Java Object Header Elimination for Reduced Memory Consumption in 64-bit Virtual Machines

KRIS VENSTERMANS, LIEVEN EECKHOUT and KOEN DE BOSSCHERE
Ghent University, Belgium

Memory performance is an important design issue for contemporary computer systems given the huge processor-memory speed gap. This paper proposes a space-efficient Java object model for reducing the memory consumption of 64-bit Java virtual machines. We completely eliminate the object header through Typed Virtual Addressing (TVA) or implicit typing. TVA encodes the object type in the object's virtual address by allocating all objects of a given type in a contiguous memory segment. This allows for removing the type information as well as the status field from the object header. Whenever type and status information is needed, masking is applied to the object's virtual address for obtaining an offset into type and status information structures. Unlike previous work on implicit typing, we apply TVA to a selected number of frequently allocated object types, hence the name Selective TVA (STVA); this limits the amount of memory fragmentation. In addition to applying STVA, we also compress the Type Information Block (TIB) pointers for all objects that do not fall under TVA.

We implement the space-efficient Java object model in the 64-bit version of the Jikes RVM on an AIX IBM platform and compare its performance against the traditionally used Java object model using a multitude of Java benchmarks. We conclude that the space-efficient Java object model reduces memory consumption by on average 15% (and up to 45% for some benchmarks). About half the reduction comes from TIB pointer compression; the other half comes from STVA. In terms of performance, the space-efficient object model generally does not affect performance, however for some benchmarks we observe statistically significant performance speedups, up to 20%.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors — *run-time environments*

General Terms: Design, Performance, Experimentation

Additional Key Words and Phrases: virtual machine, 64-bit implementation, typed virtual addressing, implicit typing, Java object model

1. INTRODUCTION

A well known design concern in today's computing systems is the large memory-processor speed gap — accessing main memory typically takes hundreds of processor cycles. One contributing factor to the memory gap is the amount of memory consumed by an application, *i.e.*, the more memory consumed, the more likely the

Contact information: Kris Venstermans, Lieven Eeckhout and Koen De Bosschere,
ELIS Department, Ghent University, Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium
Email: {kvenster,leeckhou,kdb}@elis.UGent.be

data will not fit into the processor's cache hierarchy, the more likely the application will have to access main memory. This is an important issue for 64-bit Java VM implementations. A recent study by Venstermans et al. [2006a] quantified that objects are nearly 40% larger in a 64-bit VM compared to a 32-bit VM. And about half this increase is due to the object header doubling in size.

This paper focuses on reducing the memory consumption of 64-bit Java VM implementations. Our approach to reducing the memory consumption of a 64-bit VM is to completely eliminate the object header. This is done through Typed Virtual Addressing (TVA) which means that the object type information is encoded in the object's virtual address, *i.e.*, all objects of the same type are mapped to the same contiguous memory segment. TVA enables to remove the Type Information Block (TIB) pointer field as well as the status field from the object header. As such, we are able to completely eliminate the 16 byte object header for non-array objects; for array objects, we only keep the 4 byte array length field. Accessing the TIB is then done by masking a number of bits from the object's virtual address, and using that as an offset in the TIB space that holds all the TIBs. Removing the status field from the object header is done by keeping GC bits and hash bits — 1 byte per object in our implementation — in so called side arrays. Our proposal does not apply TVA to all object types but only to a selected number of types that are frequently allocated, hence the name Selective TVA (STVA). The reason is that applying TVA to all object types would result in too much memory fragmentation because of memory pages being sparsely filled with only a few objects.

The idea of typed addressing or implicit typing is not new. Typed addressing has been proposed in the past with proposals such as Big Bag of Pages (BiBOP), typed pointers and others [Appel 1989; Dybvig et al. 1994; Hanson 1980; Shebs and Kessler 1987; Steele, Jr. 1997]. In fact, it was fairly popular in the 1970s, 1980s and early 1990s in various functional and logic programming languages. However, typed addressing has fallen into disfavor from then on because of the fact that all of these proposals applied typed addressing for all object types. As mentioned above, applying typed addressing to all objects results in memory fragmentation, and eventually performance degradation. With the advent of 64-bit Java implementations, a well designed typed virtual addressing mechanism becomes an interesting option for reducing the memory consumption of 64-bit Java VMs because the 64-bit virtual address space is huge which facilitates the implementation of implicit typing compared to 32-bit platforms.

In addition to removing the header for all TVA-enabled objects, we also compress the 64-bit TIB pointers to 32-bit pointers for all TVA-disabled objects. Previous work [Adl-Tabatabai et al. 2004] proposed pointer compression to all pointers (not just TIB pointers) in a 64-bit VM implementation. However, the limitation of compressing all pointers is that applications that require more than a 32-bit virtual address space cannot benefit from this pointer compression approach. Applying pointer compression to TIB pointers only does not suffer from this limitation; the case where more than a 32-bit virtual address space is needed for type information is highly unlikely.

We implement our space-efficient header approach in the 64-bit Jikes RVM and evaluate the reduction in memory consumption and the impact on performance

on an AIX IBM POWER4 system. (Previous work on typed addressing did not report the impact on performance of typed addressing, and in most cases did not even quantify the amount of reduction in memory consumption.) In addition, we apply statistics in order to make statistically valid conclusions. We conclude that our space-efficient Java object model reduces memory consumption by on average 15% (up to 45% for some benchmarks). On average, half the memory consumption reduction comes from TIB pointer compression; the other half comes from STVA. In terms of performance, the space-efficient Java object model has a net zero impact on performance for many benchmarks. For a small number of benchmarks we observe a statistically significant degradation in performance by only a few percent (no more than 5%). And for a number of other benchmarks we observe statistically significant performance speedups, up to 20%. In general though, we conclude that the space-efficient Java object model substantially reduces memory consumption without significantly affecting performance; however, performance improvements are observed for some benchmarks.

This paper extends our previous work [Venstermans et al. 2006b] by proposing an online STVA implementation; the prior work proposed an offline STVA implementation which required a profiling run to determine what objects to make TVA-enabled. This paper also extends the prior work by showing how the complete header can be eliminated for TVA-enabled objects. The prior work reduced the scalar object header from 16 bytes down to 4 bytes; the current work completely eliminates the object header.

2. THE 64-BIT JAVA OBJECT MODEL

The object model is a key part in the implementation of an object-oriented language and determines how an object is represented in memory. A key property of object-oriented languages is that objects have a run-time type. Virtual method calls allow for selecting the appropriate method at run-time depending on the run-time type of the object. The run-time type identifier for an object is typically a pointer to a virtual method table.

An object in an object-oriented language consists of the object data fields along with a header. For clarity, we refer to an object as the object data plus the object header throughout the paper; the object data refers to the data fields only. The object header contains a number of fields for bookkeeping purposes. The object header fields and their layout depend on the programming language, the virtual machine, *etc.* In this paper we assume Java objects and we use the Jikes RVM in our experiments. The object model that we present below is for the 64-bit Jikes RVM, however, a similar structure will be observed in other virtual machines, or other object-oriented languages. An object header typically contains the following information, see Figure 1(a):

—The first field is the *TIB pointer field*, *i.e.*, a pointer to the Type Information Block (TIB). The TIB holds information that applies to all objects of the same type. In the Jikes RVM, the TIB is a structure that contains the virtual method table, a pointer to an object that represents the object type and a number of other pointers for facilitating interface invocation and dynamic type checking. The TIB pointer is 8 bytes in size on a 64-bit platform.

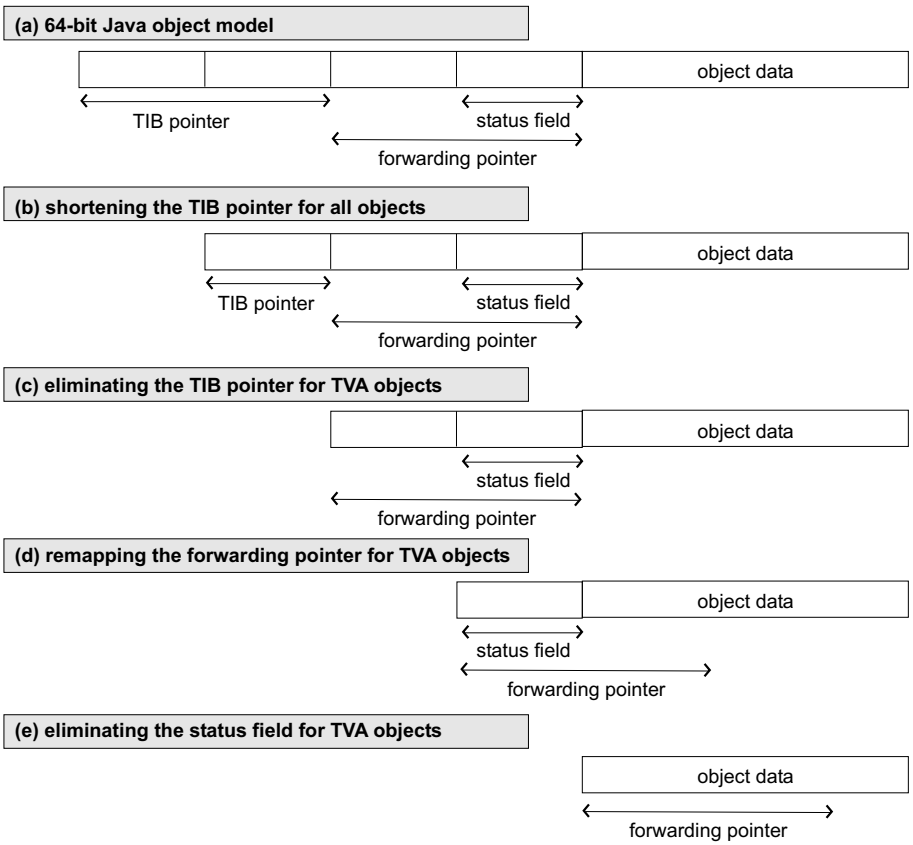

Fig. 1. The Java (scalar) object models studied in this paper.

—The second field is the *status field*. The status field can be further distinguished in a number of elements.
  —The first element in the status field is the *hash code*. Each Java object has a hash code that remains constant throughout the program execution. Depending on the chosen implementation in the Jikes RVM, the hash code can be a 10-bit hash field in the header or a 2-bit hash state.
  —The second element in the status field is the *lock element* which determines whether the object is being locked. All objects contain such a lock element. A thin lock field [Bacon et al. 1998] in the Jikes RVM is 20 bits in size.
  —The third element is related to *garbage collection*. This could be a single bit that is used for marking the object during a mark-and-sweep garbage collection. Or this could be a number of bits (typically two) for a copying or reference counting garbage collector.

—The third field is the *forwarding pointer*. The forwarding pointer is used for keeping track of objects during generational or copying garbage collection and is 8 bytes in size. The forwarding pointer overwrites the hash code and lock element in the status field, but not the garbage collection bits, see Figure 1(a).

The garbage collection bits are chosen as the least significant bits so that they do not get overwritten by the forwarding pointer.

So far, we considered non-array objects. For array objects there is an additional 4 bytes length field that needs to be added to the object header. As a result, for array objects the header field requires at least 20 bytes. But given the fact that alignment usually requires objects to start on 8 byte boundaries on a 64-bit platform, the array object header typically uses 24 bytes of storage.

3. ELIMINATING THE HEADER IN THE 64-BIT JAVA OBJECT MODEL

We eliminate the header of 64-bit Java objects in a number of steps. Our initial Java object model is the 16 byte header as shown in Figure 1(a) — we limit the discussion to scalar objects for now, and will discuss array objects later.

—We first reduce the TIB pointer size from 64-bit to 32-bit through pointer compression as proposed by [Adl-Tabatabai et al. 2004]. This is shown in Figure 1(b). This object model implies that all the TIBs are allocated in a contiguous virtual address space that is small enough to be accessed using a 32-bit offset. The TIB pointer is then computed by adding the 32-bit TIB pointer stored in the object header to a 64-bit TIB base pointer. TIB pointer compression is applied to all objects.

—As a second step we apply Selective Typed Virtual Addressing (STVA) to completely eliminate the TIB pointer from the object header, see Figure 1(c). STVA applies Typed Virtual Addressing (TVA) to a selected number of object types.

—In the third step, we map the forwarding pointer in a different way so that the forwarding pointer overlaps with the 4 byte status field and the first four bytes of the object data, see Figure 1(d). Note that the object data is already copied during garbage collection whenever the forwarding pointer gets used. As such, we can freely overwrite the first four bytes of the object data. (Obviously, this cannot be done for data structures belonging to the garbage collector itself — this is only of concern whenever the garbage collector (or the entire VM) is written in Java.) This layout requires to move the GC status bits from the least significant status field bit positions to the most significant status field bit positions so that the GC bits are not overwritten by the forwarding pointer.

—As a final step, we remove the status field from the object header, see Figure 1(e). The end result is that the object header is completely eliminated for all TVA-enabled objects.

In the following sections, we discuss STVA in great detail since it is the key enabler for completely eliminating the object header.

4. SELECTIVE TYPED VIRTUAL ADDRESSING

This section explains the idea and the implementation details behind Selective Typed Virtual Addressing (STVA) for 64-bit Java objects. We first detail on the TVA 64-bit Java scalar object model followed by the TVA array object model. We then go through a number of virtual machine implementation issues that follow from the TVA object models.

4.1 The scalar TVA object model

We consider two TVA Java object models: the small-header object model and the no-header object model.

4.1.1 *The small-header object model.* The small-header TVA object model eliminates the TIB pointer from the object header and remaps the forwarding pointer to overwrite the 4 byte status field and the first 4 bytes of the object data, see Figure 1(d). This implies that the minimum size occupied by a TVA-enabled object is 8 bytes: 4 bytes of object header and 4 bytes of data. We will refer to the obtained object model in Figure 1(d) as the small-header object model throughout the paper; this is the object model presented in our prior work [Venstermans et al. 2006b].

4.1.2 *The no-header object model.* The no-header object model extends on the small-header object model by eliminating the 4 byte status field, see Figure 1(e). This requires that we eliminate the lock, the hash and the GC elements, as well as the forwarding pointer from the object header. This is done as follows.

—We eliminate the GC elements from the object header by storing the GC elements in what we call a *side array*. We implement a side array every two pages on which TVA-enabled objects are allocated and allocate one byte per object in the side array. Note that depending on the garbage collector only one or two bits are used from the allocated byte in the side array. The position in the side array for a given object is determined by the position of the object on the memory page. Note that this can be done because all objects in a TVA region are of the same type and thus are equally sized. By consequence, during garbage collection, we do not adjust the GC elements in the header for TVA-enabled objects but we adjust the GC elements in the side arrays. The index in the side array is computed from the object's virtual address which incurs an additional overhead compared to the default and small-header object models.

—Dealing with the lock element in the no-header format is done along what is described in [Bacon et al. 2002]. Objects from a class that contains at least one `synchronized` method (or at least one of the class methods contains the `synchronized(this)` statement) have an additional implicit field member that contains the lock. This is a 4 bytes field in our implementation. This implicit lock field scheme cannot be applied to arrays or in cases where a lock is taking on an object. In these latter cases, a lock object is then created in the lock nursery [Bacon et al. 2002].

—The hash elements in the Java object model can take three states: *unhashed*, *hashed* and *hashed-and-moved* [Bacon et al. 2002; Agesen 1999]. For the first two states, unhashed and hashed, the hash code is calculated from the object's address. In case the garbage collector moves an object that is in the *hashed* state, its state then changes to *hashed-and-moved* and the hash code is attached to the end of the new version of the object. In the traditional Java object model as well as in the small-header object model, these three states are encoded using two hash bits. In the no-header object model on the other hand, we store only one bit per TVA-enabled object in the side arrays just described. The hash bit in the side array is zero if the object is *unhashed*; the hash bit in the side array is set

if the object is *hashed*. When a *hashed* object is moved by the garbage collector, we TVA-disable the object, *i.e.*, the object moves from the TVA space to the non-TVA space.

—The forwarding pointer, whenever needed during garbage collection, overwrites the first 8 bytes of the object data. This implies that the minimum size for a TVA-enabled object is 8 bytes — this is the same as for the small-header object model.

4.2 The array TVA object model

The array TVA object model more or less follows the same lines as the scalar TVA object model, however, there are some peculiarities in relation to memory management. In case of a copying collector, the memory allocator typically uses a bump pointer to allocate new objects, *i.e.*, the bump pointer is incremented by the size of the newly allocated object. In case of a mark-sweep collector, at least in the Jikes RVM, the memory allocator works with fixed-sized cells. The choice of the memory management method affects the array TVA object model.

4.2.1 *The small-header object model.* The small-header TVA array object model has an 8 byte header consisting of a 4 bytes status field and a 4 bytes array length field. We can select all arrays of all lengths to be TVA-enabled for a copying collector. Although selecting all arrays is also possible in case of a mark-sweep collector, this would result in a considerable runtime overhead and memory fragmentation because of the fixed-sized cells in the Jikes RVM implementation. As such, in case of a mark-sweep collector, we only select a single array length to be TVA-enabled.

4.2.2 *The no-header object model.* The no-header TVA array object model eliminates the complete header for array objects except for the 4 byte array length field. In order not to incur a large runtime overhead, we select at most one array length on which to apply TVA for both the copying and the mark-sweep collectors. The underlying reason is that a single array length eases accessing the side arrays; the side array index can be computed directly from the object's address.

4.3 Implications of the TVA object model

We now detail on a number of implications because of the TVA Java object model. Although some of these issues are geared towards our implementation in the Jikes RVM on an IBM AIX system, similar issues will need to be taken care of on other systems.

4.3.1 *Memory allocation.* The general idea behind Typed Virtual Addressing is to devote segments (large contiguous chunks of memory) in the virtual address space to specific object types. This means that the object type is implicitly encoded in the object's virtual address. Object types that fall under TVA are then allocated in particular segments of the virtual address space. For example, all objects of type A get allocated in the virtual address space segment with addresses ranging from address `0x04FF FFFE 0000 0000` to address `0x0500 0000 0000 0000`; all objects of type B then get allocated in the virtual address space segment in the range `0x0500 0000 0000 0000` to `0x0500 0002 0000 0000`.

The virtual memory address of a Java object in an STVA-enabled VM imple-

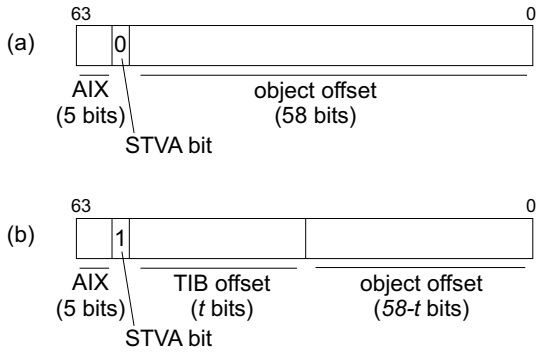

Fig. 2. The 64-bit virtual address for a TVA-disabled object (a) and for a TVA-enabled object (b).

mentation is depicted in Figure 2. The five most significant bits are AIX-reserved bits and should be set to zero. The following bit (bit 58) is the STVA bit that determines whether the given object falls under TVA. This divides the virtual address space in two regions, the TVA-disabled region and the TVA-enabled region. Note that although we consume half of the virtual address range for TVA-enabled object types, we leave 2^{58} bytes for TVA-disabled object types. If bit 58 is set, then the object is a TVA-enabled object, *i.e.*, the object follows the TVA Java object model detailed in section 4.1. If bit 58 is not set (the object type is a TVA-disabled type), the object falls under the default Java object model from Figure 1(a). In the latter case (a TVA-disabled object type), the least significant 58 bits determine the object's offset, see Figure 2(a). In case of a TVA-enabled object, see Figure 2(b), the next t bits of the virtual address constitute the TIB offset (t equals 25 in our implementation). The TIB offset determines in what memory segment the objects of the given type reside. By doing so, an object type specific memory segment is a contiguous chunk of memory of size 2^{58-t} bytes; this is 8GB in our implementation. The least $(58 - t)$ significant bits are the object offset bits (33 bits in our implementation). These bits indicate the object's offset within its type specific segment.

In order to support this memory layout, we obviously need to modify the memory allocator to support TVA. We need to keep track of multiple allocation pointers that point to the free space in the object type specific segments in order to know where to allocate the next object of the given object type. The selection of an individual allocation pointer requires an extra indirection for TVA-enabled object types. We eliminated this additional indirection by refactoring the code, *i.e.*, by inlining the allocation pointer array.

Another peculiarity related to the no-header TVA memory allocators is that we know that all objects within a type-specific segment have equal sizes. With this knowledge we can layout fixed sized cells into the TVA-enabled regions, prior to allocation. This layout will include proper alignment, so that we are able to remove the alignment burden from the memory allocator.

Yet another peculiarity relates to copying collectors. A traditional copying collector needs to figure out the size of the object to be allocated. This is done by

```
                                ;; R3 contains the object's virtual address
    tst R3, 0x0400 0000 0000 0000 ;; test bit 58 of virtual address
    bre L2                      ;; jump to L3 in case bit is not set

L1:                             ;; TVA-enabled object: mask the TIB offset
                                ;; from the object's virtual address
    rsh R4, R3, (64 - FIXED_BITS - NUM_TIB_BITS)
    lsh R4, R4, 3               ;; align offset to 8 bytes
    add R4, TIB_BASE, R4        ;; add the TIB offset to the base TIB pointer;
                                ;; the constant TIB_BASE equals the real base TIB
                                ;; pointer with bit 58 equal to zero -- as an
                                ;; optimization, bit 58 is not masked away.
    jmp L3

L2:                             ;; TVA-disabled object: read the TIB
    ld R4, R3, TIB_OFFSET       ;; pointer from the object's header

L3: ...                         ;; R4 contains the TIB value
```

Fig. 3. Computing an object's TIB pointer in an STVA-enabled VM implementation.

accessing the TIB, reading the pointer that points to the object that represents its class, and retrieving the object size from the class object. In our TVA-aware copying collector, we keep track of the object sizes for the various object types in an array structure. As such, a single array lookup yields us the object size to be allocated.

4.3.2 *TIB access.* In an STVA-aware VM implementation, reading the TIB pointer changes compared to a traditional VM implementation. In a traditional implementation (without STVA), the TIB pointer is read from the object header through a load instruction. In an STVA-aware VM implementation, we make a distinction between a TVA-enabled object and a TVA-disabled object. This is illustrated in pseudo-code in Figure 3. A TVA-disabled object follows the traditional way of getting to the TIB pointer. A load instruction reads the TIB pointer from the object header. For a TVA-enabled object, the TIB pointer is computed from the object's virtual address. This is done by masking the TIB offset from the virtual address and by adding this TIB offset to the TIB base pointer — all the TIBs from all object types are mapped in a limited address space starting at the TIB base pointer. The size of the TIB space is limited to 256MB in our implementation; this comes from the 25-bit TIB offset that we use, see Figure 2, along with a 3-bit shift left for 8 byte alignment. Note again that this is not a hard limit and can be easily adjusted by changing the address organization from Figure 2 in case a 256MB TIB space would be too small for a given application (which is unlikely for contemporary applications).

Due to the conditional jump for determining the TIB pointer, see Figure 3, our STVA-enabled implementation clearly has an overhead compared to a traditional VM implementation. The single most impediment to a more efficient implementation is the branch that is conditionally dependent on whether the object is TVA-enabled or TVA-disabled. Unfortunately, in our PowerPC implementation we could

not remove this conditional branch through predication. Nevertheless, this could be a viable solution on ISAs that support predication, for example through the `cmov` instruction in the Alpha ISA, or through full predication in the IA-64 ISA.

As an optimization to computing the TIB pointer, we limit the frequency of going through the relatively slow TIB access path.[1] This is done by marking the class tree with the TVA-enabled object types. A subtree is marked in case all types in this subtree are TVA-disabled. The TIB access then follows the fast TIB access path as in a non STVA-aware VM.

Since the TIB offset is computed from an object's virtual address, the position in memory of the TIB is obviously related to the object type specific memory segment. We cannot position the TIB independently from the object type specific memory segment. To avoid this problem, we make sure we first allocate the TIB in the TIB space. This will give us the TIB offset to be used for all objects of the given TVA object type. Once the type specific memory segment for a TVA object type is properly initialized, TVA-enabled objects can be allocated in it.

4.3.3 *Impact on garbage collection.* Implementing TVA obviously also has an impact on garbage collection. In this section we discuss garbage collection issues under the assumption of a generational garbage collector which is a widely used garbage collector type. Similar issues will apply to other collectors though. In a generational collector, there are two generations, the nursery and the mature generation. Objects first get allocated in the nursery. When the nursery fills up, a nursery collection is triggered and reachable objects are copied to the mature generation. New objects then get allocated from an empty nursery. This goes on until also the mature generation fills up. When the mature generation is full, a full heap collection is triggered.

In the original Jikes RVM implementation with a generational collector, the nursery and mature generations consist of contiguous spaces. This means that there is one or two contiguous spaces for the nursery and mature generations. In an STVA-aware VM implementation, contiguous memory segments are defined for specific object types that fall under TVA, but the union of all these memory segments is no longer contiguous. Because the nursery and mature spaces need to fall within all type-specific memory segments, these spaces can obviously no longer be contiguous. As such, we end up with non-contiguous spaces in both the nursery and mature generations. The nursery generation now consists of a contiguous space for TVA-disabled object types, and a non-contiguous space for TVA-enabled object types. The mature generation is constructed in a similar way. This is illustrated in Figure 4.

Jikes RVM however, works with contiguous spaces. Jikes RVM identifies a space by a SpaceDescriptor, a numerical value encoding the nature, size and starting address of the space. In order to be able to use non-contiguous spaces in our TVA-aware VM, we extended Jikes RVM's implementation of a space. In our system, we identify a space by the combination of its starting address and its mask. If we represent a space i by S_i, its mask by M_i, and its starting address as B_i, and if we

[1] This is only possible in the offline STVA implementation, see later for a discussion on the offline STVA implementation.

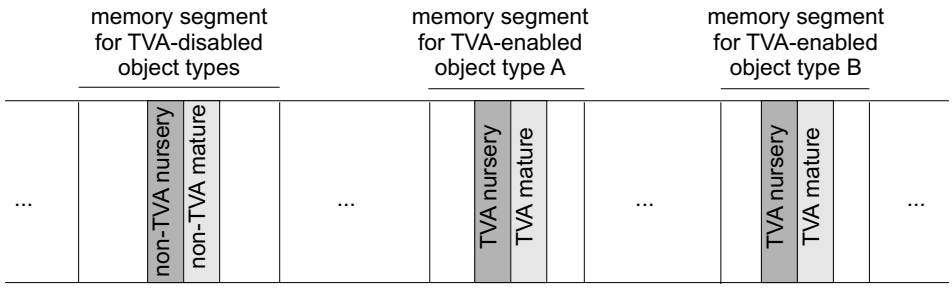

Fig. 4. Mapping the nursery and mature spaces in the virtual address space in a TVA-aware VM.

represent an address by A, then the following is true by definition: $B_i \& M_i = B_i$ and $A \in S_i \Leftrightarrow A \& M_i = B_i$, with & being the bitwise 'and' operator. A mask M consists of one or more series of `'1'`s and one or more series of `'0'`s; the following bit patterns are examples: `'00..011..100..0'`, or `'11..100..0'`. A contiguous space is just a special case in which the mask has a leading series of `'1'`s followed by a trailing series of `'0'`s, *i.e.*, the mask looks as follows: `'11..100..0'`. A non-contiguous space has a mask of any other form that does not consist of a leading series of `'1'`s followed by a trailing series of `'0'`s, for example `'00..011..100..0'`, `'11..100..011.100..0'` or `'00..011..100..011..100..0'`. We also have a simple rule to check if two spaces are non-overlapping — this is needed when allocating spaces: $\neg(A \in S_i \wedge A \in S_j) \Leftrightarrow (M_i \& M_j \neq 0) \wedge ((B_j \& M_i) \neq (B_i \& M_j)), \forall i, j$. Note that the mask is part of the definition of a space. As such, each space has a dedicated mask that does not need to be computed at run time; however, the mask is part of the TVA-aware VM implementation.

It is also interesting to point out that because of the fact that there are separate GC spaces for TVA-disabled/enabled objects, this opens up a number of opportunities for garbage collection. For example, different garbage collection strategies could be employed in different spaces of the nursery, or the TVA-enabled objects could be pretenured, *etc.* For example, Shuf et al. [2002] use non-copying collectors for prolific object types. In this paper however, we make no change in garbage collection strategy between TVA-disabled/enabled spaces, because we want to quantify the impact of STVA on the space efficiency of the TVA-aware Java object model, without intrusion of other techniques. Type-specific garbage collection strategies and related techniques will be studied in future work.

5. STVA TYPE SELECTION

As mentioned before, we do not apply TVA to all objects. Object types that are allocated infrequently would consume memory pages that are only sparsely filled with objects. This would result in too much memory fragmentation. As such, in order to limit memory fragmentation we need to limit the number of object types on to which TVA is applied. We believe that this is a key difference to prior work on typed virtual memory addressing. Prior work applied TVA to all object types. In this paper we propose to limit TVA to only a subset of well chosen object types in order to control the amount of memory fragmentation while pertaining the benefits of typed virtual addressing. We now explore two approaches to selecting object

types on which to apply TVA, namely an offline selection strategy and an online selection strategy.

5.1 Offline STVA type selection

In our offline STVA implementation, we apply the following strategy for making an object type TVA-enabled. In order to select an object type to fall under TVA, the object type needs to apply to one of the following criteria. First, an object type needs to be allocated frequently, and second, its instances are preferably long-lived. In our first criterion we make a selection of object types of which a sufficient amount of objects is allocated. Through a profiling run of the application, we collect how many object allocations are done for each object type, and what the object size is for each object type. Once this information is collected, we compute for each type the total number of allocated header bytes (16 bytes per instance), and we compute the percentage volume of these header bytes in relation to the total number of allocated bytes. We then select object types for which this percentage volume exceeds a given *memory reduction threshold (MRT)*.

In our second criterion we limit the scope to long-lived objects because long-lived objects are likely to survive garbage collections. These objects will thus remain in the heap for a fairly long period of time. Giving preference to long-lived objects under TVA maximizes the potential reductions in memory consumption. In order to classify object types into long-lived and short-lived object types, we take a pragmatic approach and inspect a profile run of the application for objects that survive garbage collections. In these runs we use a fairly large heap in order to identify truly long-lived objects. For those objects that survive a garbage collection, we again compute the percentage volume of the header bytes in relation to the total number of bytes surviving the collection. We then retain object types for which this percentage volume exceeds the *long-lived memory reduction threshold (LLMRT)*.

5.2 Online STVA type selection

An important disadvantage of the offline STVA type selection method is that a profiling run is needed for determining on what object types to apply TVA. This is not practical in a Java context. Therefore we now propose on online STVA type selection mechanism. The mechanism that we propose is a simple but effective approach. When re-compiling a method in the VM, we TVA enable all the object types that are allocated within these re-compiled methods. The underlying idea is that frequently executed methods, so called hot methods, are scheduled for re-compilation and re-optimization; if these methods allocate objects, they will allocate lots of these objects. In other words, the types of the objects allocated in methods that are scheduled for optimization, are likely to be frequently allocated object types. By consequence, these object types are good candidates for STVA selection. Note that objects allocated along infrequent paths, *e.g.*, exception handling, may be excluded from being selected. Infrequent paths such as exception handling could be detected through program analysis, or as the complement of hot paths detected through sampling. In this paper, we took a pragmatic approach by TVA-enabling objects along both frequent and infrequent paths when recompiling methods; this is a simple approach that yields good performance numbers.

Note that the online STVA type selection method is different from the offline

approach. The reason is that a direct translation of the offline approach into an online approach would be fairly complex. This translation would require that we keep track of the amount of bytes allocated for each type at run time. In addition, we have to determine *when* to convert an object type from TVA-disabled to TVA-enabled. And once we have determined when to convert an object type, we then have to recompile the methods allocating this object type. This is a fairly complex mechanism. Instead we have chosen for a much simpler alternative that triggers TVA for objects allocated in recompiled methods; this is motivated by the fact that methods need to be recompiled anyway in order to enable TVA for objects allocated in these methods. So, as for the *when* part, we choose recompilation time for triggering TVA, and for the *which object types* part, instead of determining frequently used object types, we simply select all objects allocated in hot methods. This simple approach showed to perform well in practice, as we will demonstrate in the evaluation section of this paper.

In case of the no-header object model, we do not select array types online because it is difficult for an online mechanism to select what array length to support under TVA. Also, in order to limit the total number of STVA types, we limit the number of TVA-enabled object types to a maximum which is 80 types in our implementation (which is about the maximum observed through our offline STVA type selection method even under the lowest MRT thresholds, as shown in the evaluation section). For the benchmarks that we ran, only javac ran against this limit.

Note that some object types are selected when building the TVA-aware VM. This is the so called *bootlist* TVA-enabled object types. In other words, all object types from this bootlist are TVA-enabled for all applications running on this VM. The bootlist TVA-enabled object types were chosen as the intersection of object types as selected through the offline STVA type selection method from the previous subsection. In other words, the bootlist TVA-enabled object types comprise object types that are frequently allocated across various applications as well as the VM.

6. 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.

6.1 Jikes RVM

The Jikes RVM is an open-source virtual machine developed by IBM Research [Alpern et al. 2000]. 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 GenCopy and GenMS garbage collectors. GenCopy is a generational collector that employs a SemiSpace copying strategy during full heap collections. GenMS is also a generational collector but uses a mark-sweep strategy during full heap collection. Note that Jikes RVM is a rather unique VM since it is written in Java. This affects the results presented in this paper because the reduced header Java object models also apply to Jikes RVM objects; this will be quantified in the evaluation section.

suite	benchmark	input
SPECjvm98	jess	-s100
	db	-s100
	javac	-s100
	mpegaudio	-s100
	mtrt	-s100
	jack	-s100
SPECjbb2000	pseudojbb	up to 8 warehouses
		up to 15 warehouses
Java Grande Forum	search	large
	moldyn	large
	crypt	large
	FFT	large
	heapSort	large
	LUFact	large
	SOR	large
	sparse	large

Table I. The benchmarks used in this paper.

6.2 Benchmarks

The benchmarks that we use in this study come from three different sources. We use SPECjvm98, SPECjbb2000 and the Java Grande Forum benchmarks. They are summarized in Table I. The SPECjvm98 benchmarks model client-side workloads. We use the -s100 input for all the benchmarks when reporting results; our profiling runs use the -s10 input. 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 used pseudojbb which runs for a fixed amount of work (35,000 transactions per warehouse). During our profiling runs, pseudojbb processes 12,000 transactions per warehouse. We use two inputs to pseudojbb in our evaluation; in our first input set, we run pseudojbb from 1 up to 8 warehouses, and in our second input set, we run from 1 up to 15 warehouses. The reason for employing the second input set is to increase the memory footprint. The Java Grande Forum (JGF) benchmark suite includes a set of sequential computational science and engineering codes, as well as business and financial models. These benchmarks typically work on large arrays and execute significant amounts of floating-point code. We use the largest input available when reporting final results; profiling was done using a smaller input. For the SPECjvm98 benchmarks, we set the maximum heap size to 200MB; for JGF, the maximum heap size is 384MB; for SPECjbb2000 the heap size is set to 384MB for running up to 8 warehouses and 700MB for running up to 15 warehouses.

6.3 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 in size. Our machine contains 1GB of main memory.

In the evaluation section we will measure execution times on the IBM POWER4 using hardware performance monitors. 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.

6.4 Statistical analysis

In the evaluation section, we want to measure the impact on performance of the reduced header Java object models. 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 15 measurement runs. These statistics will help us in determining whether the reduced header object models result in statistically significant or statistically insignificant performance gains or degradations. We use the unpaired or noncorresponding setup for comparing means, see [Lilja 2000] (pages 64–69).

7. EVALUATION

We now evaluate the reduced header Java object models using the experimental setup detailed in the previous section.

7.1 Feasibility study of STVA

We first inspect the potential of Selective Typed Virtual Addressing by characterizing the profile input. As mentioned in section 5, we determine whether an object type is TVA-enabled or TVA-disabled based on two criteria for offline STVA type selection. First, a type needs to be allocated frequently, *i.e.*, the potential reduction in memory consumption for the given type needs to exceed the memory reduction threshold (MRT). Or, second, the potential reduction in memory consumption for the given type in case it is a long-lived type needs to exceed the long-lived memory reduction threshold (LLMRT). We now study the sensitivity of the number of selected object types and potential memory consumption reduction to the chosen MRT and LLMRT thresholds. This is shown in Figure 5 by varying MRT from 0.05% up to 1% and by varying LLMRT over three values 0.1%, 0.5% and infinite. Note that the data in Figure 5 is for the profile input, and only gives a rough indication of what is to be expected for the reference input. In addition, Figure 5 only contains data concerning the nursery and mature generations. The data allocated in the Large Object Space (LOS) — the LOS is the space in which all large objects get allocated — is removed from this graph for clarity; STVA is expected to give only a very marginal benefit for large objects.

The top graph in Figure 5 shows the number of selected object types. As expected, we observe that the number of selected types decreases with increasing MRT and LLMRT. For example, an MRT of 0.05% selects on average 54 types whereas an MRT of 0.2% selects on average 21.5 types. The number of selected types varies over the benchmarks; for example for a 0.2% MRT, the number of selected objects varies from 8 up to 31. Note that this is only a small fraction

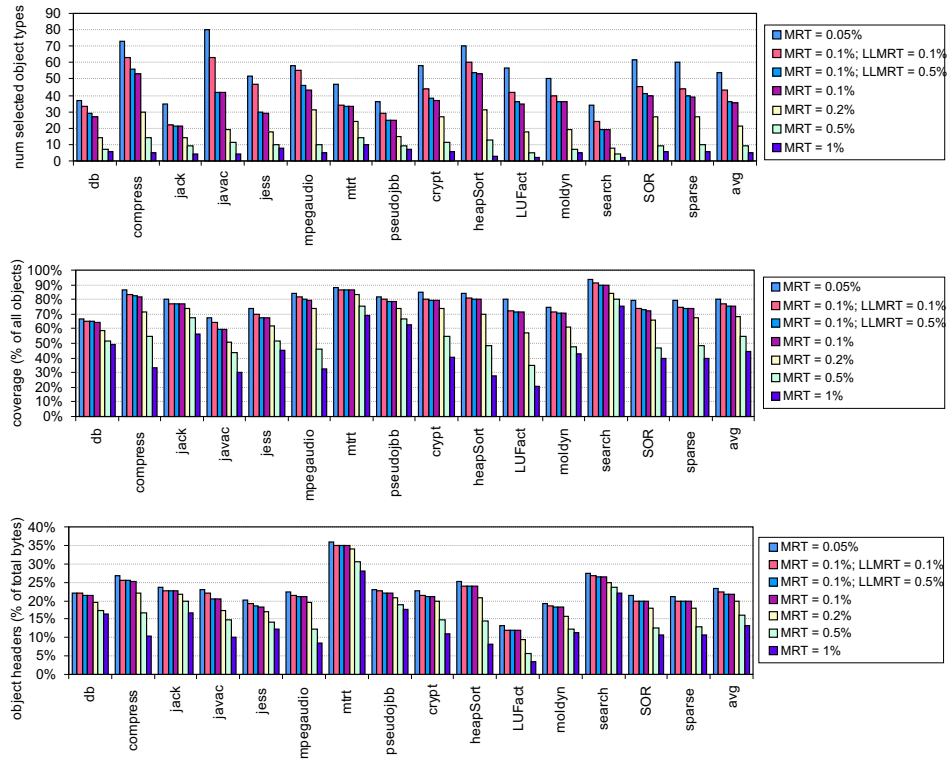

Fig. 5. The top graph shows the number of selected object types as a function of the MRT and LLMRT thresholds. The middle graph shows the coverage by the selected objects as a percentage of the total number of objects. The bottom graph shows the number of allocated bytes in the headers of the selected object types as a percentage of the total number of allocated bytes.

of the total number of object types. The total number of types allocated at least once ranges from 450 to 650 over the various benchmarks. The middle graph in Figure 5 shows the coverage by the selected object types, *i.e.*, the fraction of the total number of allocated objects that is accounted for by the selected object types. We observe that selecting only a small number of types results in a fairly large coverage. A 0.05% MRT yields an average coverage of 80.3%; a 0.2% MRT yields an average coverage of 68.4%. The bottom graph in Figure 5 shows the percentage of the total number of allocated bytes due to headers of the selected object types. This percentage shows the potential reduction in memory consumption in case the complete header would be removed from the selected objects for the profile input. For example, a 0.05% MRT potentially yields an average 23.1% potential reduction in allocated bytes, with a peak for mtrt of 36%. A 0.2% MRT yields an average potential reduction of 20%.

7.2 Memory consumption

Figures 6 and 7 show the reduction in allocated bytes for the offline and online header reduction techniques, respectively. Again, for the offline technique, these

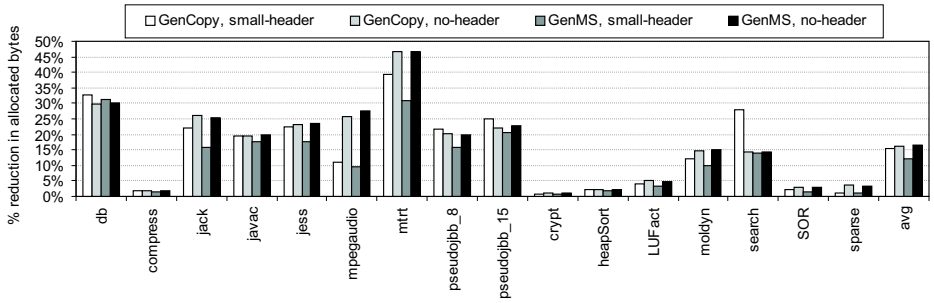

Fig. 6. Reduction in the number of allocated bytes for the offline header reduction techniques with MRT and LLMRT set to 0.1%.

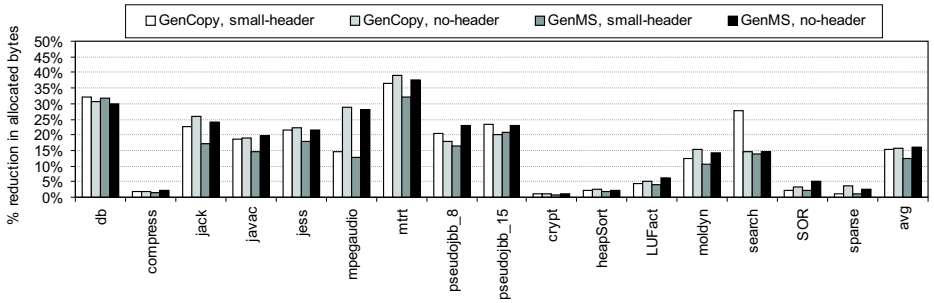

Fig. 7. Reduction in the number of allocated bytes for the online header reduction techniques.

numbers are for the reference input from a cross-validation setup. We observe an average reduction in allocated bytes of 15%. For some benchmarks we even observe a reduction in allocated bytes of 28% (search), 29% (mpegaudio), 32% (db) and 46% (mtrt). There are a number of important notes that we would like to make:

—Our first note relates to the data presented in Figure 6 compared to the data presented in Figure 5 for the feasibility study. The data in Figure 6 is for the reference runs whereas Figure 5 is for the profile input. Note that for some benchmarks such as db and mtrt we obtain larger reductions in memory consumption with the reference input than what we expected from the profile input, compare Figure 6 against Figure 5. This is explained by the fact that the reference input spends more time in the application than the profile input does. And since the VM objects tend to be larger than application objects, it is to be understood that the reduction in memory consumption is larger for the reference input than for the profile input.

—A second note we would like to make is that some benchmarks, such as compress and some of the JGF benchmarks, have a fairly low reduction in allocated bytes. The reason is that these benchmarks allocate long arrays — reducing the header size thus has a limited effect on the overall memory reduction. In addition, the data in Figure 5 shows potential memory reductions in the nursery and mature generations only, no data is included concerning the large object space (LOS).

benchmark	offline	online	in common
db	35	38	35
compress	34	32	32
jack	37	43	36
javac	56	80	48
jess	41	44	38
mpegaudio	36	36	32
mtrt	47	48	46
pseudojbb_8	44	57	39
pseudojbb_15	44	64	39
crypt	33	34	32
heapSort	34	33	32
LUFact	34	32	32
moldyn	35	33	32
search	34	33	32
SOR	33	32	32
sparse	33	33	32

Table II. Number of TVA-enabled object types for offline STVA type selection, online STVA type selection and the number of object types in common between offline and online type selection. This includes 32 bootlist TVA-enabled object types.

The data in Figures 6 and 7 show the effective memory reduction.

—A third note is that for some benchmarks, the no-header object model allocates more bytes than the small-header object model. There are two reasons for this. First, in case of a copying collector, the small-header object model is applied to arrays of all lengths whereas the no-header object model is only applied to arrays of a single length as discussed in section 4.2. Some benchmarks suffer from the fact that TVA cannot be applied to all array sizes. Second, when a TVA-enabled object, on which a hashcode is taken, is moved in the no-header object model, the object is TVA-disabled which causes the object to grow in size.

—Finally, the reduction in allocated bytes is comparable between the offline and online header reduction techniques, in spite of the different approaches taken for selecting TVA-enabled object types, as discussed in section 5. The reason is that the offline and online header reduction techniques have various selected TVA-enabled object types in common. This is quantified in Table II where the number of TVA-enabled types are shown for offline and online type selection as well as the number of object types in common between offline and online type selection.

7.2.1 *Reduction through TIB pointer compression versus STVA.* Figure 8 shows the reduction in allocated bytes partitioned by (i) the TIB pointer compression technique and (ii) the no-header STVA object model. We observe that approximately half the reduction in memory consumption comes from TIB pointer compression that is applied to all objects; the other half comes from the no-header STVA object model that can be applied only to TVA-enabled objects.

7.2.2 *Reduction in application objects versus VM objects.* As mentioned in the experimental setup in section 6, Jikes RVM is a VM that is written in Java. As a consequence, STVA also applies to objects allocated by the VM and thus, the results presented in this paper account for applying STVA to both VM objects and application objects. Other VMs that are not written in Java on the other hand, may not get similar benefits from STVA as what is presented in this paper. In order

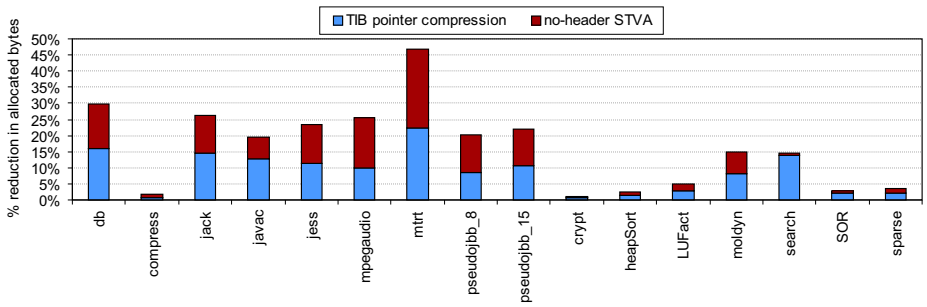

Fig. 8. The reduction in allocated bytes partitioned by TIB pointer compression and no-header STVA for the GenCopy collector.

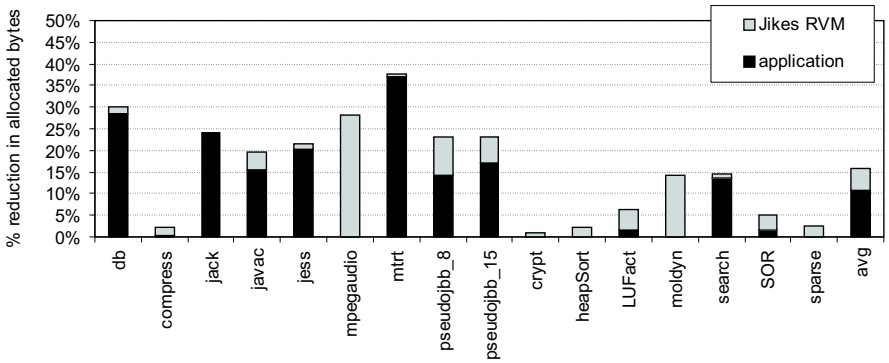

Fig. 9. Accounting the overall memory reduction to application and VM objects; this graph assumes the GenMS garbage collector and the no-header STVA object model.

to quantify the impact of our experimental setup using Jikes RVM, we classify the objects as VM and application objects and then compute the amount of memory reduction for VM and application objects separately. Classifying objects as VM and application objects is done by scanning the stack upon allocating an object until a method is reached that is either an application method or a VM method.

Figure 9 quantifies the amount of memory reduction for the application objects and VM objects. The important observation here is that the most significant part of the overall 16% memory reduction is obtained through the application objects (11% on average); about 5% is accounted for by VM objects. These results show that other VMs that are not written in Java can also benefit significantly from implementing STVA for reducing overall memory consumption.

7.2.3 *Reduction in in-use memory pages.* Figures 10, 11 and 12 show the heap size counted as the number of pages in use on the vertical axis as a function the number of allocations on the horizontal axis for jack, javac and pseudojbb with up to 15 warehouses, respectively. The curves in these graphs increase as memory gets allocated until a garbage collection is triggered after which the number of used pages drops to the amount of reachable data at that point. This explains the shape of

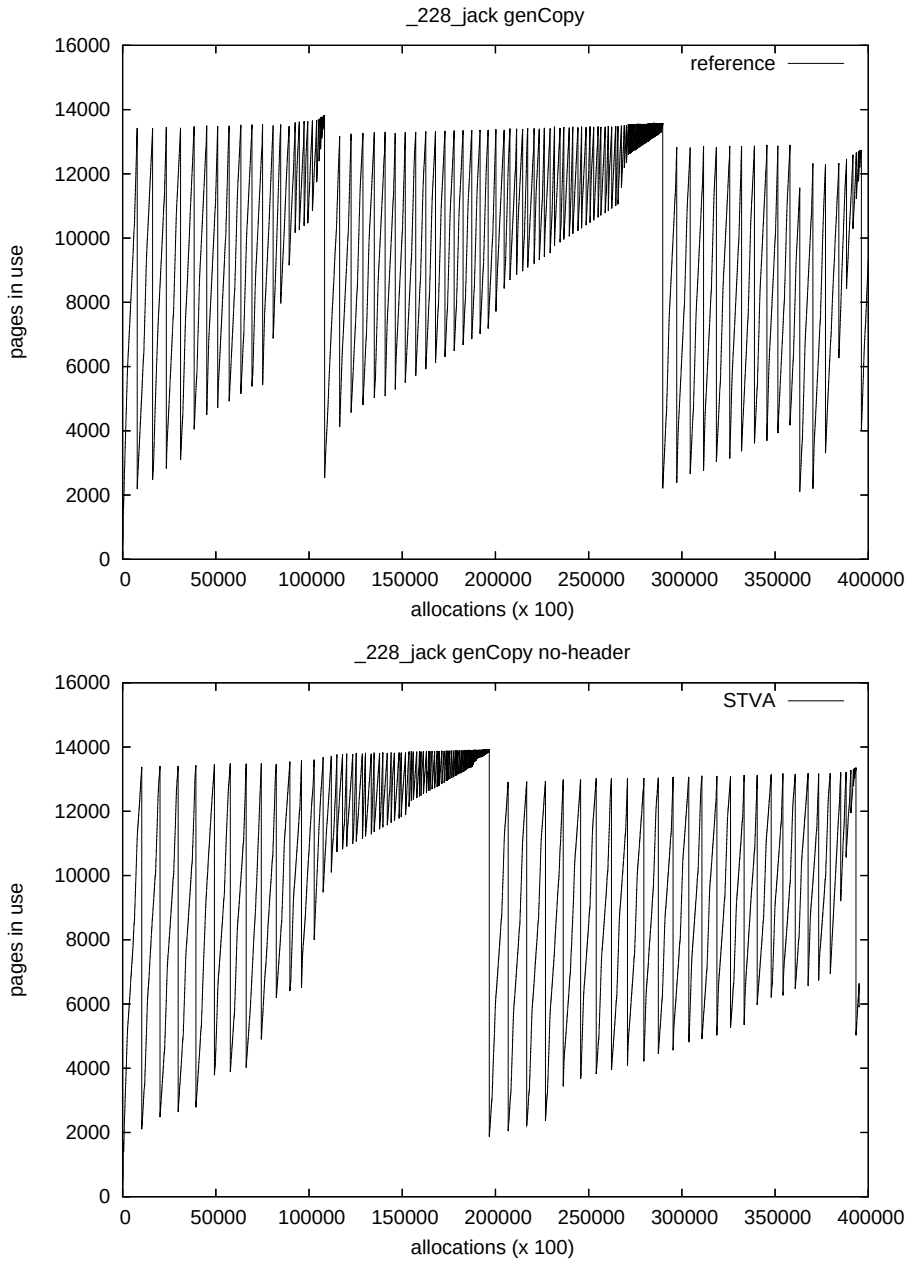

Fig. 10. The heap size is shown as a function of the number of allocations for the original Jikes RVM implementation (top graph) versus STVA (bottom graph) for jack.

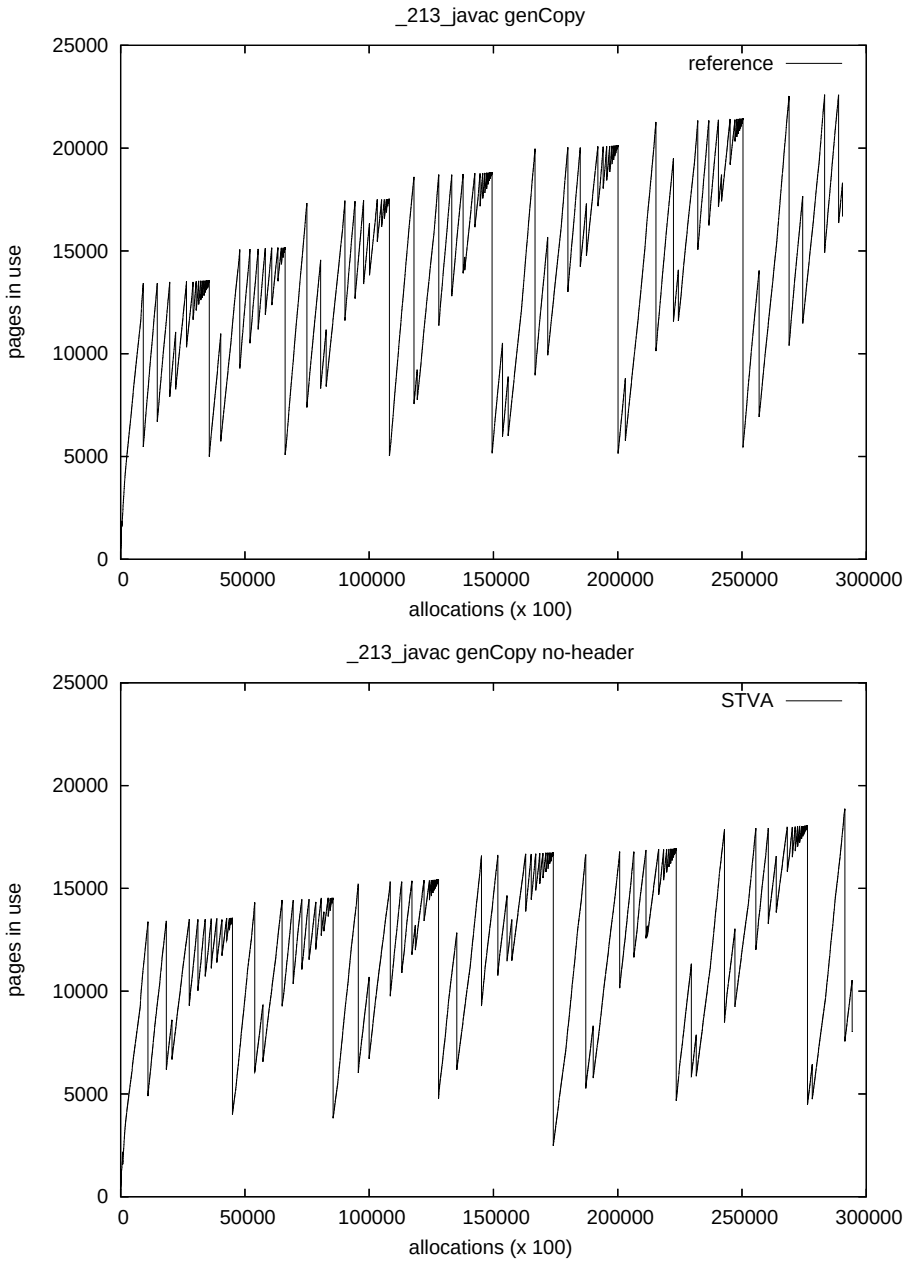

Fig. 11. The heap size is shown as a function of the number of allocations for the original Jikes RVM implementation (top graph) versus STVA (bottom graph) for javac.

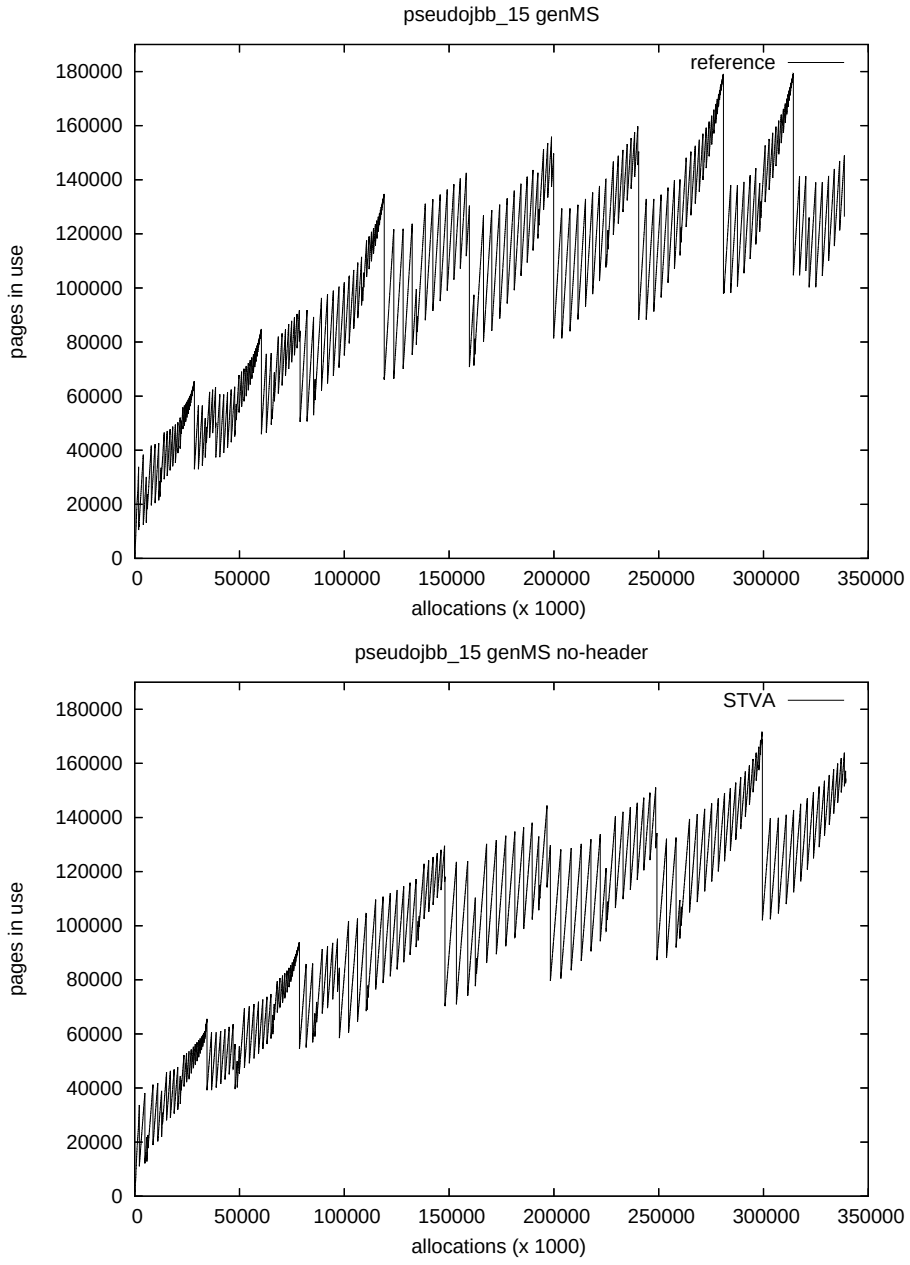

Fig. 12. The heap size is shown as a function of the number of allocations for the original Jikes RVM implementation (top graph) versus STVA (bottom graph) for pseudojbb with up to 15 warehouses.

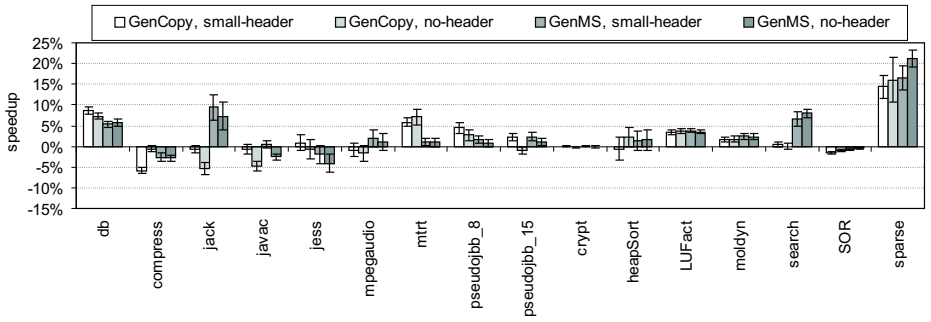

Fig. 13. Speedups along with the 95% confidence intervals for offline header reduction. The MRT and LLMRT thresholds are set to 0.1%.

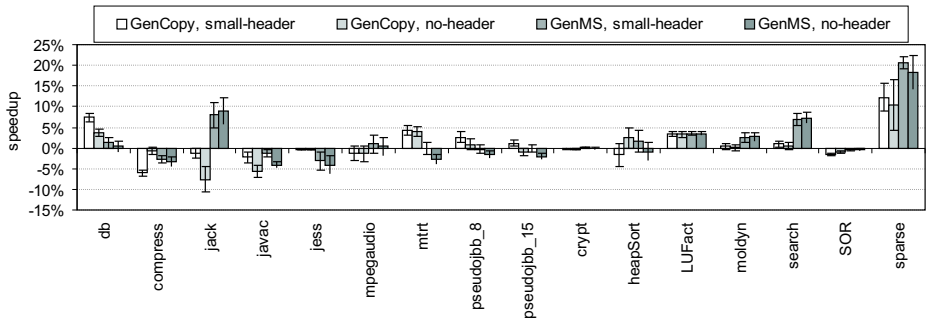

Fig. 14. Speedups along with the 95% confidence intervals for online header reduction.

these graphs. There are two important observations to be made from these graphs. First, since STVA reduces the amount of allocated bytes per allocation, garbage collections get delayed — the STVA curve is shifted to the right compared the original Jikes RVM curve. In other words, fewer garbage collections are required. Second, when garbage is collected, the number of pages in use for STVA can drop below the number of pages in use for the original Jikes RVM. The reason is that the amount of reachable bytes is smaller under STVA because of the space-efficient STVA object model.

7.3 Performance

Figures 13 and 14 show the speedup for the offline and online reduced header object models, respectively. Data is shown for the small-header and no-header object models as well as for the GenCopy and the GenMS collectors. These figures show speedups along with the 95% confidence intervals. The offline reduced header object models are obtained from a cross-validation setup, *i.e.*, we use profile inputs for selecting the TVA-enabled types, and we use reference inputs for reporting speedups. We set MRT and LLMRT to 0.1% in these experiments based on our previous work [Venstermans et al. 2006b].

We observe that for some benchmarks, STVA results in a statistically significant

performance degradation. This is the case for compress and SOR for all collectors and object models. For a number of benchmarks, we observe performance degradations for only a few collector and object model configurations. The performance degradation that we see is generally smaller than 5%, with one exception of 7% for jack for the online no-header object model. This suggests that the run-time overhead introduced by STVA has a larger impact on overall performance than the reduction in memory footprint for these benchmarks. A number of benchmarks show a significant performance improvement: db (7%), mtrt (5%), LUFact (4%), moldyn (3%), sparse (up to 20%) and jack for the GenMS collector (up to 10%). For these benchmarks, the reduction in memory consumption has a larger impact on overall performance than the increased run-time overhead due to STVA which results in a significant speedup. For all remaining benchmarks, STVA has no statistically significant impact on overall performance. In conclusion, the space-efficient object models do not have a negative impact on performance for most of the benchmarks and a couple of benchmarks even show a significant speedup.

7.4 Cache and TLB performance

We now study the impact of STVA on cache and TLB performance in more detail using hardware performance counters. Figure 15 quantifies the number of D-TLB, L1, L2 and L3 misses per thousand instructions in the reference run for the GenMS garbage collector; we obtained similar results for the GenCopy garbage collector. We observe that the number of D-TLB misses does not increase for most benchmarks. However, for a few benchmarks the number of D-TLB misses slightly increases due to the increased memory fragmentation because of STVA. The number of cache misses typically decreases, especially for the larger L2 and L3 caches; the reason is the reduced memory consumption. For some benchmarks such as db and sparse, the number of L3 misses decreases by 50%. This large decrease in L3 misses explains the speedup results reported in Figures 13 and 14. Note that the reduction in L3 cache miss rate for sparse is not due to a reduction in the amount of memory consumed by application objects, see Figure 9. Instead, the reduction in L3 cache miss rate primarily comes from a reduction in the amount of memory consumed by VM objects along with a changed data layout through STVA.

7.5 STVA versus TVA

As mentioned throughout the paper, one contribution of this paper is to show that applying TVA to a selected number of object types (*i.e.*, STVA) results in better performance than applying TVA to all object types. This is clearly shown in Figure 16 where STVA is compared against TVA. TVA performs fairly well in general and achieves similar performance as STVA for many benchmarks. However, for a number of benchmarks, TVA results in significant performance degradations compared to STVA. This is the case for compress, javac, mpegaudio, pseudojbb, LUFact and sparse. The compress benchmark even results in a 15% overall performance degradation; pseudojbb with 15 warehouses experiences an 8.4% performance degradation under TVA. As such, we conclude that implicit typing on selected object types outperforms implicit typing on all object types.

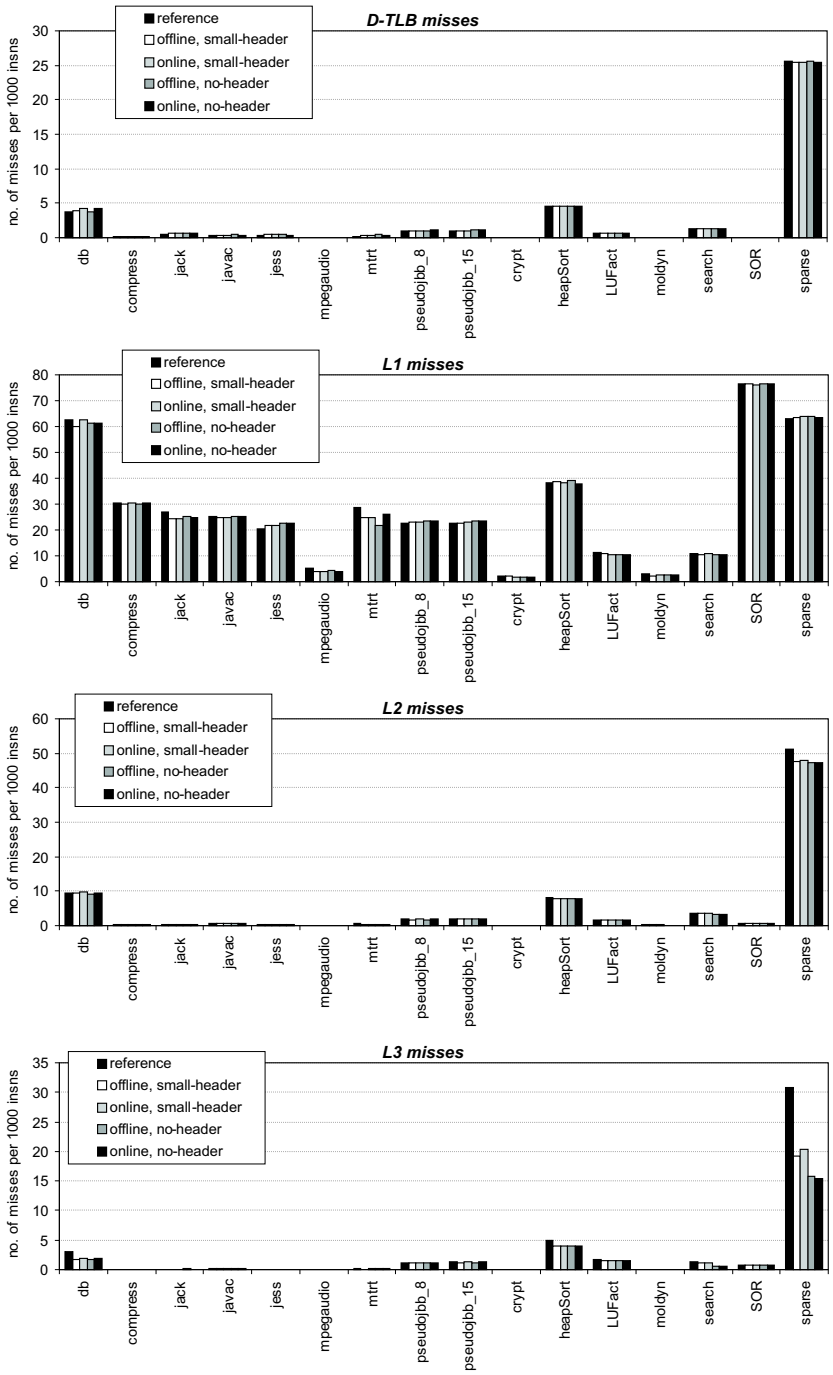

Fig. 15. Evaluating the performance of the cache hierarchy under STVA: D-TLB, L1, L2 and L3 misses per thousand instructions in the reference run for the GenMS garbage collector.

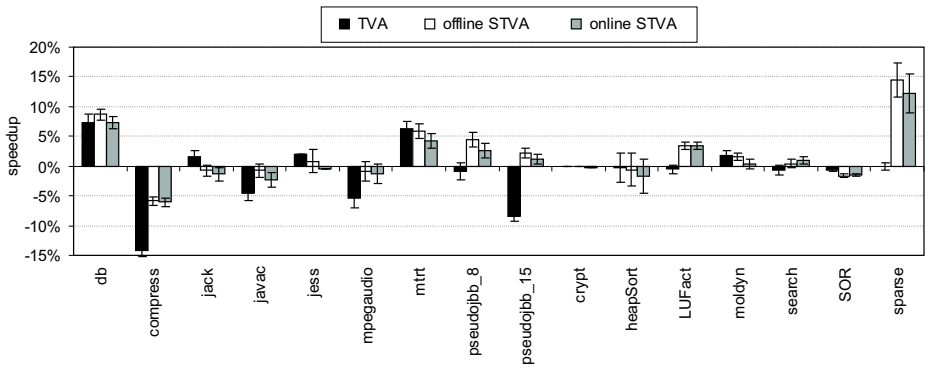

Fig. 16. Comparing STVA versus TVA in terms of speedup for the GenCopy garbage collector.

8. RELATED WORK

Dieckmann and Hölzle [1999] 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. [2006a] 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 three reasons for this. First, a reference in 64-bit computing mode is twice the size as in 32-bit computing mode. Second, the header in 64-bit mode is also twice as large as in 32-bit mode. And third, alignment issues also increase the object size in 64-bit mode. They conclude that for non-array objects, the increased header accounts for half the object size increase. In this paper, we propose STVA as a way to eliminate the object header in 64-bit VMs.

Adl-Tabatabai et al. [2004] 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 TIB pointer and the forwarding pointer in the object header and to pointers in the object itself. By compressing the TIB pointer and the forwarding pointer in the object header, they can actually reduce the size of the object header from 16 bytes (for non-array objects) to only 8 bytes. There are three key differences with our approach. First, we eliminate the TIB pointer completely from the object header for TVA-enabled objects; [Adl-Tabatabai et al. 2004] only compresses the TIB pointer. The second difference between Adl-Tabatabai et al.'s approach and our proposal is that we do not need to compress and decompress the TIB pointer. We compute the TIB pointer from the object's virtual address. And finally, the approach by Adl-Tabatabai et al. limits applications to a 32-bit address space. As such, applications that require more than 4GB of memory cannot benefit from pointer compression. STVA and TIB pointer compression do not suffer from this limitation. The only assumption we make in our proposal is that we do not need more than a 32-bit virtual address space for holding type information, however, it is highly unlikely that this would

ever be needed in practice.

Bacon et al. [2002] 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 uses the least significant bits from a memory address (which are typically zero) for other uses. The main disadvantage of bit stealing is that it frees only a few bits. Indirection represents the TIB pointer as an index into a table of TIB pointers. The disadvantages of indirection are that an extra load is needed to access the TIB pointer, and that there is a fixed limit on the number of TIBs and thus the number of object types that can be supported. The approach that we propose has the advantage over these two approaches to completely eliminate the TIB pointer from the object header. The bit stealing and indirection methods on the other hand still require a condensed form of a TIB pointer to be stored in the header.

The third header compression method discussed by Bacon et al. [2002] is called the implicit type method. The general idea behind implicit types is that the type information is part of the object's virtual address. In fact, there are number of ways of how to implement implicit typing. A first possibility is to have a type tag included in the pointer to the object. The type tag is then typically stored in the most-significant or least-significant bits of the object's virtual address. By consequence, obtaining the effective memory address requires masking the object's virtual address. Storing the type tag in the most-significant bits of the object's virtual address usually restricts the available address space. Storing the type tag in the least-significant bits of the object's virtual address on the other hand, usually forces objects to be aligned on multiple byte boundaries. A second approach is to use the type tag bits as a part of the address. By doing so, the address space gets divided into several distinct regions where objects of the same type get allocated into the same region. This is similar to the TVA implementation that we use in this paper.

A third approach is the Big Bag of Pages (BiBOP) approach proposed by Steele, Jr. [1997] and Hanson [1980]. In BiBOP, the type tag serves as an index into a table where the type is stored. BiBOP views memory as a group of equal-sized segments. Each segment has an associated type. An important disadvantage of BiBOP typing is that the type tag that is encoded in the memory address serves as an index in a table that points to the object's TIB. In other words, an additional indirection is needed for accessing the TIB. Dybvig et al. [1994] propose a hybrid system where some objects have a type tag in the least-significant bits and where other objects follow the BiBOP typing. The typed virtual memory addressing that we propose here in this paper differs from this prior work on typed virtual addressing in the following major ways. First, we propose to apply implicit typing to selected object types only; previous work applied implicit typing to all object types. Applying implicit typing to all object types results in significant memory fragmentation. We argue and show how to make a good selection on what objects to apply the implicit type method. Second, although previous work describes the implicit type method, they do not evaluate it and do not compare it against memory systems without typed virtual addressing. In this paper, we propose a practical method of how to

implement the implicit typing method for 64-bit Java VM implementations. In addition, we quantify the performance and memory consumption impact of STVA and compare that against traditional VM implementations without STVA.

Shuf et al. [2002] 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. Our STVA implementation does not have these disadvantages. 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. The STVA implementation that we propose could be viewed of as a hybrid form of the prolific approach and the implicit typed methods discussed above; we apply implicit typing to prolific types.

A number of related research studies have been done on characterizing the memory behavior of Java applications, such as [Blackburn et al. 2004; Kim and Hsu 2000; Shuf et al. 2001]. Other studies aimed at reducing the memory consumption of Java applications, for example, using techniques such as heap compression [Chen et al. 2003], object compression [Chen et al. 2005], pointer compression [Lattner and Adve 2005], *etc.*

9. CONCLUSION

This paper proposed eliminating the header from the 64-bit Java object model through Selective Typed Virtual Addressing (STVA). The idea of STVA is to apply typed virtual addressing (TVA) or implicit typing to a selected number of object types. TVA means that the object type is encoded in the object's virtual address. We apply TVA selectively, hence the name Selective TVA, on object types that are frequently allocated. The end result is that the header can be eliminated completely from the object header. The TIB pointers are stored in the TIB space and the status field information is stored in side arrays. Accessing the appropriate TIB pointer and status field is done through offsets computed from the object's virtual address. For the objects on which we do not apply TVA, we compress the TIB pointer from 64-bit to 32-bit.

We evaluated our newly proposed space-efficient Java object model in a 64-bit Java VM implementation, namely Jikes RVM, on an AIX IBM POWER4 machine. Our results show that the space-efficient object model yields a reduction in the number of allocated bytes by 15% on average (up to 45%). Half the reduction comes

from STVA; the other half comes from TIB pointer compression. In terms of performance, the space-efficient Java object model generally does not affect performance in a statistically significant way, however, some benchmarks exhibit significant performance speedups (up to 20%).

Acknowledgements

We would like to thank the anonymous reviewers for their valuable and constructive feedback; we thank both the reviewers of our CGO-2006 paper [Venstermans et al. 2006b], as well as the reviewers of this journal extension. These review comments greatly helped us improve the work reported in this journal paper. 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).

REFERENCES

Adl-Tabatabai, A.-R., Bharadwaj, J., Cierniak, M., Eng, M., Fang, J., Lewis, B. T., Murphy, B. R., and Stichnoth, J. M. 2004. Improving 64-bit Java IPF performance by compressing heap references. In *Proceedings of the Second Annual International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, 100–110.

Agesen, O. 1999. Space and time-efficient hashing of garbage-collected objects. *Theory and Practice of Object Systems 5,* 2, 119–124.

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., and Whaley, J. 2000. The Jalapeno Virtual Machine. *IBM Systems Journal 39,* 1 (Feb.), 211–238.

Appel, A. W. 1989. A runtime system. Tech. Rep. CS-TR-220-89, Princeton University, Computer Science Department, May.

Bacon, D. F., Fink, S. J., and Grove, D. 2002. Space- and time-efficient implementation of the Java object model. In *Proceedings of the Sixteenth European Conference on Object-Oriented Programming (ECOOP)*. Springer, 111–132.

Bacon, D. F., Konuru, R., Murthy, C., and Serrano, M. 1998. Thin locks: Featherweight synchronization for java. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 258–268.

Blackburn, S. M., Cheng, P., and McKinley, K. S. 2004. Myths and realities: the performance impact of garbage collection. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. ACM Press, 25–36.

Chen, G., Kandemir, M., and Irwin, M. J. 2005. 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)*. ACM Press, 68–78.

Chen, G., Kandemir, M., Vijaykrishnan, N., Irwin, M. J., Mathiske, B., and Wolczko, M. 2003. Heap compression for memory-constrained Java environments. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*. ACM Press, 282–301.

Dieckmann, S. and Hölzle, U. 1999. A study of the allocation behavior of the SPECjvm98 Java benchmarks. In *Proceedings of the 13th European Conference for Object-Oriented Programming (ECOOP)*. Springer, 92–115.

Dybvig, R. K., Eby, D., and Bruggeman, C. 1994. Don't stop the BIBOP: Flexible and efficient storage management for dynamically-typed languages. Tech. Rep. 400, Indiana University Computer Science Department. Mar.

Hanson, D. R. 1980. A portable storage management system for the Icon programming language. *Software—Practice and Experience 10,* 6, 489–500.

Kim, J.-S. and Hsu, Y. 2000. Memory system behavior of Java programs: methodology and analysis. In *Proceedings of the 2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. ACM Press, 264–274.

Lattner, C. and Adve, V. 2005. 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)*. ACM Press, 129–142.

Lilja, D. J. 2000. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press.

Shebs, S. T. and Kessler, R. R. 1987. Automatic design and implementation of language data types. In *Proceedings of the ACM SIGPLAN 1987 Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 26–37.

Shuf, Y., Gupta, M., Bordawekar, R., and Singh, J. P. 2002. Exploiting prolific types for memory management and optimizations. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM Press, 295–306.

Shuf, Y., Serrano, M. J., Gupta, M., and Singh, J. P. 2001. Characterizing the memory behavior of Java workloads: a structured view and opportunities for optimizations. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. ACM Press, 194–205.

Steele, Jr., G. L. 1997. Data representation in PDP-10 MACLISP. Tech. Rep. AI Memo 420, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Sept.

Venstermans, K., Eeckhout, L., and De Bosschere, K. 2006a. 64-bit versus 32-bit virtual machines for java. *Software—Practice and Experience 36,* 1 (Jan.), 1–26.

Venstermans, K., Eeckhout, L., and De Bosschere, K. 2006b. 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)*. IEEE Computer Society, 76–86.