memory allocated (**alloc**).

acters represent a significant fraction of the program's heap memory usage. Figure 2 shows that char arrays represent 38 percent of the total heap memory allocated and 35 percent of the average heap memory live at a garbage collection in the DaCapo benchmark suite [5] and SPECjbb2005. Char arrays are responsible for the most heap allocation of any single type in all of the benchmarks except three (`jython`, `pmd`, and `luindex`), representing more than half of the heap allocation in four benchmarks. At a garbage collection, char arrays are responsible for the most live heap memory of any single type in all benchmarks except one (`hsqldb`). When it comes to heap memory allocation, char vectors are *a*, if not *the*, common case.

The Prevalence of ISO-8859-1 in Unicode: To support internationalization, Java uses 16-bit Unicode as the native representation for its Strings and, therefore, its character arrays. As Unicode was developed significantly after Western-centric character coding schemes (*e.g.*, ASCII and ISO-8859-1), Unicode designers chose to make Unicode a super-set of these coding schemes to facilitate adoption. Unicode's first 256 encodings are identical to those of the 8-bit ISO-8859-1, which consist of the 7-bit ASCII encodings extended to include most characters necessary for representing any Latin-alphabet text. As a result, these 8-bit encodings are sufficient for storing much of the text found in the Americas, Western Europe, Oceania, and much of Africa.

As such, it should not be surprising that for many Java executions, char arrays are predominantly populated with character values of 255 or less. In all of the benchmarks we evaluated, char arrays that included characters outside the ISO-8859-1 encodings represented 1% or less of heap memory used by char arrays, as shown in Figure 3(a). In general, we find that an array either has no 16-bit characters or has many of them.

Thus, if we can allocate the ISO-8859-1 char arrays into byte arrays rather than short (16-bit) arrays, we can approximately halve the heap memory allocation from char arrays. This, in turn, will result in a 20% reduction (on average) in overall heap memory allocation rate (in MB per bytecode executed), with as much as a 30-40% reduction in four of the 10 benchmarks.

Characterizing the Arrays with Full (16-bit) Unicode Characters: Most character arrays are allocated either as

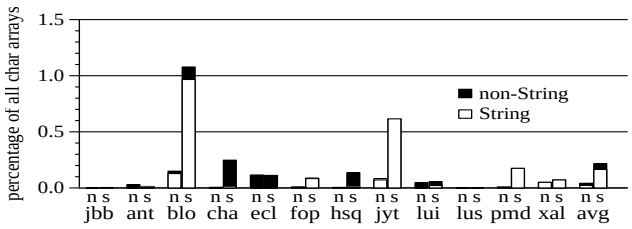

Figure 3. Incompressible character arrays are uncommon. Fraction of character arrays containing incompressible characters; data shown for both **n***umber* and **s***ize* of the arrays. String constructors are responsible for the majority of incompressible characters.

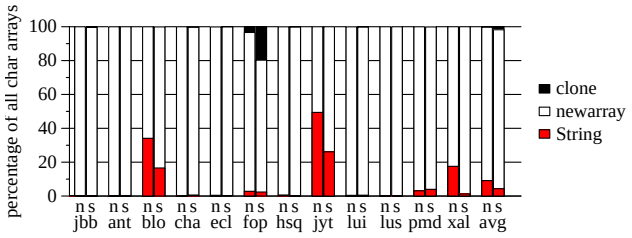

Figure 4. Most character arrays are allocated by the standard `newarray` allocation path While String constructors are responsible for the majority of incompressible arrays, they are responsible for only a fraction of the character arrays allocated overall, meaning that it is important to compress non-String constructor-allocated character arrays. Data shown for both **n***umber* and **s***ize* of the arrays.

part of Java String objects, which contain a reference to an immutable character array, or in the process of their construction. There are, however, four ways in which character arrays are allocated and they have significantly different behaviors with respect to allocating incompressible characters, as shown in Figure 3.

One important path is in the String object constructor (when a character array cannot be shared with another existing object), which account for a disproportionate fraction of the incompressible arrays. While String constructors account for about 10% of the character arrays allocated (less than 5% of the space), they are the source for more than half of the incompressible character arrays. That String construc-

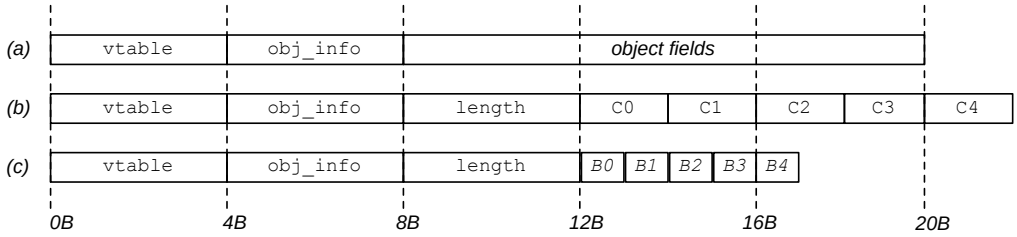

Figure 5. Java object layouts. a) 8-byte object header that is part of all objects in Harmony, b) an array object in Harmony that includes an array length field followed by the array data, c) a byte array object.

tors are the predominant source of 16-bit characters is desirable, because a String's contents are available at the time the char array is allocated (provided as a UTF-8 or Unicode array passed into the String's constructor). By inspecting the String's intended contents, we can know at allocation time whether the char array will contain 16-bit characters and allocate the array appropriately. Furthermore, because Java Strings are immutable, we know that the program will never introduce a 16-bit character into an array that previously did not contain one.

The second important path for character allocation is the generic heap allocation path, which is the most important from the perspective of getting benefit from compression. This allocation path is the one used to implement Java bytecode `newarray`, which results from common usage of `StringBuffer` and `StringBuilder`. It is responsible for the majority of character array memory allocation in all benchmarks, accounting for more than 95% in seven of the ten (as shown in Figure 4). In contrast to Strings, when char arrays are allocated by the generic array allocator, we cannot, in general, know whether it will be used to hold an incompressible character. For the programs considered, a very small fraction (less than 0.1%), of the char arrays will need to hold 16-bit characters, which will require us to *inflate*—dynamically reallocate so they can support 16-bit characters—those arrays. At the end of this section, we include a brief characterization of those arrays that require inflation.

The third and final important character allocation path is the through the `clone` method that character arrays inherit from the Java base type, `Object`. This path accounts for a substantial fraction of allocations only in the benchmark `fop` and allocates no arrays with incompressible characters in our benchmarks. Nevertheless, the `clone` method has an advantage similar to String allocation in deciding whether a compressed or uncompressed array should be allocated. Specifically, we can look at the type of the array that we are cloning and allocate an array of that type. A fourth allocation path, through the Java Native Interface (JNI), exists but was not exercised by our benchmarks.

3. Design and Implementation

In this section, we describe the changes necessary to implement accordion arrays in a Java virtual machine. For clarity of exposition, when discussing issues that depend on the specifics of a particular JVM implementation, we will focus on the implementation in the 32-bit x86 version of the Apache Harmony Dynamic Runtime Layer Virtual Machine (DRLVM) [10], to which we will refer simply as Harmony. We begin this section providing some background details on the implementation of Java and Harmony in Section 3.1. We then describe the high-level design of accordion arrays in Section 3.2. Then, we describe the changes made in each of the memory allocator (Section 3.3), code generator (3.4), garbage collector (3.6), and other virtual machine functions (3.7).

3.1 Background

In Java, the fundamental unit of data structure is the object. Every object includes a (JVM-specific) object header that generally contains: i) a pointer to a virtual function dispatch table (`vtable`), and ii) an object info (`obj_info`) field used internally by the JVM for synchronization, garbage collection, and tracking object hashes. In Harmony each of these fields is 4B large (as shown in Figure 5(a)). In addition to synchronization, Harmony uses the `obj_info` field to store relocation information during garbage collection (GC). The rest of an object's fields are placed in memory after the object header (*i.e.*, starting 8B from the beginning of the object). Object sizes in Harmony are rounded up to the closest 4B for reasons of memory alignment and garbage collector implementation.

Arrays are objects that are implemented with two fields: `length` and `data`. In Harmony, the length field is stored as a 4B integer (as shown in Figure 5(b)) and the array's data field stored at a 12-byte offset from the beginning of the object.

There are four fundamental operations that can be performed on arrays: i) allocation, ii) get length, iii) load element, and iv) store element, which correspond to the `newarray`, `arraylength`, `*aload`, and `*astore` bytecodes, respectively. When arrays are allocated, the length must be specified; if the programmer later desires a longer array, a new array must be allocated, effectively making the length field read-only after allocation. The `arraylength`

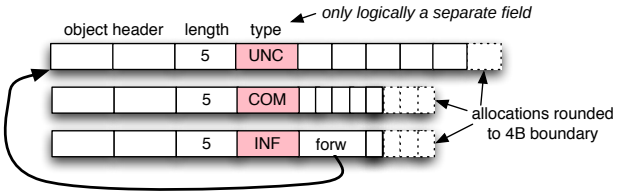

Figure 6. The three types of accordion arrays: `uncompressed` (UNC), `compressed` (COM), and `inflated` (INF). Logically, the type is stored as a field of an accordion array, but we encode it by stealing two bits from the array length field. The space where data was stored is used by `inflated` arrays to store a **forwarding** `pointer` to an `uncompressed` array where the data is now stored.

operation, whose bytecode can be used with any type of array, performs a null check—checking that an array does in fact exist—and returns the value in the object's `length` field. The element load and store operations must perform a null check and an array bounds check prior to completion of the load or store; Java has separate bytecodes for each type of array (*e.g.*, `caload` and `castore` for loading from and storing to char arrays). Java also provides a standard String class, which includes an immutable char array as a data member.

In addition to the standard bytecode execution path, Java provides the Java Native Interface (JNI) that enables non-Java code to inspect, modify, and create Java objects and to call Java methods and be called from them. To offer JVMs flexibility in their internal object representation of Java objects, the JNI does not provide direct access to object internals, but rather an interface for requesting Java data items being returned as native types that in turn can be inspected or modified and written back into Java objects.

Harmony's standard garbage collector (`gc_cc`) is a stop-the-world, mark-sweep collector that dynamically selects between copying and compacting. Regions of heap are allocated as needed. During collection, each region is allocated an associated card table, which stores a single bit for each 4-byte chunk of heap. These bits are set for live objects during the marking phase of the garbage collector. The sliding compaction phase is done with a single pass through the objects where objects are assigned a new location, pointers pointing to the object are updated, and the object is copied to its new location.

3.2 High-level Design

There are three basic concepts involved in the implementation of accordion arrays: i) the three types of arrays: compressed, uncompressed, and inflated, ii) the notion of inflation and the level of indirection it introduces, and iii) the dynamic dispatch necessary when accessing a character array. We discuss each of these first in abstract terms and then discuss how they can be encoded efficiently.

The first two types are rather straight forward: an *uncompressed* array is like a standard Java character array; a *compressed* array is similar to a Java byte array (*i.e.*, 8-bit elements), except its vtable pointer still points to the vtable for character arrays. Obviously, since reading and writing elements of these two types require different instruction sequences, we must be able to distinguish which type an array is. Thus, we logically introduce a new field into the object (as shown in Figure 6) that holds this type.

The third type is required due to the speculative nature of this optimization. Since non-String allocated character arrays (which accounts for most of them) are mutable, we need to guess at allocation time whether a given array will need to hold incompressible characters during its lifetime. If we incorrectly guess that an array will be compressible, we will need to reallocate the array (as an uncompressed one) and copy over the array's data; we refer to this process as *inflation*. Once an array has been inflated, incompressible characters can be written into it freely.

The main difficulty with inflation is that an arbitrary number of references may exist to the uninflated copy of the array (which is now stale) and we need to redirect any accesses through these references to the new copy of the array. Because the set of referring objects is not maintained, we cannot efficiently update those references at the time of the inflation. Instead, we create a third state for arrays, *inflated*, that indicates that an array no longer contains valid data. Instead, inflated arrays (as shown in Figure 6) store a forwarding pointer that indicates the location of the inflated copy.

When code wants to read or write an element of an accordion array, it must dynamically dispatch (based on the type) to code that can operate on that type of array. Flow graphs for character load and store operations are shown in Figure 7. For both uncompressed and inflated types, a character array load operation is implemented with a 16-bit load, but inflated arrays must first de-reference the forwarding pointer to get the array to load from. Compressed arrays simply require 8-bit loads. The store case is structurally equivalent to loads for uncompressed and inflated types, but, for compressed arrays, we need to test if the character being stored is compressible. If it is not, we need to first inflate the array before completing a 16-bit store.

Encoding: While Figure 6 shows the type as a separate object field, it is desirable, for performance reasons, to store the type of the array somewhere that i) does not increase object size, and ii) avoids introducing an additional load instruction into every array access sequence. For these reasons, we encode the array type in the uppermost two bits of the array length field; the length field must be loaded by array bounds checks, one of which (or an explicit load of the array length) must dominate every array access. By stealing these bits we prevent arrays of 2^{30} elements or larger from being allocated on a 32-bit machine (*i.e.*, what would be minimally one-quarter of the process's user-mode virtual memory). A check that ensures this property holds is performed at array

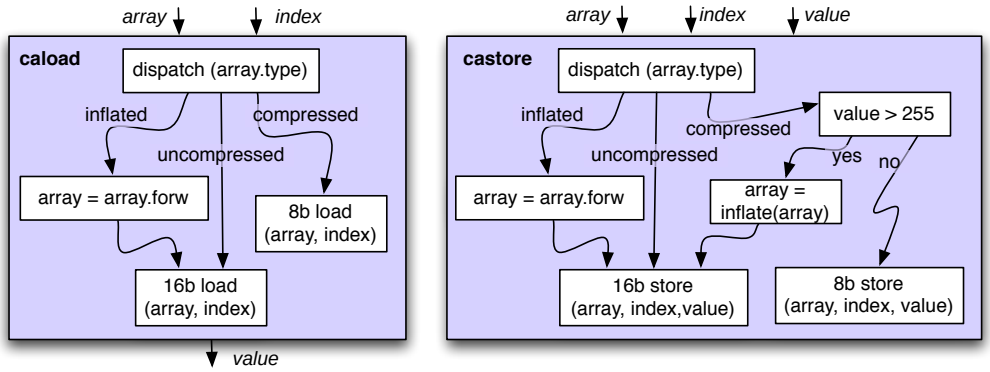

Figure 7. Implementing character array loads and stores (`caload` and `castore`) for accordion arrays requires dynamic dispatch based on whether the array is `compressed`, `uncompressed`, or has been `inflated`.

allocation time. By assigning the `compressed` case the encoding `00` and the other two cases the encodings `10`, and `11`, we can isolate the common case (`compressed`) with a single conditional branch that checks whether the length field is positive or negative.

We store the forwarding pointer for inflated arrays at the same place where the first elements of the array's data would be stored. Because Harmony rounds heap allocations up to 4B boundaries, we are guaranteed to have room for this 4B pointer for arrays of size 1 or greater; arrays of size 0 (which **are** allocated) should never need to be inflated.

3.3 Modifications to the Memory Allocator

Based on the data presented in Section 2 that showed an overwhelming majority of character arrays are compressible, a reasonable policy for those workloads is to, by default, allocate every character array as compressed. Applications running in East Asian locales, in contrast, will likely find a substantial fraction of character arrays containing characters with values greater than 255. While it is likely quite feasible to build a system that adaptively disables the use of accordion arrays for such executions, doing so is beyond the scope of this paper.

Without much work, we can actually do quite a bit better than a default policy of compress everything or compress nothing. Using information from the run-time, we can allocate compressed arrays unless we can easily know at allocation time that an uncompressed array will be needed (as we can for Strings and the `Object.clone()` method). This requires no extra work in the standard array allocator, if we define the default character size (for computing the object size) to be 8 bits and use the 00 encoding for type.

When allocating a character array in a String constructor or as a result of `Object.clone()`, we make the compression decision based on program data. For `clone`, we use the type of the array being cloned, which adds little overhead since the array length field needs to be read anyway. For Strings, we need to inspect the characters that will be written into the array. This requires almost no overhead for Strings created from UTF-8 arrays, as such arrays already need to be scanned for multi-byte characters to compute the number of Java Unicode characters they contain. When Strings are created from Unicode arrays, however, an additional step of scanning the array for compressible characters is performed[1]. In either case, when uncompressed arrays are allocated, we need to compute the array's size with 16-bit characters and correctly annotate the length field with the array's type.

3.4 Modifications to the Code Generator

Because most of the interactions with accordion arrays will be through the generated code, it is important that this code can be implemented efficiently. For the discussion that follows, we focus on the store case (`castore`) because it comprises a superset of the issues encountered by `caload`. In Figure 8, we show the code to perform a store on a Unicode character array and a byte array, which corresponds to the case of a compressed character array. To build the code required for an accordion array store, we have to add code to dispatch based on type, for de-referencing the forwarding pointer for inflated arrays, and for inflating compressed arrays on demand.

For efficient dispatch, we use a series of conditional branches. Since the dispatch is based on type, we must first (after the NULL check) load the array length field, which contains the type specifier. Because the `compressed` type is the most frequent, we use the first branch to isolate that case, so there is only one dispatch branch on that path (as shown in Figure 9). After performing the bounds check, the compressed path must check if the value to be stored is less than 256. If so, it performs the store. Otherwise, it branches to a call to the inflation method described in Section 3.3.

A second dispatch branch is required to select between paths for `uncompressed` and `inflated` types. Inflated types de-reference the forwarding pointer to get a new ar-

[1] This operation could be efficiently performed using SIMD vector instructions on many architectures, but we have not yet implemented this optimization

```
castore(16b array)                          castore(8b array)
beq   rbase, 0, A                           beq   rbase, 0, A                 # Null reference check
lw    rlength, 8(rbase)                     lw    rlength, 8(rbase)           # load length
bge   rindex, rlength, B                    bge   rindex, rlength, B          # Bounds check
sll   rindex2, rindex, 1                                                      # scale index
add   raddr, rbase, r_index2                add   raddr, rbase, r_index       # base + scaled index
sh    rchar, 12(raddr)                      sb    rchar, 12(raddr)            # store (16/8b) element
```

(a) *(b)*

Figure 8. Example code for `castore` in pseudo assembly. a) code for writing to a standard Unicode (16b) array, b) corresponding code required for storing to byte (8b) arrays. The primary differences are the lack of a shift operation and a different sized store in the second version.

```
castore:  lw     rlength, 8(rbase)                 #  1 load length field (w/type)
          blt    rlength, 0, uncomp1               #  2 branch if not compressed
compres:  bge    rindex, rlength, out-of-bounds    #  3 perform bounds check
          bgt    rchar, 255, inflate               #  4 char too large? need to inflate
          add    rtmp, rbase, rindex               #  5 add base and index
          sb     rchar, 12(rtmp)                   #  6 perform 8b store
          j      done
inflate:  call   inflate_char_array                #  7 call VM to inflate & copy
          move   rbase, r_new_array                #  8 put return value in rbase
          j      uncomp3                           #  9 complete 16b store
uncomp1:  bgt    rlength, 0xbfffffff, uncomp2      # 10 branch if not inflated
indirec:  lw     rbase, 12(rbase)                  # 11 get address of inflated version
uncomp2:  and    rlength, rlength, 0x3fffffff      # 12 mask off type field from length
          bge    rindex, rlength, out-of-bounds    # 13 perform bounds check
uncomp3:  sll    rindex2, rindex, 1                # 14 scale index
          add    rtmp, rbase, rindex2              # 15 add base and index
          sh     rchar, 12(rtmp)                   # 16 perform 16b store
done:
```

Figure 9. Complete `castore` code for supporting accordion arrays.

ray pointer, but there is no need to re-load the array length field since we know the type and length of the array. After reconverging, the `uncompressed` and `inflated` paths strip off the type field from the array length, perform the bounds check, and complete the store.

In addition, for the `arraylength` bytecode, we add an additional instruction that masks out the type specifier.

While we have shown this code as assembly for ease of exposition, `caload` and `castore` operations get expanded in the high-level intermediate representation (HIR) by the code lowering pass. This exposes these operations for optimization and redundancy elimination by the compiler.

3.5 Handling Inflations

When a incompressible character needs to be written to a compressed array, we need to first inflate it. In the worst case, we need to allocate a new uncompressed array, copy over the old data (converting the 8-bit characters to 16-bit characters), update the type of the old array to `inflated`, and install the forwarding pointer. In addition, the inflation potentially presents a race condition, if another thread could be reading or writing the contents of the array during the inflation. For this worst case, we guarantee that this race condition is not observed, by pausing all other threads using the "stop the world" mechanism used by the garbage collector.

Luckily, the worst case scenario is not frequently necessary. First of all, inflations are quite rare in general. In all benchmarks, less than 1% of the character arrays need to be inflated. Second, in most cases, the inflation occurs very early in the array's lifetime. Figure 10 shows code for three inflation sites representative of the ones we observe to be most frequent dynamically. In each case, after standard inlining expands calls to *getChars* and `System.arraycopy`, the code responsible for the inflation is in the same scope as the allocation site. This permits a very simplistic (intraprocedural) form of escape analysis [8] to be performed to identify cases where references to the allocated array could not yet have escaped the thread at the point of a character store. When such analysis succeeds, we statically replace the call to the `inflate` function with a call to a version that does not perform a "stop the world." In the runs of our benchmarks, all but 23 of the over 16,000 dynamic calls invoke the cheaper version of the call.

In addition, the array is often the most recently allocated object by this thread. This means that frequently we can inflate an object **in place**, by extending the object's heap allocation, which avoids adding a level of indirection. The

```
char[] characters = new char[wordLength];
word.getChars(0, wordLength, characters, 0);
```

```
output = new char[count];
i = o - offset;
System.arraycopy(value, offset, output, 0, i);
```

```
char[] dst = new char[length];
if (ASSERT) Assert.assrt("...");
for(int i=0; i<length; i++){
    dst[i]=getChar(rawData, stringOffset+getCharOffset(i));
}
```

Figure 10. Three examples demonstrating the scope between the allocation of an array and the operations that potentially inflate it. These examples all cause inflations in the DaCapo benchmarks.

inflation code—both versions with and without the "stop the world"—checks dynamically whether this was the last object allocated (by comparing the current end of the array to be inflated to the next free address) and whether additional memory can be allocated to extend this allocation. Even when it can be statically proven that this object must have been the most recently allocated, a run-time check must be perform that the end of a page or a pinned object will not prevent an object from being expanded in place. We find that the majority of objects ($> 95\%$) can be expanded in place.

The overhead of the "stop the world" pauses that remain are not completely devastating in our experiments, but we believe this to be in part because our workloads are not heavily multithreaded. Techniques that mitigate this overhead for very parallel Java executions remains an open area of research, as we discuss in the Section 6.

3.6 Modifications to the Garbage Collector

As noted in Section 3.1, Harmony (by default) uses a compacting garbage collector, which presents two requirements and one opportunity for optimization. The first requirement is that to benefit from the array's compression after garbage collection, the GC needs to know the size of accordion arrays. This is accomplished by testing the array length (which needs to be loaded anyway) for negative values (indicating that type bits are set), and using an alternate path to compute the array size.

The second requirement is that the forwarding pointer held by inflated arrays needs to be exposed to the garbage collector. There are two reasons for this: i) the compressed version of the inflated array may hold the only pointer to the uncompressed version and we need to make sure that the uncompressed version is identified as live during the mark phase of the garbage collection, and ii) if the uncompressed version of the array is moved, the forwarding pointer in the compressed version needs to be updated. Both of these are relatively straight-forward to accomplish by exposing the forwarding pointer offset of an inflated array as a `slot` like any other object variable that holds a pointer.

The opportunity introduced by garbage collection is to remove the level of indirection introduced by inflation. During the compaction process, it is necessary to identify all of the pointers to a given object so they can be updated to point to the object's new location. The key idea of our optimization is that, rather than update references to point to an inflated array's new location after compaction, we update them to point to the new location of the target of its forwarding pointer (*i.e.*, the uncompressed version of the array). In this way, we can eliminate all references to the original compressed version of an inflated array and, therefore, consider the object dead. Figure 11 describes the steps of this optimization as implemented in the `gc_cc` in Harmony. All additional steps require work that scales with the number of inflated arrays that are compacted. Because very few inflations require forwarding pointers in practice, this optimizations provides negligible benefit.

Most inflated arrays, like most character arrays in general, do not live beyond a single garbage collection. Only in `eclipse` do a significant number live past a garbage collection. Only for long lived arrays would it make sense to consider *deflating* (*i.e.*, reallocating an uncompressed array as a compressed array if it no longer contained any incompressible characters), and we see no potential benefit for such an optimization.

3.7 Other Modifications to the Virtual Machine

While the above outlines the main changes required to implement accordion arrays, there are a few other parts of the JVM that directly access character arrays and these need to be modified to dynamically dispatch based on whether the array is compressed and handle inflated arrays. A few of these changes are small and isolated: we mask the type field for the standard `vector_get_length()` accessor and provide our own `vector_get_raw_length()` accessor for when we want the type field intact, and the JVM tool interface (JVMTI) needs to know the real size of objects. Two other changes are more substantial.

The Java Native Interface API provides functions that permit reading and writing the contents of Java character arrays, reading the contents of Strings as either Unicode UCS-2 or UTF-8 arrays, and querying the lengths of both character arrays and strings. These functions need to be modified to correctly handle all types of accordion arrays. These changes are straightforward using the same principles used to modify the code generator (Section 3.4).

The other major change is for the `System.arraycopy` method. While a correct version of this method could be generated from Java bytecode, Harmony includes a higher performance C implementation of this function. This C implementation must be extended to handle the cross-product of copying between compressed and uncompressed arrays, and

1. Marking of live objects is performed; **inflation objects that will be compacted are recorded on an inflation stack.**
2. **Objects on the inflation stack can be unmarked.**
3. New memory is allocated for objects to be compacted; relocation pointers stored in each object's `obj_info` field.
4. **For all objects, `obj`, on the inflation stack:** *# copy the uncompressed version's new location to* `obj_info`

```
    if obj.forward.will_be_relocated(): # will the uncompressed version be compacted?
        obj.obj_info = obj.forward.obj_info
    else:
        obj.obj_info = obj.forward
```

5. **The inflation stack can be discarded.**
6. Live objects are traversed, updating their references to any moved objects (as indicated by `obj_info`).
7. Compacted objects are copied to their new locations.

Figure 11. Optimization to eliminate inflation objects during garbage collection. The steps in bold are the ones added by the optimization. Alternately inflation objects can be inserted on the inflation stack during inflation.

potentially de-referencing forwarding pointers of inflated arrays. Again, these changes are rather straightforward.

4. Results

In this section, we explore the performance impact of accordion arrays as implemented in Harmony. We use the SPECjbb2005 benchmarks and 9 of the 11 DaCapo benchmarks from the 2006-10 version of the suite. Timing experiments are performed on idle dual-core machines running Red Hat Enterprise Linux (2.6.9-42.08.ELsmp) on a 2.66 GHz Core 2 Duo processor with 2GB of memory.

In this work, we are concerned with asymptotic performance, so we use Harmony's `server` execution manager configuration, which is a profile-directed adaptive optimizer that tries to maximize asymptotic performance at the expense of compilation time. To avoid including compilation time in our measurements, we use the DaCapo benchmarks' `-converge` repeatedly run the benchmarks until the execution time stabilizes. For SPECjbb2005, we use a configuration file that performs two runs back-to-back and record the results from the second run. Because we observe some variation in run times, we repeat each configuration three times and select the runs with the shortest execution time.

We run the default size versions of the DaCapo benchmarks. We only present results for 9 of the DaCapo benchmarks, because `antlr` and `jython` did not run to completion with Harmony's optimizing compiler. With the exception of `hsqldb` whose live heap memory grows as large as 72MB, all of the DaCapo benchmarks have maximum live memory sizes of 0.1MB - 30MB [5]. We used a 64MB heap size—`-Xms64m -Xmx64m`—for all of the benchmarks except `hsqldb` where a 256MB heap was used. For SPECjbb2005, we used 8 warehouses and a 512MB heap. Results are presented as relative execution times for the DaCapo benchmarks and relative throughput for SPECjbb, which runs a predefined amount of time.

Figure 12 shows the performance improvements resulting from our implementation of accordion arrays. With individual benchmarks ranging from an almost 8% reduction to a 1% increase in execution time, the average speedup

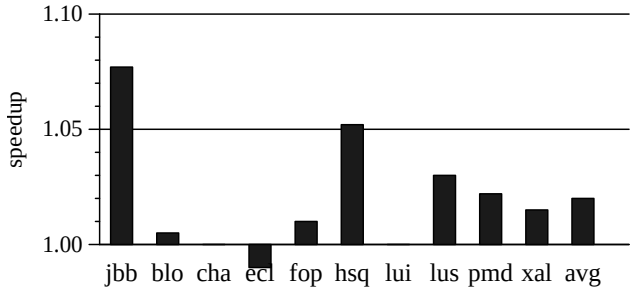

Figure 12. Accordion arrays achieve speedups of 1.08 to .99, averaging 1.02 across a benchmark suite containing SPECjbb2005 and 9 DaCapo benchmarks.

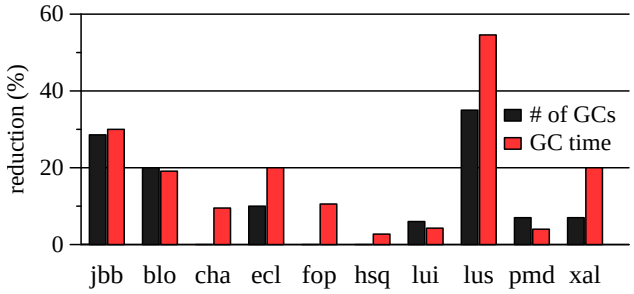

Figure 13. Accordion arrays reduce a Java program's heap allocation rate, which, in turn, reduces the rate at which garbage must be collected. The figure shows the reduction in the number of garbage collections performed during the run (in black) and in the total time spent on garbage collection (in grey).

across the benchmark suite is 2%. Due the significant non-uniformity of the speedups, we used instrumentation within the JVM to help identify the principal causes of the speedup.

From our analysis, we identified the benefit of the techniques to be correlated most strongly to two factors: 1) the working set of the application, and 2) the fraction of that working set that was made up of char arrays (see Figure 2). This reduction in working set translates into improved mutator (*i.e.*, non-garbage collector) execution, because this smaller working set of live objects results in better cache and TLB locality and a reduction of memory system bandwidth.

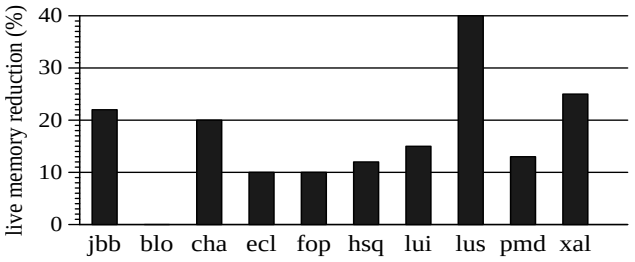

Figure 14. Accordion arrays reduce the size of live memory. Compressed arrays require less memory, resulting in a reduction in the size of memory live after a collection. This smaller live set reduces the work to perform a copying or compacting collection and also improves data locality.

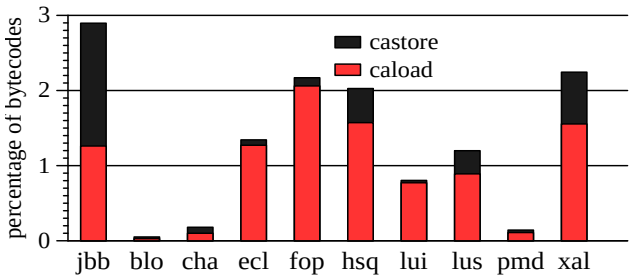

Figure 15. Reading and writing character arrays is relatively uncommon. `caload` and `castore` bytecodes account for less than 3% of the bytecodes executed across all of the benchmarks.

Given that garbage collection (GC) time is quite modest for most of these benchmarks, it is this improvement in mutator performance that is responsible for the bulk of the performance improvement for the benchmarks that see significant speedups.

Nevertheless, we found correlations between overall performance and the two aspects of the impact on the GC. The first is a reduction in allocation rate and the corresponding reduction in garbage collection (GC). Figure 13, shows the reduction in both GC frequency (*i.e.*, the number of GC's performed) and time spent performing garbage collection. Garbage collection frequency is reduced because the heap is not used up as quickly when compressed character arrays are allocated. In a number of cases, the reduction of time spent is greater than the reduction of number of collections, suggesting that each collection is getting faster. This occurs for copying and compacting collectors when there is a net reduction in the amount of live heap memory that needs to be copied/compacted.

Furthermore, while accordion arrays do not affect the number of live objects at a garbage collection, they do result in a smaller amount of memory being live if some of the live objects are compressed character arrays. The reduction of live heap memory is significant in a number of benchmarks, as shown in Figure 14.

One benchmark, `eclipse`, observes a small slowdown; this is the result of it having the highest number of inflations and the largest number of inflations requiring "stop the world" pauses. Three benchmarks achieve little benefit from accordion arrays: `bloat`, `chart`, and `luindex`. These benchmarks have rather low memory allocation rates, spending 1% or less of their execution time performing garbage collection in our experiments. With little upside potential, the fact that these programs do not see significant slowdowns attests to the low overheads that accordion arrays permit. In particular, we have drawn three conclusions to explain the absence of significant overhead.

First, we find that the overhead is low because character arrays are rather infrequently accessed. Specifically, we find that `caloads` and `castores` together typically only make up 1-3% of the total bytecodes executed, as shown in Figure 15. This result is corroborated by previous work characterizing the most frequently executed bytecodes in the SpecJVM benchmark suite [6, 17].

Second, the dynamic dispatch branches added to the implementations of `caload` and `castore` are extremely biased since more than 99% of the character arrays are of a single type. Furthermore, even when the minority type arrays are being processed, there is generally a locality in the branch outcomes that modern history-based dynamic branch predictors can exploit, because typically multiple characters are processed when an array is accessed. Because the majority of non-compressible arrays can be known as such at allocation time, inflations are diminishingly rare events; as a result their branches are highly biased and, hence, predictable, as well.

Third, because the dispatch and inflation branches are predictable, much of the work added by the accordion array implementation is "off the critical path." In a modern dynamically-scheduled—sometimes referred to as out-of-order—processor, instructions are only prevented from executing by true data dependences. As shown in Figure 16, the code that is added to implement accesses to accordion arrays is in the form of compare and branch sequences that do not produce values for other instructions. Instead these instructions only serve to validate the predictions made by the branch predictor and, as we previously noted, these branches are predictable. They do not delay the scheduling of the actual character load or store instruction, the only instruction in the `caload` and `castore` sequence that could be on the critical path.

5. Related Work

There is a significant body of work relating to improving the memory efficiency of Java, but, to our knowledge, there is no published work on compression of character arrays. In addition, much of the work has been done in the context of embedded Java implementations where it is allowable to sacrifice execution speed to reduce peak memory requirements.

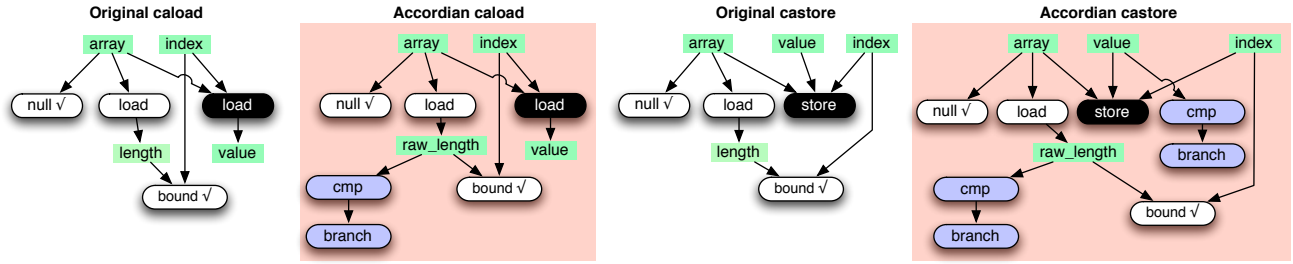

Figure 16. Accordion operation dataflow (as executed on an out-of-order processor) only adds predictable branches and ALU operations in the common case. The added operations (lightly shaded) for the accordion array operations do not delay the execution of the critical path (black).

Given that many Java objects are small, an important focus of research has been to minimize the size of object headers [3]. A key technique to simplifying object headers is to use a "thin lock" that only provides the simplest subset of Java object locking support; less common cases are handled by dynamically allocating memory to hold the full lock state, a process called inflation [4]. Ananian and Rinard reduce the size of the `vtable` pointer, by instead storing in the object an index to a table of `vtable` pointers, a space-time trade-off [2]. Dybvig et. al. propose taking this process one step further by entirely removing the `vtable` pointer and allocating objects in memory based on type such that the upper bits of the objects virtual address can be used to locate the object's `vtable` [16].

Chen et. al. observe that there is significant value locality in the contents of certain object fields across objects of the same type [7]. To exploit this locality, they propose an elaborate scheme that partitions objects into portions with low and high information content and compresses the low information content fraction. This approach reduces heap usage, but comes at a performance penalty. Ananian and Rinard attack the same locality but from a program analysis standpoint with a number of optimizations that reduce the size or eliminate object fields [2]. Their techniques include field size reduction (guided by bit-width analysis), elimination of unread or constant fields, and field externalization (storing in a hash table those objects that contain values that differ from the dominant value). Again, while these techniques can significantly reduce minimum heap size, they can impose execution overhead. Neither technique targets arrays of primitives.

With the advent of 64-bit architectures, there has been a significant amount of work in pointer compression. Adl-Tabatabai et. al. describe a straight-forward technique for compressing heap pointers on a 64-bit architecture, by storing a 32-bit offset from the beginning of the heap [1]. When pointers are loaded the heap base is added to the pointer before it is dereferenced. This approach provides substantial speedup relative to 64-bit pointers but is limited to a 4-GB heap. Lattner et. al. describe program analysis-based optimization that automatically compresses 64-bit pointers to 32-bit offsets in linked data structures [12]. Their approach differs from the one previously mentioned in that it is applied to a single data structure at a time, and because the offsets are relative to a per-data structure base pointer, each data structure can use up to 4 GB of memory.

In addition, there is extensive work in compressing Java code. Researchers have explored how to compress Java class files for storage and efficient network transmission (*e.g.*, [15]). In addition, Clausen et. al. proposed a macro instruction approach—using unused bytecodes to encode common application-specific sequences of bytecodes—to reduce the amount of bytecode storage necessary in an embedded Java interpreter [9].

6. Conclusion

In this paper, we have demonstrated the idea of accordion arrays, a straight-forward approach to reducing the heap memory consumed by Unicode character arrays in locales where the 8-bit ISO-8859-1 subset of UCS-2 is sufficient for representing the bulk of the text. We have presented a characterization of character array usage in non-numeric Java applications, described an abstract design of the accordion arrays, and demonstrated, in the context of the Harmony DRLVM, that the technique permits efficient implementations that can improve memory efficiency, which in turn can improve overall program execution time.

A number of avenues of research relating to Java character compression remain. First, and foremost, is identifying a race-free approach to inflation that completely avoids "stop the world" pauses. One approach that merits further research is the use of hardware support, either in the form of a hardware transactional memory [11, 13] or a fine-grain memory protection technique [14, 18]. Second, applications from non-western locales need to be characterized and techniques need to be developed for gracefully discontinuing use accordion arrays when they negatively impact performance. Finally, our performance results are collected with relatively loose constraints on the amount of heap memory available. As accordion arrays could prove to provide a substantial performance benefit in memory constrained environments,

studying their behavior over a broad range of heap sizes is warranted.

Acknowledgments

The author would like to thank Cliff Click of Azul Systems for suggesting Java character compression as a research topic and for discussion relating to the idea. In addition, the anonymous reviewers are thanked for their useful and insightful suggestions. This research was supported in part by NSF CAREER award CCR-03047260 and a gift from the Intel corporation.

References

[1] A.-R. Adl-Tabatabai, J. Bharadwaj, M. Cierniak, M. Eng, J. Fang, B. T. Lewis, B. R. Murphy, and J. M. Stichnoth. Improving 64-Bit Java IPF Performance by Compressing Heap References. In *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*, pages 100–110, 2004.

[2] C. S. Ananian and M. Rinard. Data Size Optimizations for Java Programs. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool Support for Embedded Systems*, pages 59–68, June 2003.

[3] D. Bacon, S. Fink, and D. Grove. Space- and Time-efficient Implementation of the Java Object Model. In *Proceedings of the Sixteenth European Conference on Object-Oriented Programming*, June 2002.

[4] D. F. Bacon, R. B. Konuru, C. Murthy, and M. J. Serrano. Thin Locks: Featherweight Synchronization for Java. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 258–268, 1998.

[5] 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 *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, New York, NY, USA, Oct. 2006. ACM Press.

[6] K. Bowers and D. Kaeli. Characterizing the SPEC JVM98 benchmarks on the Java virtual machine. Technical report, Northeastern University, Dept. of ECE, 1998.

[7] G. Chen, M. Kandemir, and M. J. Irwin. Exploiting Frequent Field Values in Java Objects for Reducing Heap Memory Requirements. In *VEE'05*, pages 68–78, June 2005.

[8] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for java. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 1–19, New York, NY, USA, 1999. ACM Press.

[9] L. Clausen, U. Schultz, C. Consel, and G. Muller. Java Bytecode Compression for Low-end Embedded Systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(3):471–489, May 2000.

[10] Harmony Dynamic Runtime Layer Virtual Machine (DRLVM). http://incubator.apache.org/harmony/subcomponents/drlvm/index.html.

[11] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.

[12] C. Lattner and V. Adve. Transparent Pointer Compression for Linked Data Structures. In *Proceedings of the ACM Workshop on Memory System Performance (MSP'05)*, Chigago, Illinois, June 2005.

[13] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles. Hardware atomicity for reliable software speculation. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 174–185, June 2007.

[14] N. Neelakantam and C. Zilles. UFO: A general-purpose user-mode memory protection technique for application use. Technical Report UIUCDCS-R-2007-2808, University of Illinois at Urbana-Champaign, 2007.

[15] W. Pugh. Compressing Java Class Files. In *Proceedings of the SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 247–258, May 1999.

[16] D. E. R. Kent Dybvig and C. Bruggeman. Don't Stop the BiBOP: Flexible and Efficient Storage Management for Dynamically-Typed Languages. Technical Report Technical Report number 400, Indiana University, March 1994.

[17] R. Radhakrishnan, J. Rubio, N. Vijaykrishnan, and L. K. John. Execution Characteristics of JIT Compilers. Technical report, University of Texas, Dept. of ECE, 1999.

[18] E. Witchel et al. Mondrian memory protection. In *Proceedings of ASPLOS-X*, Oct 2002.