Object-Relative Addressing: Compressed Pointers in 64-Bit Java Virtual Machines

Kris Venstermans, Lieven Eeckhout, and Koen De Bosschere

ELIS Department, Ghent University, Belgium
{kvenster,leeckhou,kdb}@elis.UGent.be

Abstract. 64-bit address spaces come at the price of pointers requiring twice as much memory as 32-bit address spaces, resulting in increased memory usage.

This paper reduces the memory usage of 64-bit pointers in the context of Java virtual machines through pointer compression, called Object-Relative Addressing (ORA). The idea is to compress 64-bit raw pointers into 32-bit offsets relative to the referencing object's virtual address. Unlike previous work on the subject using a constant base address for compressed pointers, ORA allows for applying pointer compression to Java programs that allocate more than 4GB of memory.

Our experimental results using Jikes RVM and the SPECjbb and DaCapo benchmarks on an IBM POWER4 machine show that the overhead introduced by ORA is statistically insignificant on average compared to raw 64-bit pointer representation, while reducing the total memory usage by 10% on average and up to 14.5% for some applications.

1 Introduction

In our recent work [1], we reported that Java objects increase by 40% in size when comparing 64-bit against 32-bit Java virtual machines. About half of this increase comes from the increased header which doubles in size. The other half comes from increased object fields containing pointers or references.

Running 64-bit Java virtual machines can thus be costly in terms of memory usage. This is a serious concern on heavy-loaded systems with many simultaneously running programs that are memory-intensive. In fact, overall system performance can quickly deteriorate because of memory page swapping once physical memory gets exhausted. One way of dealing with the excessive memory usage on 64-bit systems is to have more physical memory in the machine as one would provide on a 32-bit system. However, this is costly as physical memory is a significant cost in today's computer systems.

This paper proposes to address the increased memory usage in 64-bit Java virtual machines through *Object-Relative Addressing (ORA)*. Object-relative addressing is a pointer compression technique that compresses pointers in object fields as 32-bit offsets relative to the current object's address. The 64-bit virtual address of the referenced object is then obtained by adding the 32-bit offset to the 64-bit virtual address of the referencing object. In case the referenced object is further away than what can be represented by a 32-bit offset, object relative addressing interprets the 32-bit offset as an index in the *Long Address Table (LAT)* that translates the 32-bit offset into a 64-bit virtual address.

E. Ernst (Ed.): ECOOP 2007, LNCS 4609, pp. 79–100, 2007.
© Springer-Verlag Berlin Heidelberg 2007

The advance of object-relative addressing over prior work on the subject by Adl-Tabatabai *et al.* [2], is that object-relative addressing is not limited to Java programs that consume less than 4GB of heap, or the 32-bit virtual address space. Object-relative addressing enables pointer compression to be applied to all Java programs, including Java programs that allocate more than 4GB of memory.

We envision that object-relative addressing is to be used in conjunction with a memory management strategy that strives at limiting the number of inter-object references that cross the 32-bit address range. Crossing the 32-bit address range incurs overhead because the LAT needs to be accessed for retrieving the 64-bit address corresponding to the 32-bit offset. Limiting the number of LAT accesses thus calls for a memory allocator and garbage collector that strives at allocating objects within a virtual memory region that is reachable through the (signed) 32-bit offset. Such memory allocators and garbage collectors can be built using techniques similar to object colocation [3], connectivity-based memory allocation and collection [4,5], region-based systems [6], *etc*.

The experimental results using the SPECjbb2000 and the DaCapo benchmarks and the Jikes RVM on an IBM POWER4 machine show that object-relative addressing does not incur a run time overhead. Some applications experience a performance improvement up to 4.0% while other applications experience a slowdown of at most 3.5%; on average though, no statistically significant performance impact is observed. The benefit of ORA comes in terms of memory usage: the amount of allocated memory reduces by 10% on average and for some applications up to 14.5%.

This paper is organized as follows. After having discussed prior work in object pointer compression in section 2, we will present object-relative addressing in section 3. Section 4 will then detail our experimental setup. The evaluation of ORA in terms of overall performance, memory hierarchy performance and memory usage will be presented in section 5. Finally, we will discuss related work in section 6 before concluding in section 7.

2 Object Pointer Compression: Prior Work

The prior work on the subject by Adl-Tabatabai *et al.* [2] propose a straightforward compression scheme for addressing the memory usage in 64-bit Java virtual machines. They represent 64-bit pointers as 32-bit offsets from a base address of a contiguous memory region. Dereferencing or decompressing a pointer then involves adding the 32-bit offset to a base address yielding a 64-bit virtual addess. Reverse, compressing a 64-bit virtual address into a 32-bit offset requires substracting the 64-bit address from the base address; the lower 32 bits are then stored. A similar approach was proposed by Lattner and Adve [7] for compressing pointers in linked data structures.

The fact that 64-bit virtual addresses are represented as 32-bit offsets from a base address implies that this pointer compression technique is limited to Java programs that consume less than 4GB of storage. If a Java program allocates more than 4GB of memory, the virtual machine has to revert to the 64-bit pointer representation. This could for example be done by setting the maximum heap size through a command line option: if the maximum heap size is larger than 4GB, uncompressed pointers are used; if smaller than 4GB, compressed pointers are used.

Adl-Tabatabai *et al.* apply their pointer compression method to both vtable pointers and pointers to other Java objects, so called object references. The 32-bit object references are then relative offsets to the heap's base address; the 32-bit vtable pointers are relative offsets to the vtable space's base address.

In this paper, we focus on compressing object references and do not address vtable pointer compression. The reason is that vtable pointers are not that big of an issue when it comes to pointer compression. The 32-bit vtable pointer offsets are highly likely to be sufficient even for programs that allocate very large amounts of memory; it is highly unlikely to require more than 4GB of memory for allocating vtables. In other words, the pointer compression method by Adl-Tabatabai *et al.* is likely to work properly when applied to vtable pointers. Moreover, recent work by Venstermans *et al.* [8] has proposed a technique that completely eliminates the vtable pointer from the object header through typed virtual addressing. We also refer to the related work section of this paper for a discussion on object header reduction techniques.

3 Object-Relative Addressing

Object-Relative Addressing (ORA) is a pointer compression technique for 64-bit Java virtual machines that does not suffer from the 4GB heap limitation in Adl-Tabatabai *et al.*'s method. The goal of ORA is to enable heap pointer compression for all Java programs, even for programs that allocate more than 4GB of memory.

3.1 Basic Idea

Figure 1 illustrates the basic idea of object-relative addressing (ORA) and compares ORA against the traditional way of referencing objects in 64-bit Java virtual machines. We call the referencing object the object that contains a pointer in its data fields. The object being referenced is called the referenced object. ORA references objects through 32-bit offsets. The 'fast' decompression path then adds this 32-bit offset to the referencing object's virtual address for obtaining the virtual address of the referenced object.

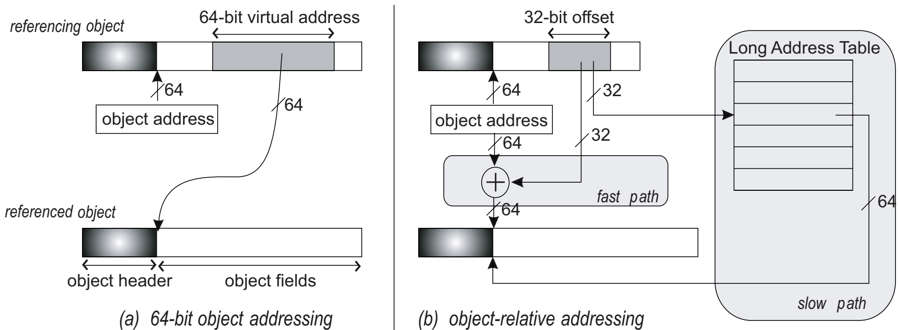

Fig. 1. Illustrating the basic idea of object-relative addressing (on the right) compared against the traditional 64-bit addressing (on the left)

```
read 32-bit object reference;
if (least significant bit of 32-bit reference is NOT set) {
    /* fast decompression path */
    add 32-bit object reference to 64-bit object
      virtual address to form 64-bit object address;
}
else {
    /* slow decompression path */
    index LAT for reading 64-bit object address;
}
```

Fig. 2. High-level pseudocode for decompressing 32-bit object references

This is the case when both the referencing object and the referenced object are close enough to each other so that a 32-bit offset is sufficiently large. In case both objects are further away from each other in memory than what can be addressed through a 32-bit offset, ORA follows the ‘slow’ decompression path. The 32-bit offset is then considered as an index into the *Long Address Table (LAT)* which holds 64-bit virtual addresses corresponding to 32-bit indexes.

The end result of object-relative addressing is that only 32 bits of storage are required for storing object references. This reduces the amount of memory consumed compared to the traditional way of storing object references which requires 64 bits of storage. We now go through the details of how ORA can be implemented. We discuss (i) how pointers are decompressed, (ii) how to compress pointers, (iii) how to deal with null pointer representation, (iv) how to manage the LAT, (v) what the implications are for garbage collection, (vi) how ORA compares to Adl-Tabatabai *et al.*’s method in terms of anticipated runtime overhead, and finally (vii) what the implications are for memory management.

3.2 Decompressing Pointers

Decompressing 32-bit object references requires determining whether the fast or slow path is to be taken. This is done at runtime by inspecting the least significant bit of the 32-bit offset; in case the least significant bit is zero, the fast path is taken; otherwise, the slow path is taken. This is illustrated in Figure 2 showing the high-level pseudocode for decompressing 32-bit object references into 64-bit virtual addresses.

The way how the high-level pseudocode is translated into native machine instructions has a significant impact on overall performance. And in addition, efficient pointer decompression is likely to result in different implementations on platforms with different instruction-set architectures (ISAs). For example, in case predicated execution is available in the ISA [9], a potential implementation could predicate the fast and slow paths. Or, in case a ‘base plus index plus offset’ addressing mode is available in the ISA, computing the address of an object field being accessed in the referenced object could be integrated into a single memory operation, *i.e.*, the decompression arithmetic could be combined with the field access. The referencing object’s virtual address plus

```
                            ;; R4 contains the referencing
                            ;;   object's virtual address
      ld4  R1, [R4 + offset] ;; load 32-bit object offset and
                            ;;   sign-extend it into R1
                            ;; fast decompression path
      add  R2, R4, R1       ;; compute 64-bit address
      tst  R1, 1            ;; test least significant bit (LSB)
      bre  L2               ;; jump to L2 in case non-zero
L1:   ...                   ;; referenced object's virtual
                            ;;   address is in R2 here
      ...
L2:                         ;; slow decompression path
      mask R1               ;; compute LAT index by masking R1
      ld8  R2, [R5 + R1]    ;; load 64-bit address from LAT
                            ;; R5 contains LAT address and
                            ;;   R1 contains LAT index
      jmp  L1
```

Fig. 3. Low-level pseudocode for decompressing 32-bit object references: the if-then decompression approach

the 32-bit offset plus the offset of the object field in the referenced object could then be encoded in a single addressing mode.

In our experimental setup using a PowerPC setup, we were not able to implement these optimizations because the PowerPC ISA does not provide predication, nor does it support the 'base plus index plus offset' addressing mode. Instead, we consider two implementations to pointer decompression that are generally applicable across different ISAs. These two decompression implementations have different performance trade-offs which we discuss now and which we will experimentally evaluate in section 5.

If-then pointer decompression. The if-then implementation is shown in Figure 3. The assembler code generated for decompressing 32-bit object references optimizes the corresponding high-level pseudocode by optimizing for the most common case, namely the fast path. We (speculatively) compute the virtual address of the referenced object by adding the 32-bit offset with the referencing object's virtual address. In case the least significant bit of the 32-bit offset is zero, we then continue fetching and executing along the fall-through path. Only in case the least significant bit of the 32-bit offset is set, we jump to the slow path. The slow path selects a number of bits from the 32-bit offset that will serve as index into the LAT. The slow path then indexes the LAT which reads the 64-bit virtual address of the referenced object.

Patched pointer decompression. Patched pointer decompression optimizes the common case even further by assuming that the fast path is always taken. This results in the code shown in Figure 4. In other words, the 32-bit offset is added to the referencing object's virtual address to obtain the referenced object's virtual address. This avoids the conditional branch as needed in the if-then decompression implementation. In case the referenced object may not be reachable using a 32-bit offset, the decompression code

```
                                 ;; R4 contains the referencing
                                 ;;   object’s virtual address
     ld4  R1, [R4 + offset]      ;; load 32-bit object offset and
                                 ;;   sign-extend it into R1
                                 ;; fast decompression path
     add  R2, R4, R1             ;; compute 64-bit address
L1:  ...                         ;; referenced object’s virtual
                                 ;;   address is in R2 here
```

Fig. 4. Low-level pseudocode for decompressing 32-bit object references: the patched decompression approach before code patching is applied

```
                                 ;; R4 contains the referencing
                                 ;;   object’s virtual address
     ld4  R1, [R4 + offset]      ;; load 32-bit object offset and
                                 ;;   sign-extend it into R1
     jmp L2

L1:  ...                         ;; referenced object’s virtual
                                 ;;   address is in R2 here
     ...
L2:  add  R2, R4, R1             ;; compute 64-bit address
     tst  R1, 0                  ;; test least significant bit (LSB)
     bre L1                      ;; jump to L1 in case zero
                                 ;; slow decompression path
     mask R1                     ;; compute LAT index by masking R1
     ld8  R2, [R5 + R1]          ;; load 64-bit address from LAT
                                 ;; R5 contains LAT address and
                                 ;;   R1 contains LAT index
     jmp  L1
```

Fig. 5. Low-level pseudocode for decompressing 32-bit object references: the patched decompression approach after code patching is applied

needs to be patched. Code patching is done at run time whenever pointer compression reveals that objects may no longer be reachable using compressed pointers, as will be discussed in the next section. The decompression code after patching is shown in Figure 5. Code patching replaces the addition (of the 32-bit offset with the referencing object’s virtual address) with a jump to a piece of code that does the pointer decompression using the if-then approach. Since most object references will follow the fast path, the patched decompression approach (before patching is applied) will be substantially faster than the if-then decompression approach.

3.3 Compressing Pointers

Compressing 64-bit pointers to 32-bit offsets is done the other way around, see Figure 6. We first compute the difference between the 64-bit virtual addresses of the referenced

```
compute difference between 64-bit virtual addresses
  of the referenced object and the referencing object;
if (difference is smaller than 2GB) {
    /* fast compression path */
    store 32-bit offset;
}
else {
    /* slow compression path */
    allocate entry in LAT;
    store the referenced object's address in the
      LAT in allocated entry;
    store LAT index as a 32-bit value while setting
      the LSB of 32-bit value being stored;

    /* for the patched approach */
    patch pointer decompressions that need to;
}
```

Fig. 6. High-level pseudocode for compressing 64-bit object references

and referencing objects. If this difference is smaller than 2GB, *i.e.*, can be represented by a 32-bit offset, we then store the difference as a 32-bit offset in the referencing object's data fields. If on the other hand the difference is larger than 2GB, we allocate a LAT entry and store the referenced object's virtual address in the allocated LAT entry. The LAT entry's index is then stored in the referencing object's data fields while setting the LSB of the stored LAT index. In case of the patched decompression approach, all pointer decompressions that may read the 32-bit offset need to be patched. The patching itself is done as described in the previous section. This requires that the VM keeps track of the accesses to a given data field in an object of a given type.

3.4 Null Pointer Representation

An important issue when compressing references is how to deal with null pointers. The representation of a null value in native code is typically a 64-bit zero value. Compressing a 64-bit null value to a 32-bit representation under ORA is not trivial. A naive approach would represent the compressed null value as a 32-bit zero value. However, the 32-bit null value would then be decompressed to the `this` pointer, *i.e.*, the pointer to the object itself. This would make the null value indistinguishable from the `this` pointer.

For dealing with null pointer representation, we take the following approach. We first add the 32-bit compressed pointer to the referencing object's 64-bit virtual address. In case the least significant 32 bits of the resulting value are zero, we consider the 32-bit compressed pointer as the null value. This means we no longer have a single null value. As a result, a special treatment is required when comparing two pointers. In case both pointers represent the null value, a simple comparison may evaluate to not equal, for example, in case both compressed pointers come from different objects. As

such, we need to capture this special case in the virtual machine's code generator when generating code that compares pointers. In addition, given that all memory addresses with the 32 least significant bits set to zero represent null values, we cannot allocate objects at these 4GB memory boundaries.

3.5 Managing the LAT

Another important issue to deal with is how to manage the Long Address Table (LAT). Allocating LAT entries is very straightforward by advancing the LAT head pointer. Managing LAT entries is done during garbage collection (GC). Let us first consider non-generational garbage collection. A SemiSpace garbage collector for example, which copies reachable objects from one space to the other upon a GC, requires that the LAT be recomputed, *i.e.*, a new LAT is built up during GC and the old LAT is discarded. A Mark-Sweep garbage collector that does not need to copy reachable objects, in theory, does not require recomputing the LAT. However, in order not to let the LAT explode because of entries pointing to dead objects, a good design choice is to also recompute the LAT upon a mark-sweep collection.

For generational garbage collectors, we recommend using two LATs, one associated with the nursery space and another one associated with the mature space. The nursery LAT contains references in and out of the nursery space; the mature LAT contains all other references. Upon a nursery GC, all reachable nursery objects are copied to the mature space; as such, the nursery LAT can be discarded and the mature LAT possibly needs to be updated for the newly copied objects. Upon a full GC, the same strategy can be used as under a non-generational GC, *i.e.*, the mature LAT needs to be rebuilt and in addition, the nursery LAT is discarded.

In case of the unlikely event of the LAT running full—the LAT can be chosen to be sufficiently large, and, in addition, a good object allocation strategy would strive at reducing the number of LAT entries allocated—a garbage collection could be triggered to reclaim unreachable memory. GC will rebuild the LAT, and as a result the LAT will likely shrink (or if needed, the LAT size could be increased). A data structure linking memory pages makes increasing the LAT relatively easy, *i.e.*, the LAT does not need to be copied.

3.6 Implications to Copying Garbage Collectors

Object-relative addressing raises the following issue to copying garbage collectors. Consider the case where object A has a reference to object B in its data fields. Assume object A is reachable; by consequence, object B is also reachable. The garbage collector has to assume both objects are live and a copying collector will thus have to copy both objects. Assume the copying collector first copies object A. The compressed pointer in A referencing to B then needs to be updated because object A was copied which changes the compressed pointer's base address. Upon copying object B, the compressed pointer in A referencing to B needs to be computed again because now B is moved. In other words, the compressed pointer in A needs to be recomputed twice under a copying garbage collector.

In order not to recompute the compressed pointer twice, we do the following. During garbage collection, we maintain both the original object A and a copied version of object A in the scan list, and we use the original object A to retrieve the virtual address of the referenced object B. As such, we need to recompute the compressed pointer only once, namely upon scanning object B.

3.7 Discussion

Note that pointer compression and decompression in ORA cannot be optimized as in the simple pointer compression technique proposed by Adl-Tabatabai *et al.* [2]. Adl-Tabatabai *et al.* report that it is "crucial to optimize the unnecessary compression and decompression in order to get net performance gains". This can be done by considering the phase ordering between code optimization and compression/decompression arithmetics to make sure the additional compression/decompression arithmetics get optimized whenever possible. The optimizations by the Adl-Tabatabai *et al.* approach include for example:

– *load-store forwarding*: If a loaded 32-bit offset is subsequently stored, the 32-bit offset does not need to be decompressed and subsequently compressed again; the 32-bit offset can be stored right away. This is not the case for ORA because the base address to which the 32-bit offset relates is the virtual address of the referencing object. And since the objects from which the 32-bit offset is loaded is likely to be different from the object to which the 32-bit offset needs to be stored, the 32-bit offset to be stored needs to be recomputed.
– *reference comparison*: Comparing objects' virtual addresses can be done easily by comparing the 32-bit offsets in the Adl-Tabatabai *et al.* approach. This is not the case for ORA; the 64-bit virtual addresses need to be decompressed from the 32-bit offsets before allowing for a comparison, the reason being that the base addresses are likely to be different for both 32-bit compressed pointers.
– *reassociation of address expressions*: Computing the address of an object field or array element involves two additions in Adl-Tabatabai *et al.*'s approach: the heap base needs to be added to the 32-bit offset plus the object field's offset. Under many circumstances, one addition can be pre-computed at compile time. For example, in case of an object field access, the heap base address and the object field's offset are both constants and can be pre-computed. Again, this is an optimization that cannot be applied to ORA because the base address is not constant. A related optimization is to apply common subexpression elimination. For example, if multiple fields of the same object are accessed, then the heap base address plus the 32-bit offset is a common subexpression that can be eliminated, *i.e.*, does not need to be recomputed over and over again. The latter optimization can also be applied under ORA.

In summary, the pointer compression approach by Adl-Tabatabai *et al.* allows for a number of optimizations that cannot be applied to ORA. Hence, it is to be expected that ORA will perform poorer than the pointer compression technique proposed by Adl-Tabatabai *et al.* However, ORA can apply pointer compression to Java programs that allocate more than 4GB of heap memory, which cannot be done using Adl-Tabatabatai *et al.*'s method.

It is interesting to note that, in case the ‘base plus index plus offset’ memory addressing mode would be available in the host ISA—again, which is not the case in our PowerPC setup—ORA would be able to apply an important optimization that would likely close (part of) the gap between ORA and Adl-Tabatabai *et al.*’s technique. Pointer decompression can then be combined with field offset computation into a single address expression. In that case, the optimization done by Adl-Tabatabai *et al.* to pre-compute constants would be subsumed by combining the pointer decompression with field offset computation.

3.8 Implications for Memory Management

As mentioned in the introduction, object-relative addressing is envisioned to be used in conjunction with a dedicated memory management approach for allocating objects in memory regions such that all inter-object references within a memory region can be represented by a 32-bit offset. To this end, ORA can rely on previously proposed memory management approaches that allocate connected objects into memory regions while minimizing the number of references across memory regions. Example memory management approaches that serve this need are object colocation [3], connectivity-based garbage collection [4,5] and region-based systems [6]. The smarter the memory management strategy, the smaller the number of LAT accesses, the smaller the compression/decompression overhead, and thus the higher overall performance.

In this context, it is also important to note that ORA is flexible in the sense that ORA can be activated and deactivated for particular object types; or, if needed, ORA can even be activated/deactivated for particular references between pairs of object types. It was this insight on ORA’s flexibility that lead us to our compression/decompression scheme with patching. The slow decompression path is not called for at the beginning of the program execution as the heap is small enough—as such we always execute the fast path and thus eliminate executing the if-then statement. Once an inter-object reference is detected that cannot be represented by a 32-bit value, all the code that may possibly read the compressed pointer needs to be patched. ORA is flexible enough to handle such cases as a safety net in case the memory management strategy would fail to allocate objects so that all pointers can be represented as 32-bit offsets.

4 Experimental Setup

We now detail our experimental setup: the virtual machine, the benchmarks and the hardware platform on which we perform our measurements. We also detail how we performed our statistical analysis on the data we obtained.

4.1 Jikes RVM

The Jikes RVM is an open-source virtual machine developed by IBM Research [10]. We used the recent 64-bit AIX/PowerPC v2.3.5 port. We extended the 64-bit Jikes RVM in order to be able to support the full 64-bit virtual address range. In this paper, we use the GenMS garbage collector. GenMS is a generational collector that copies reachable

Table 1. The benchmarks used in this paper

suite	benchmark	description
SPECjbb2000	pseudojbb	models middle tier of a three-tier system
DaCapo	antlr	parses one or more grammar files and generates a parser and lexical analyzer for each
	bloat	performs a number of optimizations and analysis on Java bytecode files
	fop	takes an XSL-FO file, parses it and formats it, generating a PDF file
	hsqldb	executes a JDBCbench-like in-memory benchmark, executing a number of transactions against a model of a banking application
	jython	interprets the pybench Python benchmark
	pmd	analyzes a set of Java classes for a range of source code problems

objects from the nursery to the mature space upon a nursery space garbage collection. A full heap collection then collects the heaps using the mark-sweep strategy.

4.2 Benchmarks

The benchmarks that we use in this study come from the SPECjbb2000 and DaCapo benchmark suites, see Table 1. SPECjbb2000 is a server-side benchmark that models the middle tier (the business logic) of a three-tier system. Since SPECjbb2000 is a throughput benchmark that runs for a fixed amount of time, we use pseudojbb which runs for a fixed amount of work (35,000 transactions) and an increasing number of warehouses going from 1 up to 8 warehouses. The initial heap size is set to 256M and the maximum heap size is set to 512MB. The DaCapo benchmark suite [11] is a relatively new set of open-source, client-side Java benchmarks. The DaCapo benchmarks exhibits more complex code, richer object behaviors and more demanding memory system requirements than the SPECjvm98 client-side benchmarks. We set the maximum heap size to 512MB with a 100MB initial heap size in all of our experiments. We use the DaCapo benchmarks under version beta-2006-08. Unfortunately, we were unable to run all DaCapo benchmarks on Jikes RVM v2.3.5; we use the 6 DaCapo benchmarks mentioned in Table 1. For bloat and jython we use the small input—the large input failed to run. The other 4 DaCapo benchmarks are run with the large input.

4.3 Hardware Platform

The hardware platform on which we have done our measurements is the IBM POWER4 which is a 64-bit microprocessor that implements the PowerPC ISA. The POWER4 is an aggressive 8-wide issue superscalar out-of-order processor capable of processing over 200 in-flight instructions. The POWER4 is a dual-processor CMP with private L1 caches and a shared 1.4MB 8-way set-associative L2 cache. The L3 tags are stored on-chip; the L3 cache is a 32MB 8-way set-associative off-chip cache with 512 byte lines. The TLB in the POWER4 is a unified 4-way set-associative structure with 1K entries. The effective to real address translation tables (I-ERAT and D-ERAT) operate as caches

for the TLB and are 128-entry 2-way set-associative arrays. The standard memory page size on the POWER4 is 4KB. Our 615 pSeries machine has one single POWER4 chip. The amount of RAM-memory equals 1GB.

In the evaluation section we will measure execution times on the IBM POWER4 using hardware performance counters. The AIX 5.1 operating system provides a library (`pmapi`) to access these hardware performance counters. This library automatically handles counter overflows and kernel thread context switches. The hardware performance counters measure both user and kernel activity.

4.4 Statistical Analysis

In the evaluation section, we want to measure the impact on performance of ORA. Since we measure on real hardware, non-determinism in these runs results in slight fluctuations in the number of execution cycles. In order to be able to take statistically valid conclusions from these runs, we employ statistics to determine 95% confidence intervals from 8 measurement runs. We use the unpaired or noncorresponding setup for comparing means, see [12] (pages 64–69).

5 Evaluation

In the evaluation section of this paper, we first measure the performance impact of ORA and subsequently focus on the reduction in memory usage and its impact on the memory subsystem.

5.1 Performance

For quantifying the performance impact of ORA applied to Java application objects, we consider five scenarios that we compare against the base case. Our base case is a 64-bit version of Jikes RVM which assumes 64-bit pointer representations in object data fields. Figure 7 shows the performance for each of the following five scenarios relative to the base case. Initially, we assume that all pointer compressions and decompressions occur through the fast path, *i.e.*, all inter-object references can be represented as 32-bit offsets. We then subsequently quantify the overhead of pointer compression and decompression through the slow path accessing the LAT.

Compressed pointers with zero heap base. The ‘compressed pointer with zero heap base’ is the scenario where all 64-bit pointers in object data fields are compressed to 32-bit pointers with the heap base address being zero. This means that loading the 32-bit compressed pointers (with zero extension) yields the virtual address of the referenced object; storing a compressed pointer is done by storing the four least significant bytes of the virtual address to memory. This scenario shows the best possible performance that can be achieved through compressed pointer representation: pointers are compressed and there is no compression/decompression overhead. The average performance gain is 5.0%, and up to 14.2% for hsqldb. This performance gain is a direct consequence of the memory savings through a reduced number of data cache misses and D-TLB misses.

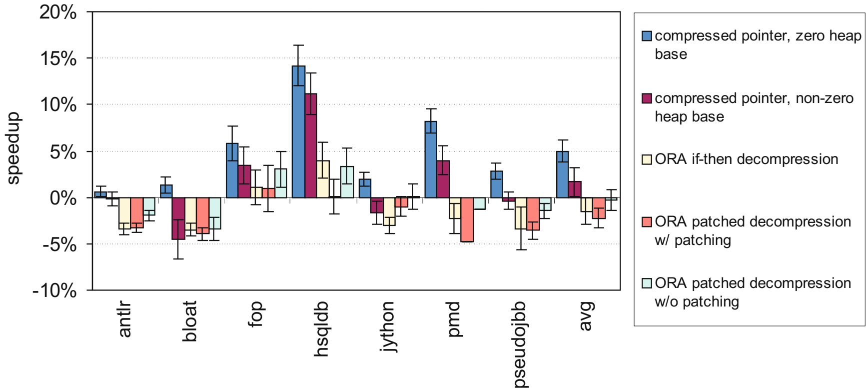

Fig. 7. Evaluating object-relative addressing in terms of performance

Compressed pointers with non-zero heap base. The 'compressed pointer with non-zero heap base' is similar to the previous scenario except that the heap base address is non-zero. In other words, decompressing a 32-bit pointer requires adding the 32-bit offset to the 64-bit heap base address. This scenario corresponds to Adl-Tabatabai *et al.*'s approach: it assumes that the heap space is no larger than 4GB, and assumes a fixed heap base address. The average performance gain for compressed pointers with a non-zero heap base drops to 1.7%; the maximum performance gain is observed for hsqldb (11.1%) and the largest slowdown is observed for bloat (-4.6%).

The 1.7% average performance gain over the base case is smaller than what is reported by Adl-Tabatabai *et al.* in [2]. The reason is that our results are for the PowerPC ISA using Jikes RVM whereas the results by Adl-Tabatabai *et al.* are for the Intel Itanium Processor Family (IPF) using ORP and StarJIT. As a result, not all optimizations implemented by Adl-Tabatabai *et al.* may be implemented in our system. Note however that the goal of this scenario is not to re-validate the approach proposed by Adl-Tabatabai *et al.*, but rather to quantify the overhead of pointer compression/decompression in our framework.

ORA with if-then decompression. The 'ORA if-then decompression' scenario implements object-relative addressing using the if-then decompression implementation. This scenario includes testing the LSB of the 32-bit compressed pointer for determining whether to take the fast or the slow path. This scenario incurs an average slowdown of 1.5%. The highest slowdown observed is 3.5% (bloat); the highest speedup observed is 4.0% (hsqldb).

ORA with patched decompression. There are two 'ORA patched decompression' scenarios. The first 'w/ patching' scenario assumes that all loads are patched, *i.e.*, all pointer decompressions are done by jumping to an if-then decompression scheme as shown in Figure 5. The second 'w/o patching' scenario assumes that none of the loads are patched, *i.e.*, all pointer decompressions are done by adding the 32-bit offset to the referencing object's virtual address as shown in Figure 4. As expected, the 'w/ patching'

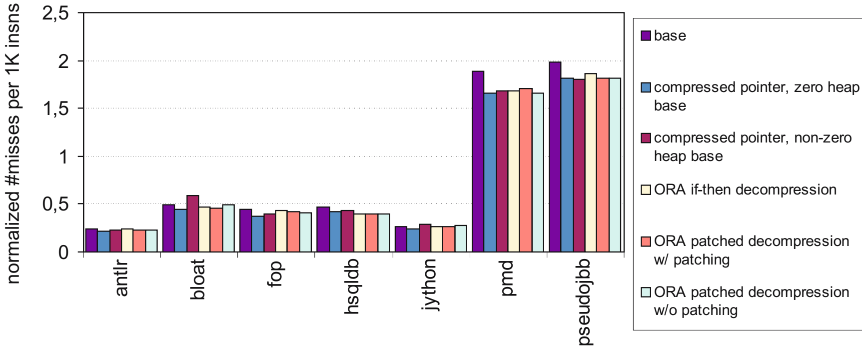

Fig. 8. The number of L2 misses per 1K instructions of the base run

scenario incurs a higher overhead than the 'if-then decompression' because of the jump instruction, however, this overhead is very small and not statistically significant. The 'w/o patching' scenario, which eliminates the jump instruction in the decompression scheme and which is the most realistic scenario in case an appropriate memory management strategy is available, results in a statistically insignificant average slowdown of 0.2%. The maximum slowdown observed is 3.4% (pmd) and the maximum speedup observed is 3.4% (hsqldb).

LAT access overhead. So far, we assumed that all decompressions occur along the fast path, *i.e.*, the slow decompression path is never taken. In order to quantify the overhead of going through the slow path we have set up a benchmarking experiment in which the nursery and mature space are located more than 4GB away from each other. This benchmarking experiment implies that all inter-generational pointers—from nursery objects to mature objects, and vice versa—have to pass through the LAT. In other words, a LAT entry is allocated for all inter-generational pointers, and the slow path is taken when compressing/decompressing inter-generational pointers. On average, 15.5% of all references go through the slow path, up to 23.6% (fop) and 36.6% (bloat). The average slowdown of this benchmarking experiment is 4.1% $\pm$ 1.3%. We want to emphasize that the sole purpose of this benchmarking experiment is to quantify the overhead due to taking the slow compression/decompression path; the goal of this experiment is not to present a use case scenario. In practice, when an appropriate memory management strategy is employed that limits the number of LAT accesses, even smaller slowdowns are to be expected.

5.2 Cache Hierarchy Performance

Figures 8 and 9 show the number of L2 and L3 misses per 1K instructions of the base run, respectively. In these graphs, we normalize the number of L2 and L3 misses for the various scenarios from above to the number of instructions in the base run. We clearly observe that the number of L2 misses and L3 misses (main memory accesses) reduces

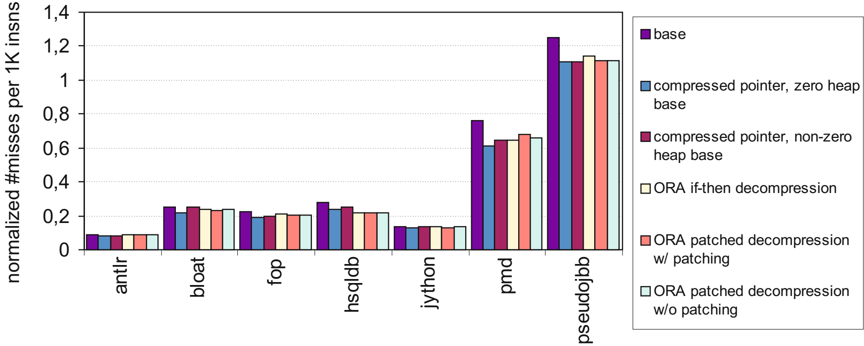

Fig. 9. The number of L3 misses per 1K instructions of the base run

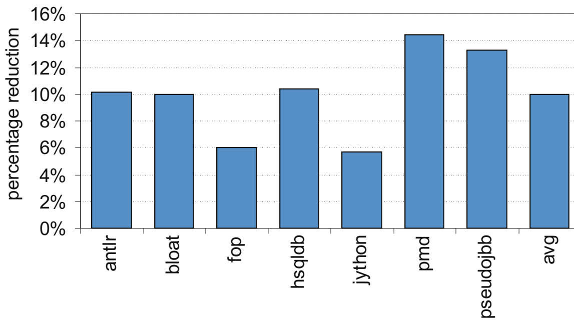

Fig. 10. Reduction in the number of allocated bytes through ORA

through ORA, up to 12.6% and 22.4% for pmd and hsqldb. In other words, ORA better utilizes the cache hierarchy reducing the pressure on main memory.

5.3 Memory Usage

We now analyze the impact of ORA on memory usage and quantify the impact of ORA on the number of bytes allocated and the number of memory pages touched.

Figure 10 shows the reduction in the number of allocated bytes through object-relative addressing. Compressing 64-bit object references reduce the number of allocated bytes by 10% on average and reductions up to 14.5% for pmd.

Figures 11 and 12 show the number of memory pages in use on the vertical axis as a function of time (measured in the number of allocations) on the horizontal axis for pseudojbb and hsqldb, respectively. Each figure shows two graphs, one for the base 64-bit pointer representation (top graph), and one for the compressed pointer representation through object-relative addressing (bottom graph). (We observed similar

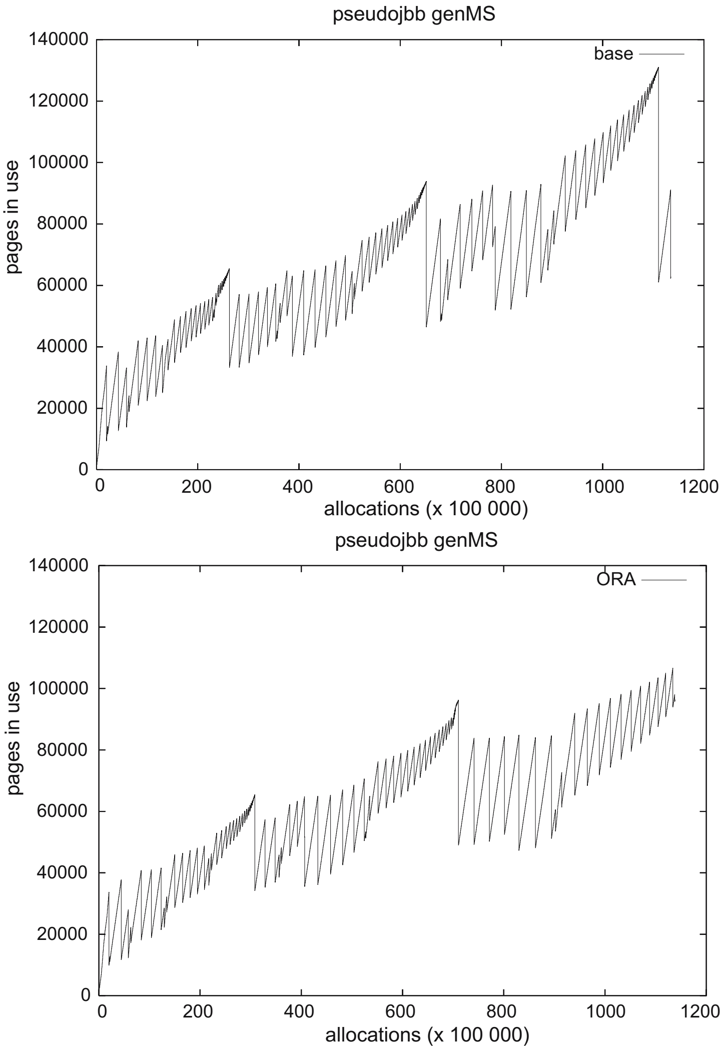

Fig. 11. Number of pages in use as a function of time for pseudojbb: the base 64-bit pointer representation (at the top) and the ORA compressed pointer representation (at the bottom)

curves for the other benchmarks.) The curves in these graphs increase as memory gets allocated until a garbage collection is triggered after which the number of used pages drops. The small drops correspond to nursery collections; the large drops correspond to mature collections collecting the full heap. The graph for hsqldb shows that the number of pages in use is substantially lower under ORA than under the base 64-bit pointer representation. The graph for pseudojbb shows that the reduced number of pages in use delays garbage collections, *i.e.*, it takes a longer time before a garbage collection is triggered.

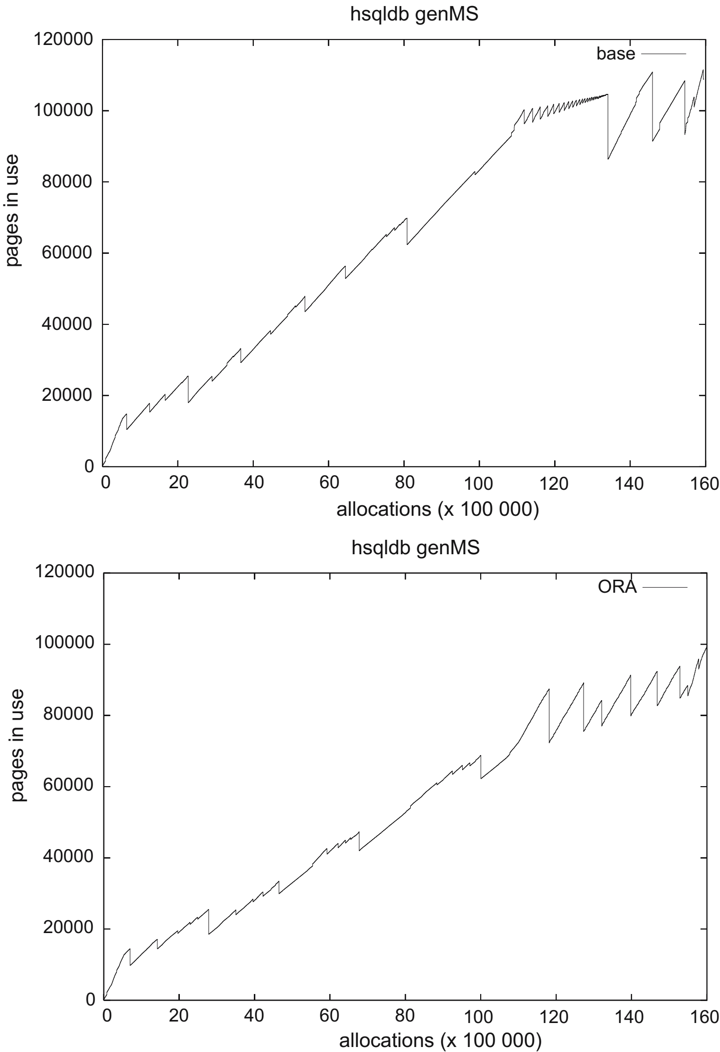

Fig. 12. Number of pages in use as a function of time for hsqldb: the base 64-bit pointer representation (at the top) and the ORA compressed pointer representation (at the bottom)

6 Related Work

6.1 Java Program Memory Characterization

Dieckmann and Hölzle [13] present a detailed characterization of the allocation behavior of SPECjvm98 benchmarks. Among the numerous aspects they evaluated, they also quantified object size and the impact of object alignment on the overall object size. This study was done on a 32-bit platform.

Venstermans *et al.* [1] compare the memory requirements for Java applications on a 64-bit virtual machine versus a 32-bit virtual machine. They concluded that objects are nearly 40% larger in a 64-bit VM compared to a 32-bit VM. There are two primary reasons for this. First, the header in 64-bit mode is twice as large as in 32-bit mode. This accounts for approximately half the object size increase. Second, a reference in 64-bit computing mode is twice the size as in 32-bit computing mode. This causes the data fields that contain references to increase; this accounts for roughly the other half of the total object size increase between 32-bit and 64-bit. Pointer compression as proposed in this paper addresses the size increase because of references in the object data fields.

A number of related research studies have been done on characterizing the memory behavior of Java applications, such as [14,15,16]. Other studies aimed at reducing the memory usage of Java applications, for example, using techniques such as heap compression [17], object compression [18], *etc.*

6.2 Pointer Compression

Mogul *et al.* [19] studied the impact of pointer size on overall performance on a Digital Alpha system using a collection of C programs. They compared the performance of the same application in both 64-bit and 32-bit mode. They concluded that while performance was often unaffected by larger pointers, some programs experienced significant performance degradations, primarily due to cache and memory page issues. The study done by Venstermans *et al.* [1] confirms these findings for Java programs.

Adl-Tabatabai [2] address the increased memory requirements of 64-bit Java implementations by compressing 64-bit pointers to 32-bit offsets. They apply their pointer compression technique to both the Type Information Block (TIB) pointer—or the vtable pointer—and the forwarding pointer in the object header and to pointers in the object itself. As mentioned before, the approach by Adl-Tabatabai *et al.* is limited to applications within a 32-bit address space. As such, applications that require more than 4GB of memory cannot benefit from pointer compression.

Lattner and Adve [7,20] apply a similar approach to compressing pointers in linked data structures. Linked data structures are placed in a memory region where pointers are represented relative to the memory region's base address.

Zhang and Gupta [21] compress 32-bit integer values and 32-bit pointer values into 15-bit entities, applied to 32-bit C programs. For integer values, in case the 18 most significant bits are identical, *i.e.*, all 1's or all 0's, the integer value can be compressed into a 15-bit entity by discarding the 17 most significant bits. A pointer in an object's field is compressed if the 17 most significant bits of the referencing object's virtual address is identical to the 17 most significant bits of the referenced object's virtual address; only the 15 least significant bits are then stored. This is similar to ORA at first sight, but there is a subtle but important difference. Whereas ORA allows for compressing pointers in case the referenced object is reachable with an n-bit offset from the referencing object, Zhang and Gupta's approach requires that both objects reside in the same 2^n-bit memory region. This may lead to the situation where two objects are close to each other, *i.e.*, the difference between both object's virtual addresses is smaller than what can be represented by an n-bit offset, yet the pointers cannot be compressed.

Pairs of compressed 15-bit entity compressed are packed together into a single 32-bit word. Accelerating compression/decompression is done through data compression extensions to the processor's ISA.

Kaehler and Krasner [22] describe the Large Object-Oriented Memory (LOOM) technique for accessing a 32-bit virtual address space on a 16-bit machine. Objects in secondary memory have 32-bit pointers to other objects. Primary (main) memory serves as a cache to secondary memory. Object pointers in main memory are represented as short 16-bit indices into an Object Table (OT). This OT contains the full 32-bit address of the object. Objects need to be moved to main memory before they can be referenced. Translation between 32-bit pointers and 16-bit indices is performed when moving objects to main memory.

6.3 Object Header Compression

A number of studies have been done on compressing object headers which we briefly discuss here.

Bacon *et al.* [23] present a number of header compression techniques for the Java object model on 32-bit machines. They propose three approaches for reducing the space requirements of the TIB pointer in the header: bit stealing, indirection and the implicit type method. Bit stealing and indirection still require a condensed form of a TIB pointer to be stored in the header. Implicit typing on the other hand, completely eliminates the TIB pointer. Various flavors of implicit typing have been proposed in the literature, such as Big Bag of Pages (BiBOP) approach by Steele [24] and Hanson [25], a hybrid BiBOP/bit-stealing approach by Dybvig *et al.* [26], and Selective Typed Virtual Addressing [8].

Shuf *et al.* [27] propose the notion of prolific types versus non-prolific types. A prolific type is defined as a type that has a sufficiently large number of instances allocated during a program execution. In practice, a type is called prolific if the fraction of objects allocated by the program of this type exceeds a given threshold. All remaining types are referred to as non-prolific. Shuf *et al.* found that only a limited number of types account for most of the objects allocated by the program. They then propose to exploit this notion by using short type pointers for prolific types. The idea is to use a few type bits in the status field to encode the types of the prolific objects. As such, the TIB pointer field can be eliminated from the object header. The prolific type can then be accessed through a type table. A special value of the type bits, for example all zeros, is then used for non-prolific object types. Non-prolific types still have a TIB pointer field in their object headers. A disadvantage of this approach is that the number of prolific types is limited by the number of available bits in the status field. In addition, computing the TIB pointer for prolific types requires an additional indirection. The advantage of the prolific approach is that the amount of memory fragmentation is limited since all objects are allocated in a single segment, much as in traditional VMs.

7 Conclusion

Pointers in 64-bit address spaces require twice as much memory as in 32-bit address spaces. This results in increased memory usage which degrades cache and TLB

performance; in addition, physical memory gets exhausted quicker. This paper presented object-relative addressing (ORA) for implementation in 64-bit Java virtual machines. ORA compresses 64-bit pointers in object fields into 32-bit offsets relative to the referencing object's virtual address. The important benefit of ORA over prior work, which assumed 32-bit offsets relative to a fixed base address, is that ORA enables pointer compression for programs that allocate more than 4GB of memory. Our experimental results using Jikes RVM on an IBM POWER4 machine using SPECjbb and DaCapo benchmarks show that ORA incurs a statistically insignificant impact on overall performance compared to raw 64-bit pointer representation, while reducing the amount of memory allocated by 10% on average and up to 14.5% for some applications.

Acknowledgements

The authors would like to thank the anonymous reviewers for their valuable comments. Kris Venstermans is supported by a BOF grant from Ghent University. Lieven Eeckhout is a Postdoctoral Fellow with the Fund for Scientific Research—Flanders (Belgium) (FWO—Vlaanderen). This research is also partially supported by the IWT and HiPEAC.

References

1. Venstermans, K., Eeckhout, L., De Bosschere, K.: 64-bit versus 32-bit virtual machines for Java. Software—Practice and Experience 36(1), 1–26 (2006)
2. Adl-Tabatabai, A.R., Bharadwaj, J., Cierniak, M., Eng, M., Fang, J., Lewis, B.T., Murphy, B.R., Stichnoth, J.M.: Improving 64-bit Java IPF performance by compressing heap references. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO), pp. 100–110. IEEE Computer Society Press, Los Alamitos (2004)
3. Guyer, S.Z., McKinley, K.S.: Finding your cronies: static analysis for dynamic object colocation. In: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 237–250. ACM Press, New York (2004)
4. Hirzel, M., Diwan, A., Hertz, M.: Connectivity-based garbage collection. In: Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA), pp. 359–373. ACM Press, New York (2003)
5. Hirzel, M., Henkel, J., Diwan, A., Hind, M.: Understanding the connectivity of heap objects. In: Proceedings of the International Symposium on Memory Management (ISMM), pp. 36–39 (June 2002)
6. Cherem, S., Rugina, R.: Region analysis and transformation for Java programs. In: Proceedings of the 4th International Symposium on Memory Management (ISMM), pp. 85–96. ACM Press, New York (2004)
7. Lattner, C., Adve, V.: Transparent pointer compression for linked data structures. In: Proceedings of the Third 2005 ACM SIGPLAN Workshop on Memory Systems Performance (MSP), pp. 24–35. ACM Press, New York (2005)
8. Venstermans, K., Eeckhout, L., De Bosschere, K.: Space-efficient 64-bit Java objects through selective typed virtual addressing. In: Proceedings of the 4th Annual International Symposium on Code Generation and Optimization (CGO), pp. 76–86 (March 2006)

9. Mahlke, S.A., Hank, R.E., McCormick, J.E., August, D.I., Hwu, W.W.: A comparison of full and partial predicated execution support for ILP processors. In: Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA), pp. 138–149 (June 1995)
10. Alpern, B., Attanasio, C.R., Barton, J.J., Burke, M.G., Cheng, P., Choi, J.D., Cocchi, A., Fink, S.J., Grove, D., Hind, M., Hummel, S.F., Lieber, D., Litvinov, V., Mergen, M.F., Ngo, T., Russell, J.R., Sarkar, V., Serrano, M.J., Shepherd, J.C., Smith, S.E., Sreedhar, V.C., Srinivasan, H., Whaley, J.: The Jalapeño Virtual Machine. IBM Systems Journal 39(1), 211–238 (2000)
11. Blackburn, S.M., Garner, R., Hoffman, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: Java benchmarking development and analysis. In: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA), pp. 169–190. ACM Press, New York (2006)
12. Lilja, D.J.: Measuring Computer Performance: A Practitioner's Guide. Cambridge University Press, Cambridge (2000)
13. Dieckmann, S., Hölzle, U.: A study of the allocation behavior of the specjvm98 Java benchmarks. In: Guerraoui, R. (ed.) ECOOP 1999. LNCS, vol. 1628, pp. 92–115. Springer, Heidelberg (1999)
14. Blackburn, S.M., Cheng, P., McKinley, K.S.: Myths and realities: the performance impact of garbage collection. In: Proceedings of the joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS), pp. 25–36. ACM Press, New York (2004)
15. Kim, J.S., Hsu, Y.: Memory system behavior of Java programs: methodology and analysis. In: Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS), pp. 264–274. ACM Press, New York (2000)
16. Shuf, Y., Serrano, M.J., Gupta, M., Singh, J.P.: Characterizing the memory behavior of Java workloads: a structured view and opportunities for optimizations. In: Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS), pp. 194–205 (2001)
17. Chen, G., Kandemir, M., Vijaykrishnan, N., Irwin, M.J., Mathiske, B., Wolczko, M.: Heap compression for memory-constrained Java environments. In: Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 282–301. ACM Press, New York (2003)
18. Chen, G., Kandemir, M., Irwin, M.J.: Exploiting frequent field values in Java objects for reducing heap memory requirements. In: Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments (VEE), pp. 68–78. ACM Press, New York (2005)
19. Mogul, J.C., Bartlett, J.F., Mayo, R.N., Srivastava, A.: Performance implications of multiple pointer sizes. In: USENIX Winter, pp. 187–200 (1995)
20. Lattner, C., Adve, V.: Automatic pool allocation: improving performance by controlling data structure layout in the heap. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 129–142. ACM Press, New York (2005)
21. Zhang, Y., Gupta, R.: Data compression transformations for dynamically allocated data structures. In: Computational Complexity, pp. 14–28 (2002)
22. Kaehler, T., Krasner, G.: LOOM: large object-oriented memory for Smalltalk-80 systems. In: Readings in object-oriented database systems, pp. 298–307. Morgan Kaufmann Publishers Inc. San Francisco (1990)

23. Bacon, D.F., Fink, S.J., Grove, D.: Space- and time-efficient implementation of the Java object model. In: Magnusson, B. (ed.) ECOOP 2002. LNCS, vol. 2374, pp. 111–132. Springer, Heidelberg (2002)
24. Steele, Jr., G.L.: Data representation in PDP-10 MACLISP. Technical Report AI Memo 420, Massachusetts Institute of Technology, Artificial Intelligence Laboratory (September 1997)
25. Hanson, D.R.: A portable storage management system for the Icon programming language. Software—Practice and Experience 10(6), 489–500 (1980)
26. Dybvig, R.K., Eby, D., Bruggeman, C.: Don't stop the BIBOP: Flexible and efficient storage management for dynamically-typed languages. Technical Report 400, Indiana University Computer Science Department (March 1994)
27. Shuf, Y., Gupta, M., Bordawekar, R., Singh, J.P.: Exploiting prolific types for memory management and optimizations. In: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 295–306. ACM Press, New York (2002)