

# Improving 64-Bit Java IPF Performance by Compressing Heap References

Ali-Reza Adl-Tabatabai, Jay Bharadwaj, Michal Cierniak<sup>†</sup>, Marsha Eng,  
Jesse Fang, Brian T. Lewis, Brian R. Murphy, and James M. Stichnoth  
*Microprocessor Technology Lab, Intel Corporation*  
<sup>†</sup>*Microsoft Corporation*  
*james.m.stichnoth@intel.com*

## Abstract

*64-bit processor architectures like the Intel® Itanium® Processor Family are designed for large applications that need large memory addresses. When running applications that fit within a 32-bit address space, 64-bit CPUs are at a disadvantage compared to 32-bit CPUs because of the larger memory footprints for their data. This results in worse cache and TLB utilization, and consequently lower performance because of increased miss ratios. This paper considers software techniques for virtual machines that allow 32-bit pointers to be used on 64-bit CPUs for managed runtime applications that do not need the full 64-bit address space. We describe our pointer compression techniques and discuss our experience implementing these for Java<sup>1</sup> applications. In addition, we give performance results with our techniques for both the SPEC JVM98 and SPEC JBB2000 benchmarks. We demonstrate a 12% performance improvement on SPEC JBB2000 and a reduction in the number of garbage collections required for a given heap size.*

## 1. Introduction

Processor architectures such as the Intel® Itanium® Processor Family [8] (IPF) enable large applications (such as enterprise applications) to use a 64-bit address space instead of the traditional 32-bit address space. A well-known drawback of 64-bit addressing is that pointer-based data structures consume more memory than their 32-bit counterparts, and the application therefore loses performance as a result of additional cache misses, TLB faults, and paging [13]. The increased memory footprint is especially troubling when running an application whose memory requirements actually fit entirely within a 32-bit address space, and yet the application is forced to pay the price of wide pointers.

---

<sup>1</sup> Other brands and names are the property of their respective owners.

In this paper, we describe our experience with compressing 64-bit pointers into 32 bits within the framework of a Java virtual machine [5]. Java programs offer a key advantage over arbitrary C programs in that pointers are always distinguishable from non-pointers. This gives us wide latitude in choosing a pointer compression scheme. Furthermore, the compression can be performed completely within application software (i.e., the virtual machine) without depending on platform-specific hardware or operating system features, making it more portable across different architectures and operating systems.

### 1.1. Solution overview

We compress raw 64-bit pointers into 32 bits by representing a pointer as an unsigned 32-bit offset from the base of a contiguous memory region. We focus compression on pointers within the Java heap, because the vast majority of memory in a typical Java application consists of heap-allocated Java objects. The compressible pointers within those Java objects consist of vtable pointers and pointers to other Java objects. This means that a compressed vtable pointer is represented as an offset into the contiguous vtable area, and a compressed Java reference pointer is represented as an offset into the contiguous Java heap.

Before dereferencing a pointer loaded from the Java heap the system must first decompress it by adding the heap base address. Similarly, the system must compress a pointer by subtracting the heap base address, before storing the lower 32 bits into the heap. *Compression* and *decompression* are the operations used to convert between raw and compressed pointers.

Our compression technique is simple and straightforward. Its compression and decompression, however, require additional instructions and also frequently extend the critical path of some computations. When we started this work it was unclear whether the improvement in memory stalls would outweigh this overhead. Our results

show this is indeed true and that there is a significant improvement in performance across a range of benchmarks.

Most 64-bit architectures (including the IPF) can support 32-bit applications directly. However, our compression technique allows a virtual machine (VM) to run an application with the advantages of 32-bit addressing, but with more than 32-bits of storage. In particular, since the VM is running as a 64-bit application, it can use a full 4GB for the garbage-collected heap. It can allocate storage for code, virtual method tables, and other VM-allocated data structures outside of this 4GB space. Furthermore, using the shifted offset technique described below, the VM can expand the heap beyond the normal 32-bit addressing limit of 4GB.

Compression is not possible if there is so much data that some compressed pointers do not fit in 32 bits. In this event, the VM can revert to using uncompressed (raw) pointers. Most VMs require that the maximum heap size be specified at VM startup time (e.g., by using a command line argument), allowing the VM to decide whether to use 32-bit or 64-bit raw pointers. If the specified size is too large (e.g., more than 4GB), raw pointers will be used.

It is important to note that the solution we describe represents a single point within a larger spectrum of implementation choices. There are alternative ways to choose which pointers are compressed and how they are compressed. Broadly speaking, there are three categories of locations where pointers reside:

- Per-instance locations: vtable pointers and the references in both instance fields and array elements.
- Per-class locations: static fields, method pointers in vtables, and some fields in internal VM data structures.
- Other locations: various internal VM data structures and temporary storage such as memory stack, registers, and so on.

Our compressed pointer design focuses on per-instance pointer fields within the Java heap, because of their large contribution to the memory footprint of an application. We also compress object references contained in static fields because this allows the JIT to assume that all reference fields are of uniform size.

Our design considers only one compressed pointer representation (i.e., a 32-bit unsigned offset from the memory base), but alternative representations could be used as well. For example, pointers from one heap object to another could be represented as a 32-bit signed offset from the base of the first object. More complex functions could be used if the memory region is not contiguous. In addition, the minimum alignment requirement of all objects can be used to expand the 4 GB addressing limit by treating compressed pointers as shifted offsets. This shifting trick can be applied to compressed vtable pointers and compressed reference pointers.

Each such extension to the basic design introduces its own implementation complexities and performance considerations, and could lead to either better or worse performance. In this paper, however, we focus on the simple compression scheme.

In addition to deciding which pointers to compress, another system-wide decision has to be made: whether reference arguments and return values should be passed as compressed pointers or raw pointers. We chose to use raw pointer arguments since we believed this to be more efficient in JIT generated code. Furthermore, this minimized changes in our system. However, using compressed pointer arguments, or some combination of the two approaches, could also be considered.

## 1.2. System overview

Our solution has been implemented for Java [5] applications running on the ORP [1] virtual machine with the StarJIT [2] just-in-time compiler. Our techniques apply equally well to Common Language Infrastructure (CLI) [6] applications even though our current implementation supports only Java.

ORP is a high-performance VM that supports both Java and CLI applications. It was designed with performance and flexibility as the main goals, and its flexibility allows us to easily experiment with new optimizations. ORP's modular design allows us to dynamically load different garbage collectors (GCs) as well as multiple just-in-time (JIT) compilers, the most recent of which is StarJIT. The default garbage collector used by ORP performs parallel sliding compaction to maximize application throughput, and typically accounts for only a small portion (just 3% for SPEC JBB2000) of an application's execution time. Using StarJIT and the default GC, ORP's performance is competitive with the best commercial Java VMs.

The optimizing StarJIT compiler uses a single SSA-based intermediate representation and global optimization framework to compile both Java and CLI. It starts compilation by translating bytecodes to the StarJIT intermediate representation (STIR). After bytecode translation, StarJIT's global optimizer operates on this representation to perform mostly architecture-neutral classical optimizations (such as dead code elimination and redundancy elimination), as well as optimizations targeted to type-safe object-oriented programs (such as devirtualization and run-time check elimination). STIR's low-level operators and types expose finer-grain operations to global optimizations. Both STIR and the global optimizer directly support compressed pointers. STIR includes operators to manipulate compressed pointers, and types to represent compressed pointer values. The global optimizer uses optimizations that reduce in strength or height sequences of operations on compressed pointers.

After the global optimizer runs, StarJIT's code generator for IPF lowers STIR instructions into IPF code sequences, performs machine-level optimizations (such as register allocation and scheduling), generates runtime tables (such as those used for stack unwinding and exception handling), and emits the bits that the processor executes. The code generator does trace scheduling; its global scheduler schedules one trace at a time, and performs register allocation, predication and speculation during scheduling. Because the code generator runs after the global optimizer, and on a lower-level program representation, it can further optimize code sequences involving compressed references.

### 1.3. Organization of the paper

The remainder of the paper is organized as follows. Section 2 describes in detail the issues of compressing the object header, including the vtable pointer. Section 3 describes the more complex details of compressing reference pointers within the Java heap. Section 4 presents the performance results across a set of benchmark applications. Section 5 describes related work, and Section 6 presents conclusions.

## 2. Object header compression

### 2.1. Overview

In ORP, every object includes a standard header, which consists of a vtable pointer and a combined synchronization/hashcode value. The vtable pointer normally contains a raw pointer to a vtable structure. The synchronization value is partitioned into 16 bits to hold the thread ID of the current locker, 8 bits to hold the lock recursion count, and 8 bits to hold the default hashcode. If the recursion count overflows beyond 8 bits, the synchronization value is treated as a link to a fatter lock structure. This organization of the object header is fairly common across commercial and research VMs[9][11].

On a system with 64-bit pointers, it is natural to allocate 8 bytes for the vtable pointer. It is also convenient to allocate 8 bytes for the synchronization value, so that the virtual machine can assign object field offsets assuming an initial 8-byte alignment. This yields a per-object overhead of 16 bytes, compared to 8 bytes on a 32-bit system.

In the 64-bit implementation of ORP, we reduce the overhead to 8 bytes per object by using only 4 bytes for

the vtable pointer and 4 bytes for the synchronization value. Instead of being a raw pointer, the value in the vtable slot is treated as an unsigned offset from the base of the vtable area in memory. Figure 1 depicts the layout of the object header for the raw and the compressed cases on a 64-bit system. At ORP startup time, the VM allocates a single contiguous chunk of memory for holding all vttables. The vtable space must be contiguous, generally meaning that the VM must put a bound at startup on the amount of vtable space available. (If such a bound is unacceptable, there are ways to relax this requirement, but that is beyond the scope of this paper.) The interface function *orp\_get\_vtable\_base()* provides a pointer to the start of the vtable area. ORP guarantees that this is a constant function, so the value can be cached by the GC and emitted as a constant in JIT-generated code. The VM and GC components access an object's vtable information in similar ways, and both are similarly modified to support compressed object headers. Each time a vtable pointer is loaded from an object and dereferenced, we instead load the 32-bit value, decompress it, then dereference the resulting 64-bit pointer. Decompressing a 32-bit vtable pointer involves zero-extending it to 64 bits and adding the 64-bit value *orp\_get\_vtable\_base()*. On many processor architectures including the IPF, the zero extension operation can be folded into the 32-bit load instruction.

During object allocation, the vtable pointer is set by subtracting *orp\_get\_vtable\_base()* from the raw vtable address and storing the lower 32 bits into the object. Converting from raw to compressed vtable references may add an additional operation to the critical path, but in our experience, relatively few such operations are performed by the VM and GC components so the performance impact is not noticeable.

It is fairly common for a garbage collector (particularly a stop-the-world collector) to temporarily hijack part of the object header during garbage collection and use it for GC purposes. For example, it might overwrite an object's synchronization value with a forwarding pointer if the object is being moved. In this example, if the synchronization value is compressed to 32 bits, then the forwarding pointer can be treated as a compressed reference, using techniques described in Section 1.1.

Using similar techniques, the vtable pointer in an object could be narrowed even further, to 24 or perhaps 16 bits. If all vttables are aligned at 64-byte boundaries, then a 16-bit vtable pointer value can allow up to 4MB of total vtable space, which is more than adequate in our experience. However, this introduces an additional shift opera-

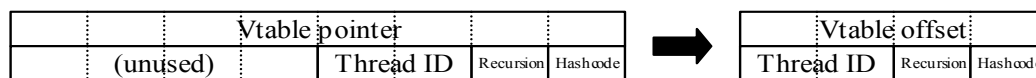


Figure 1: Compressing the object header

tion into the critical path when dereferencing an object's vtable. Furthermore, to have any impact on the live heap size, the application must have classes that not only have many instances, but also have 1- or 2-byte fields, such that moving a field into the free space in the object header causes the aligned object size to decrease. To illustrate, Java's String class contains only reference and 32-bit integer fields, so it would not benefit from shrinking its header by an additional 2 bytes.

## 2.2. JIT implementation and issues

The JIT-generated code uses the vtable pointer in two ways: for exact type checks and for dereferencing the

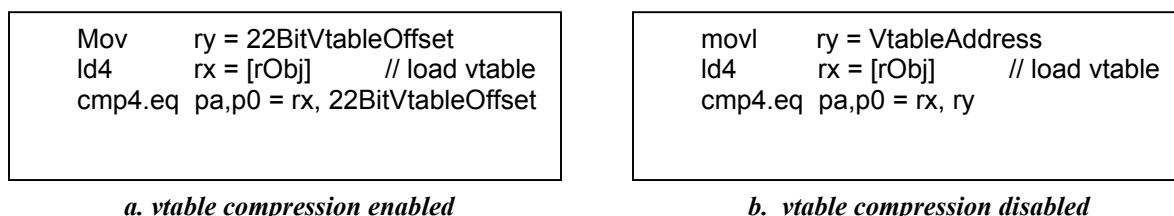


Figure 2: IPF instruction sequence for exact type check

The code sequence for dereferencing an object's vtable typically loads the 64-bit vtable pointer from the object, adding a constant (the virtual method's offset in the vtable), and dereferencing the result. When using compressed vtable pointers the address of the base of vtables (*orp\_get\_vtable\_base()*) needs to be added to the

compressed vtable pointer as well. Since in ORP this base address is a JIT-time constant, we can precompute its addition to the virtual method offset. This precomputation can be done ahead of the load of the vtable. The code sequence for IPF is shown in Figure 3.

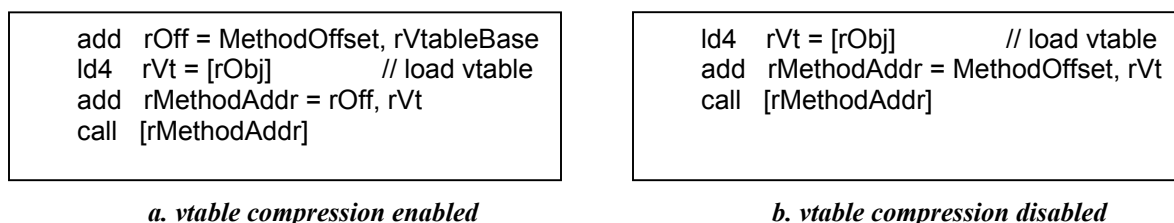


Figure 3: IPF instruction sequence for virtual method dispatch

## 3. Reference compression

This section describes the key implementation details of compressing references in ORP. As previously discussed, compression is applied to object references within Java objects and arrays, as well as static reference fields. Other references, such as those within argument lists and vtables, are not compressed. The following sections refer to the value *heap\_base*, which ORP exposes through the interface function *orp\_heap\_base\_address()*. ORP guarantees that this function returns a constant value during each ORP execution.

### 3.1. Null pointer representation

A key issue in compressing references is how to represent a null reference, both in compressed and raw form. To motivate the discussion, we define three types of null values:

- **Managed null:** the representation of a null value used in managed (i.e., JIT-generated) code.
- **Compressed null:** the compressed representation of a managed null value.
- **Platform null:** the representation of a null value used in native code.

On a 64-bit system without compressed references, both the managed null and the platform null are usually a 64-bit zero value, and the compressed null is not used.

When using compressed references, the simple compression method of subtracting *heap\_base* assumes that raw reference pointers fall within the contiguous address range [*heap\_base*..*heap\_base*+size-1]. However, this assumption is invalid if the managed null is represented by a 64-bit pattern of zeros, so the resulting compressed reference does not fit within 32 bits.

A practical choice of null representations should have the following properties:

- The compressed null should be a 32-bit pattern of zeros, to simplify the object allocation sequence. Most GC implementations clear the heap in large chunks before object allocation, satisfying the Java and CLI requirements that all object fields be initialized to 0 or null at allocation. If the compressed null is a zero, the allocation sequence remains simple and fast.
- The platform null should not be changed, to avoid major changes in native code (including VM and GC code) that manipulates Java objects. Native code is typically written assuming that null references are represented by the platform null.
- The managed null representation should be compatible with the VM's support for hardware null pointer exceptions (if any).

To satisfy these properties, our solution uses *heap\_base* to represent a managed null. If references are passed to or from native code (e.g. when invoking a native method or calling a VM runtime helper function), the VM marshals the values and translates between the managed null and the platform null. Such translations are relatively rare, in comparison to calls between JIT-generated methods. A minor consequence for the garbage collector is that it must not allow *heap\_base* to be the address of an allocated object, so as not to conflict with the managed null.

ORP does not currently support hardware null pointer exceptions on IPF. StarJIT makes null pointer checks explicit because it eases safety analysis, and it simplifies speculative scheduling of load instructions above null pointer checks. Nonetheless, it is straightforward to combine hardware null pointer exceptions and compressed heap references if the first few pages of the heap are marked as unreadable and unwritable. If the application attempts to access memory near the managed null address, the virtual memory system will automatically catch the error, and the VM can then throw a null pointer exception.

Note that a managed null needs special treatment for compressed references but not for compressed vtable pointers. This is simply because a valid vtable pointer is never null.

## 3.2. VM and GC implementation

To support compressed references, we modified the VM and GC to convert between compressed and raw references. We also changed null checks and other code that depends on the representation of a managed null. While numerous, these changes were straightforward except for those that translate between managed and platform nulls when transitioning to or from native code.

The garbage collector can support a 4GB-aligned heap (the benefits of an aligned heap are discussed below). Given a user-specified maximum heap size of *M* bytes, the collector allocates from the operating system a contiguous region of virtual address space of size *M*+4GB (without actually committing the space), and then commits an *M*-byte sub-region starting at a 4GB-aligned address.

## 3.3. JIT implementation

The JIT-generated code needs to decompress a reference to an object that is loaded from the heap before accessing the vtable or any field of that object. If the reference is instead passed as an argument or a return value from one method to another, we have chosen with our calling convention to decompress the reference prior to passing it into or out of a method. The method receiving a reference often immediately accesses its vtable which, being at offset 0 from the base of the object, is directly addressed by a raw reference. In such situations it is easier to hide the latency of the decompression in the method that passes the argument rather than in the method that receives it.

To get net performance gains from reference compression, we found it crucial to optimize unnecessary compression and decompression to reduce the critical path and code size, as described below. It is important to consider the phase ordering between classical optimizations and the pass that lowers the compression/decompression arithmetic to ensure that they are optimized.

- **Load-store forwarding:** If after loading a reference from the heap, it is later stored to the heap, the loaded compressed form should be directly used in the store. If the sole use of the loaded reference is the store to the heap, any inserted decompression can be eliminated as dead code.
- **Null checks and reference comparisons:** A null reference check can be performed as a comparison of the compressed reference to the compressed null, or as a comparison of the raw reference to the managed null. If both forms of the reference are available, then the one available first should be used. For example, the null check on a reference loaded from the heap may be better done as a com-

parison with the compressed null, and on an incoming raw reference method argument as a comparison with the managed null. If the heap base is 4 GB aligned, then we can use a 4-byte comparison to directly compare any two references, and perform a null check by comparing with 0. Otherwise, a comparison of two references may have to first uncompress one of them.

- **Reassociation of address expressions:** Computation of the address of a field or array element given a compressed object reference  $R_{comp}$ , involves two additions:  $((R_{comp} + heap\_base) + offset)$ . This expression may be reassociated and computed as  $(R_{comp} + (heap\_base + offset))$  if  $offset$  is available before the compressed reference. In the case of an object field,  $offset$  and  $heap\_base$  are both compile-time constants and  $(offset + heap\_base)$  can be precomputed. Even for a non-constant  $offset$  such as an array element, if the same element of multiple objects is accessed, the  $(offset + heap\_base)$  subexpression is common and should be precomputed. On the other hand, if multiple fields of the same object are accessed the decompressed reference  $(R_{comp} + heap\_base)$  is a common subexpression. The StarJIT compiler used to generate the experimental results presented in the next section does reassociation for object field accesses only. Additionally it performs local CSE of the decompression arithmetic  $(R_{comp} + heap\_base)$ .
- **Reference compression optimization:** JIT generated code must compress a raw reference before storing it into the heap. Compression of a reference involves subtracting  $heap\_base$  from the reference. If  $heap\_base$  is 4 GB aligned, however, its lower 32 bits are 0 and this operation is equivalent to masking off the top 32 bits. In the IPF architecture, a raw reference in a register can be stored with a 4-byte store, which just saves the lower 32 bits, so the compression operation is unnecessary.
- **Materializing heap base:** Since  $heap\_base$  is a commonly used run-time constant, it should be subject to CSE. Since this is a global constant (across both VM and JIT-generated code), we considered the benefit of dedicating a register to hold this value. This way  $heap\_base$  does not need to be put into a register upon each method entry, although this may hurt register allocation, and after any native code the VM must restore the register contents. We see some net benefit in our experiments below.

Reference compression also affects enumeration of live references by the JIT to the GC. The JIT must report whether references that are live at a garbage collection point are compressed or raw, so the GC can update references to any moved objects appropriately. It is a simple

matter to store this additional information along with the live reference bookkeeping, so it can be used at enumeration time.

## 4. Experimental Results

To demonstrate the benefit of our compression techniques, we give performance measurements for several benchmark applications with different ORP configurations. These numbers were taken on a 4-processor, 1.5 GHz Itanium® 2 machine with 6 MB L3 cache, 16 GB physical memory, and running Windows Server 2003, 64-bit edition. We measured performance on the seven component applications of SPEC JVM98 [4] as well as the SPEC JBB2000 [3] benchmark. Our techniques result in as much as 68% speedup for one of the SPEC JVM98 applications, and 12.7% for SPEC JBB2000. We considered the following ORP configurations. All configurations except the baseline include the first three optimizations described in Section 3.3. “load-store forwarding”, “null checks and reference comparisons”, and “reassociation of address expressions”.

- **Baseline:** Neither object headers nor references are compressed.
- **Compressed headers:** Only object headers are compressed.
- **Compressed references:** Only references within the heap are compressed. This configuration also uses the “reference compression optimization” from Section 3.3.
- **Both:** Both object headers and references are compressed. This configuration also uses the “reference compression optimization”.
- **Both, heap base in register:** The Both configuration is used; however, the VM ensures that the heap base is always stored in a specific register when executing JIT-compiled code. This is the “materializing heap base” optimization described in Section 3.3. This configuration also uses the “reference compression optimization”.
- **Both, unaligned heap base:** The Both configuration is used; however, the GC forces the heap to be allocated on other than a 4 GB boundary (normally the GC succeeds in acquiring a 4 GB aligned heap). This prevents the JIT from using the “reference compression optimization”, most likely leading to less efficient code.

### 4.1. SPEC JVM98

The first set of results is for the SPEC JVM98 benchmark suite using a 96 MB heap. Figure 4 shows the speedup over the baseline configuration. Positive values indicate better performance than the baseline. The results

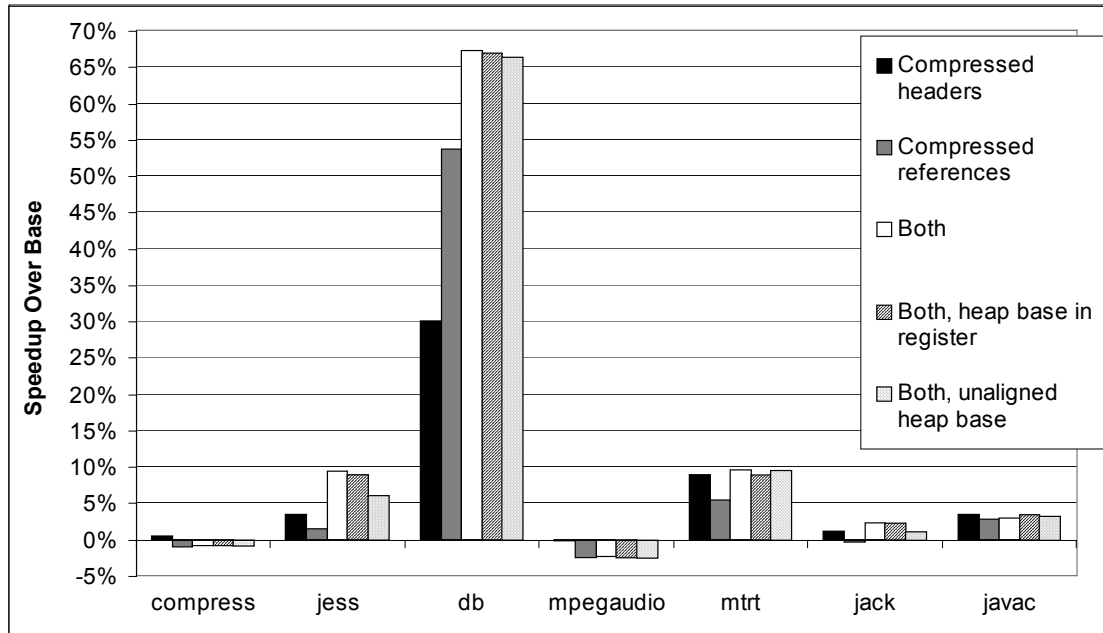


Figure 4: SPEC JVM98 performance relative to base configuration

show as much as 68% speedup (for db using both compressed headers and references), and no more than a 3% slowdown (for mpegaudio running compressed references alone).

Note that compressed headers almost always result in higher performance. This is because adding the vtable base can almost always be folded into other vtable address calculations, as discussed in Section 2.2.

Both compressed headers and compressed references dramatically improve db's performance. We have seen that db's performance is heavily influenced by cache size; pointer compression reduces the average object size, effectively decreasing the working set size.

Most other SPEC JVM98 applications improved between 4 and 10%. However, compress and mpegaudio slowed slightly because they have fewer pointer-based data structures. For these applications, our compression techniques save less memory and the overhead outweighs any gains. Neither materializing the heap base nor requiring a 4 GB aligned heap affected performance significantly, given an experimental noise margin of +/-1%. In general, SPEC JVM98 models client-side applications, which do not typically require significant amounts of memory and therefore benefit less from our techniques.

#### 4.2. SPEC JBB2000

SPEC JBB2000 is designed to model server applications, which are more likely to represent the applications run on Itanium® 2 machines. We ran the benchmark with each ORP configuration using a 4 GB heap. In addition to

comparing the six different ORP configurations, we used EMON 7.0 to probe the Itanium® 2 processor's performance counter registers [20] for detailed stall cycle information. Figure 5 shows the SPEC JBB2000 speedup over the base configuration. Compressed headers improve performance by 5.5%, compressed references by 6.5%, and the combination by 11.5%. The "materializing heap base optimization" increases the speedup to 12.7%. The apparent small improvement from using an unaligned heap base is unexpected but within a +/-1% experimental noise margin. The Itanium® 2 processor's performance counters allow us to classify every execution cycle as either useful work ("unstalled cycles") or stalled cycles ("stalls"). Since the Itanium® 2 has an in-order microarchitecture, the stalls are precise events. The most important stalls are:

- **Memory dependency stalls:** These result from cache misses.
- **L1D pipeline stalls:** These primarily result from DTLB misses.
- **Front-end stalls:** These result from instruction cache and ITLB misses.
- **Branch misprediction stalls.**

We measured and classified the stalls for each ORP configuration. Since SPEC JBB2000 is a throughput-oriented benchmark, we normalized the stall counts for each test by the throughput to get per-transaction stall counts. We then compared each set of normalized counts to the baseline counts, to understand the effect of compression on overall program behavior.

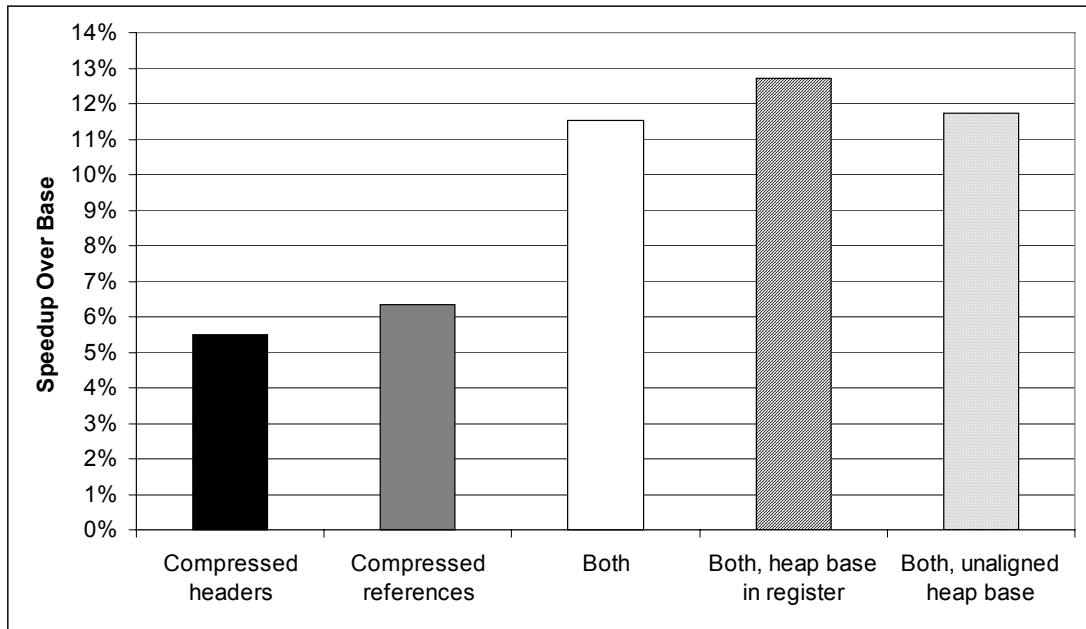


Figure 5: SPEC JBB2000 performance relative to base configuration

Figure 6 shows the reduction in cycles per transaction for each ORP configuration. Negative values indicate the cycles in these categories increased; there is an overall performance improvement as long as there is a net positive value, indicating a net cycle reduction. By far the largest reason for the performance improvement is a reduction in memory dependency stalls, followed by L1D pipeline stalls. This is a direct result of reducing object sizes: smaller footprints mean fewer cache misses and less opportunity for DTLB misses. The increase in

uninstalled cycles for the compressed references configuration is expected from the compression/decompression operations introduced. Overall, there are some minor changes to front-end and miscellaneous stall cycles. When multiple stalls occur in a cycle, the cycle is attributed to one stall category based on a predetermined priority ordering. In general the shifts in front-end and miscellaneous stalls are attributed to their being exposed when other higher priority stalls (mostly memory dependencies) disappear.

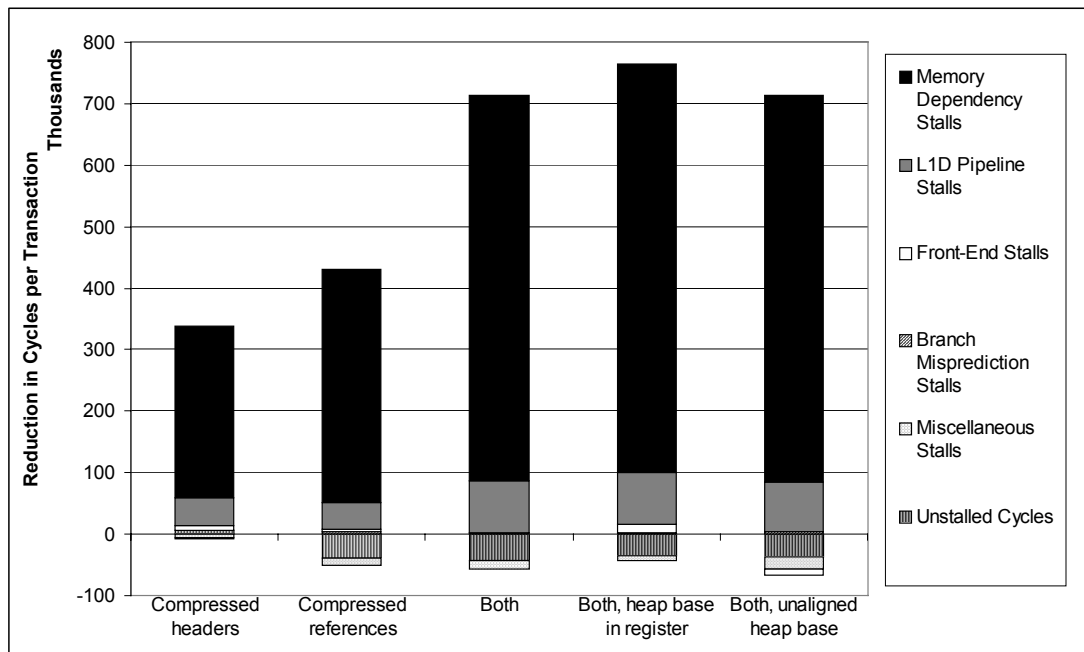


Figure 6: Reduction in cycles per SPEC JBB2000 transaction



	Reduction in heap space allocated	Reduction in GCs
Compressed headers	14.4%	13.2%
Compressed references	13.4%	14.2%
Both	27.3%	25.4%

Table 1: Reductions in heap space allocated and total GCs

Table 1 illustrates the reduction in heap space allocated and the resulting decrease in full GCs. We only consider three configurations plus the base, ignoring “heap base in register” and “unaligned heap base” because those options have no effect on object size. These reductions further show the impact of pointer compression: because compression reduces the object footprint, less memory must be allocated. This causes the heap to fill less quickly, which in turn reduces the need for GCs. While not measured for this paper, the reduction in allocated heap space means that running ORP with pointer compression requires a smaller heap to obtain the same performance as running without compression.

## 5. Related work

Supporting multiple pointer sizes has been considered before by teams porting code to architectures with larger pointer sizes. Reducing object size in Java and similar languages has also been considered. To our knowledge, the two topics have not previously been combined.

In [13], Mogul, et al., studied the effects of using 64-bit pointers for programs that ran on 32-bit systems. They compared C programs on a Digital Alpha system; the architecture simply zero-extended 32-bit pointers to 64-bits. The code was unchanged, so only the data size was affected. They found that, while performance was often unaffected by larger pointers, some programs had much lower performance if, for example, their working set no longer fit in the L2 cache. If the working set was too large for the L2 cache using either size, the performance difference was much smaller. Similar effects were seen for each cache level, TLB faults, and paging. Further information on Digital’s 32-bit to 64-bit address transition appears in [16][17].

Bacon, et al., [11] compared several schemes to reduce Java object size by shrinking object header fields. They describe several methods (related to our vtable pointer compression) to reduce storage for vtable pointers. If certain bits of these in all object headers are constant, they “steal” those bits to store other information (e.g., GC state). This did not hurt performance, perhaps because vtable decoding can be overlapped with other

operations on a multiple-issue processor. Our compressed vtable pointer scheme could be viewed as stealing 32 bits to pack the rest of the object header into 64 bits. We might compress headers further by stealing more bits and using other of their techniques, but we 64-bit align our object fields for better memory access performance. They also considered saving space with an index into a table of vtable pointers, but found this significantly hurt performance. Shuf, et al., [19] proposes vtable compression to a 4-bit index for the most common types; they show a space improvement, but omit other performance data.

Ananian and Rinard [12] describe additional techniques for reducing Java memory use. They use the index into a vtable pointer table scheme, as well as a number of other techniques. They use off-line program analysis to eliminate unused fields and bits of fields that are run-time constants, specialize classes and methods to increase the number of such fields, and reorder fields to improve object layout. Chen, et al., [14] focus on page-level compression to reduce program working set size. Dieckmann and Hölzle [15] did not try to optimize object size, but studied memory use in SPEC JVM98. They observed that object overhead can have a significant effect: there, one extra header word could increase allocation rate by about 20%.

A related technique is pointer *swizzling* [20][18], which can be used to support very large virtual memory spaces with smaller pointers. Pages from a large address space stored on, e.g., a disk, are mapped into a smaller addressable memory as needed. References in the large address space must be translated before use by translating, or swizzling, them into addresses in the smaller memory.

Hardware compression techniques compress cache [23][24][25] or DRAM [22] contents, increasing the effective capacity of these structures. The software technique described in this paper compresses contents throughout the memory system; it not only increases the effective capacity of the cache and DRAM, but also improves DTLB performance. Software compression leverages metadata indicating which values are pointers and which prefix value is redundant across pointers. Software compression also eliminates the compression and decompression latencies via scheduling and other optimizations.

## 6. Conclusions

This paper considers software techniques for pointer compression to improve Java performance on processors with 64-bit pointers. Our techniques include compressing both object headers and heap references.

We compress object headers to 8 bytes by compressing the 64-bit vtable pointer to a 32-bit offset into a memory region containing vtable structures. Because operations to decode the vtable offset can usually be overlapped or folded into other operations, this usually avoids decoding cost.

Similarly, we compress heap references in memory from 64 bits to a 32-bit offset from the base of the garbage-collected heap. While the concept is simple, we learned several lessons: the need for a special mapping for null references; that compiler optimizations are needed to reduce the cost of reference compression/decompression; and that garbage collection must support both compressed and raw pointers.

We studied the performance impact of these techniques on Java programs running on the ORP virtual machine on a 4-processor, 1.5 GHz Itanium® 2 machine with 16 GB memory. The two pointer compression techniques improved performance of SPEC JVM98 benchmarks overall. The improvements ranged from 68% on db to a 3% degradation on mpegaudio. For the SPEC JBB2000 benchmark, they improved the benchmark result by as much as 12% over a highly tuned baseline, and reduced heap space allocated by more than 27% and the number of garbage collections by 25%. We used the EMON 7.0 tool to examine performance counters and verified that memory dependence stalls, L1D pipeline stalls, and L3 cache misses were significantly reduced.

From previous studies of C programs, we anticipated that compressing pointers would reduce object sizes and the program working set, and so improve performance by reducing cache misses and TLB misses. The performance benefit we obtained for Java was even greater than had been reported for C. In particular, the ability in a managed runtime environment to recognize and compress heap references provides additional opportunities to reduce space usage. We expect our results will extend to other 64-bit processors and managed runtime environments.

## 7. References

- [1] M. Cierniak, M. Eng, N. Glew, B. Lewis, and J. Stichnoth. The Open Runtime Platform: A Flexible High-Performance Managed Runtime Environment. *Intel Technology Journal*, February 2003. <http://developer.intel.com/technology/itj/2003/volume07issue01>
- [2] A. Adl-Tabatabai, J. Bharadwaj, D-Y. Chen, A. Ghuloum, V. Menon, B. Murphy, M. Serrano, and T. Shpeisman. The StarJIT Compiler: A Dynamic Compiler for Managed Runtime Environments. *Intel Technology Journal*, February 2003. <http://developer.intel.com/technology/itj/2003/volume07issue01>
- [3] SPEC Java Business Benchmark 2000. Standard Performance Evaluation Corporation.
- [4] SPEC JVM98. Standard Performance Evaluation Corporation.
- [5] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*, Second Edition, Addison-Wesley, 1999.
- [6] *Common Language Infrastructure*, Standard ECMA-335. ECMA, December 2002. <http://www.ecma-international.org/publications/Standards/ecma-335.htm>.
- [7] *Intel® Architecture Software Developer's Manual*, Intel Corp., 1997.
- [8] *Intel® Itanium® Architecture Software Developer's Manual, rev. 2.1*. Intel Corp., October, 2002.
- [9] M. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño dynamic optimizing compiler for Java. *Proceedings of the ACM 1999 Java Grande Conference*, 1999, pp. 129-141.
- [10] P. Briggs and K. D. Cooper. Effective Partial Redundancy Elimination. *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI)*, Orlando, Florida, June 20-24, 1994. *SIGPLAN Notices* 29(6) (June, 1994), pp. 159-170.
- [11] D. F. Bacon, S. J. Fink, and D. Grove. Space- and Time-Efficient Implementation of the Java Object Model. B. Magnusson (Ed.): *ECOOP 2002 – Object-Oriented Programming, 16<sup>th</sup> European Conference*, Malaga, Spain, June 10-14, 2002, Proceedings. *Lecture Notes in Computer Science* 2374, Springer, 2002, pp. 111-132.
- [12] C. S. Ananian and M. Rinard. Data Size Optimizations for Java Programs. *Proceedings of the 21003 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*. San Diego, California, USA, June 11-13, 2003, pp. 59-68.
- [13] J. C. Mogul, J. F. Bartlett, R. N. Mayo, A. Srivastava. Performance Implications of Multiple Pointer Sizes. In *USENIX 1995 Technical Conference on UNIX and Advanced Computing Systems (USENIX 1995)*, New Orleans, Louisiana, USA, January 16-20, 1995, pp 187-200.
- [14] G. Chen, M. Kandemir, N. Vijaykrishnan, M.J. Irwin, B. Mathiske, and M. Wolczko. Heap Compression for Memory-Constrained Java Environments. To appear in *18<sup>th</sup> Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'03)*, Anaheim, California, October 26-30, 2003.
- [15] S. Dieckmann and U. Hölzle. A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks. In R. Guerraoui (ed.), *ECOOP'99 – Object-Oriented Programming, 13<sup>th</sup> European Conference*, Lisbon, Portugal, June 14-18, 1999, Proceedings. *Lecture Notes in Computer Science* 1628, Springer, 1999, pp. 92-115.
- [16] M. S. Harvey and L. S. Szubowics. Extending OpenVMS for 64-bit Addressable Virtual Memory. *Digital Technical Journal* 8(2), 1996, pp. 87-71.

- [17] T. R. Benson, K. L. Noel, R. E. Peterson. The OpenVMS Mixed Pointer Size Environment. *Digital Technical Journal* 8(2), 1996, pp. 72-82.
- [18] P. R. Wilson. Operating System Support for Small Objects. In *International Workshop on Object Orientation in Operating Systems*, pp. 80-86, Palo Alto, California, October, 1991.
- [19] Y. Shuf, M. Gupta, R. Bordawekar, J. P. Singh. Exploiting Prolific Types for Memory Management and Optimizations. In *Proceedings of the 29<sup>th</sup> Annual ACM Symposium on the Principles of Programming Languages (POPL 2002)*. Portland, Oregon, USA, January, 2002, pp. 295-306.
- [20] P. Wilson and S. Kakkad. Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Address Spaces on Standard Hardware. In *Proceedings of the 1992 IEEE International Workshop on Object Orientation and Operating Systems (IWOOS '92)*, Dourdan, France, 1992, pp 364-377.
- [21] Intel® Itanium® 2 Processor Reference Manual for Software Development and Optimization. Intel Corp., April 2003.  
<http://www.intel.com/design/itanium2/manuals/251110.htm>.
- [22] Tremaine, R., Franaszek P., Robinson, J., Schulz, C., Smith, T., Wazlowski, M., and Bland, P., "IBM Memory Expansion Technology (MXT)." In *IBM Journal of Research and Development*, Vol. 45, No. 2, march 2001.
- [23] Lee, J.-S., Hong, W.-K., Kim, S.-D. "Design and Evaluation of a Selective Compressed Memory System." In *Proceedings of the IEEE International Conference On Computer Design, VLSI in Computers and Processors*, Austin, 1999.
- [24] Yang, J., Zhang, Y., Gupta, R. "Frequent value compression in data caches." In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, 2000.
- [25] Zhang, Y., Yang, J., Gupta, R. "Frequent value locality and Value-Centric Data Cache." In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, Cambridge, MA, 2000.