Space-Efficient 64-bit Java Objects through Selective Typed Virtual Addressing

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

Abstract

Memory performance is an important design issue for contemporary systems given the ever increasing memory gap. This paper proposes a space-efficient Java object model for reducing the memory consumption of 64-bit Java virtual machines. We propose Selective Typed Virtual Addressing (STVA) which uses typed virtual addressing (TVA) or implicit typing for reducing the header of 64-bit Java objects. The idea behind TVA is to encode the object's type in the object's virtual address. In other words, all objects of a given type are allocated in a contiguous memory segment. As such, the type information can be removed from the object's header which reduces the number of allocated bytes per object. Whenever type information is needed for the given object, masking is applied to the object's virtual address. Unlike previous work on implicit typing, we apply TVA to a selected number of frequently allocated and/or long-lived object types. This limits the amount of memory fragmentation. We implement STVA in the 64-bit version of the Jikes RVM on an AIX IBM platform and compare its performance against a traditional VM implementation without STVA using a multitude of Java benchmarks. We conclude that STVA reduces memory consumption by on average 15.5% (and up to 39% for some benchmarks). In terms of performance, STVA generally does not affect performance, however for some benchmarks we observe statistically significant performance speedups, up to 24%.

1 Introduction

Java is very popular on various computing systems, ranging from embedded devices to high-end servers. There are several reasons for the popularity of Java: its platform-independence as it relies on virtual machine (VM) technology, its object-oriented programming paradigm, its reliance on automatic memory management, etc. All of these factors contribute to the productivity enhancement of software development using Java.

Another well known phenomenon today is that the memory gap, *i.e.*, the gap between processor and memory speed continues to grow. As such, it is important to study techniques both in hardware and in software that help addressing the memory gap. On the hardware side, a multitude of techniques have been proposed to tackle the memory gap. Some examples are memory hierarchies, non-blocking caches, hardware prefetching, load address prediction, etc. On the software side, the memory gap can be tackled by reducing the amount of memory being consumed by applications, or by improving the data locality of an application, or by employing latency hiding techniques such as software prefetching, etc.

This paper focuses on reducing the memory consumption of 64-bit Java VM implementations. Our approach to reducing the memory consumption of 64-bit Java implementations is by proposing a Typed Virtual Addressing (TVA) mechanism that results in a space-efficient 64-bit Java object model. The TVA-aware VM that we propose removes the Type Information Block (TIB) pointer field from the object's header. Along with a number of other header layout modifications (that intelligently position the forwarding pointer in the object's header, as will be detailed in the paper) we reduce the object header size from 16 bytes to only 4 bytes for non-array objects, and from 24 bytes to only 8 bytes for array objects. The technology that enables removing the TIB pointer field is Typed Virtual Addressing which means that the object type is encoded in the object's virtual address. This is done by mapping all objects of the same type to the same contiguous memory segment. 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. Our proposal does not apply TVA to all object types but only to a selected number of types that are frequently allocated and tend to be long lived, 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 [3, 7, 8, 10, 12]. In fact, it was fairly popular in the 1970s, 1980s and early 1990s in languages such as Lisp, Scheme, Smalltalk, Prolog, ML, etc. 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 for a number of reasons. The virtual address space on a 64-bit computing system is huge which facilitates the implementation of implicit typing compared to 32-bit platforms. In addition, given the wide pointers on 64-bit platforms, reducing the amount of memory space consumed because of pointers becomes an important design issue.

We implement our STVA 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 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 STVA implementation reduces memory consumption by on average 15.5% (up to 39% for some benchmarks). In terms of performance, STVA 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%). For a number of other benchmarks we observe statistically significant performance speedups, up to 24%. In general though, we conclude that STVA substantially reduces memory consumption without significantly affecting performance; however, performance improvements are observed for some benchmarks. This paper is only a first step into a completely online Java 64-bit VM solution. Our current approach requires a profiling step for determining on what object types to apply STVA and extending our approach to an online VM technique is subject to future work.

This paper is organized as follows. Section 2 revisits the 64-bit Java object model as it is used in existing VM implementations. We subsequently discuss prior work in section 3 and position our work against this prior work. Section 4 then discusses our Selective Typed Virtual Addressing proposal in great detail. In section 5 we then detail our experimental setup and subsequently present the evaluation of our STVA approach in section 6. Finally, we conclude in section 7.

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's data fields along with a header. For clarity, we refer to an object as the object's data plus the object's header throughout the paper; the object's data refers to the data fields only. The object's 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's header typically contains the following information:

- 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's 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.

- 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 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 [5] 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 fourth element of the status field is the *forwarding pointer*. The forwarding pointer is used for keeping track of objects during generational or copying garbage collection. This forwarding pointer is 8 bytes in size. In practice, the status field is 8 bytes in size of which four bytes are used for the hash code, the lock element and the garbage collection bits. The forwarding pointer overwrites the hash code and lock element in the status field, but not the garbage collection bits. 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's 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.

Comparing the object model of a 64-bit VM versus the object model of a 32-bit VM, we conclude that the header is twice as large in the 64-bit implementation than in the 32-bit implementation. The TIB pointer doubles in size; the same is true for the forwarding pointer.

3 Prior work

Before presenting our method of reducing the object's header size in 64-bit Java implementations, we first discuss prior work in this area.

Venstermans *et al.* [13] 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, as discussed in the previous section. And third, alignment purposes also increase the object size in 64-bit mode as one wants to align objects on 8-byte boundaries. They conclude that for non-array objects, the increased header accounts for half the object size increase. In this paper, we propose STVA which is a way of reducing the object header in 64-bit VMs.

Adl-Tabatabai *et al.* [1] address the increased memory requirements of 64-bit Java implementations by compressing the 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's header which results in a 4 byte header, an effective memory consumption reduction of 12 bytes. 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 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. Our approach does not suffer from this limitation.

Bacon *et al.* [4] 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's 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.* [4] is called the implicit type method. The general idea behind implicit types is that the type information is part of the object's 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 address. By consequence, obtaining the effective memory address requires masking the object's address. Storing the type tag in the most-significant bits of the object's address usually restricts the available address space. Storing the type tag in the least-significant bits of the object's 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 [12] and Hanson [8]. In BiBOP, the type tag serves as an index into a table where the type is stored. BiBOP views memory as a group of equalsized 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.* [7] 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.* [11] 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's 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's 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.

4 Selective Typed Virtual Addresses for Java objects

This section explains the idea and the implementation details behind Selective Typed Virtual Addresses (STVA) for 64-bit Java objects. We first detail the TVA 64-bit Java object model. We subsequently discuss how we determine for which objects to implement TVA. And finally, we will go through a number of STVA implementation details.

4.1 The TVA Java object model

As mentioned in section 2, the header of a 64-bit Java object is 16 bytes in size and consists of an 8 byte TIB pointer field and an 8 byte status field. We reduce the header size for an object under TVA from 16 bytes to only 4 bytes. This is done in two steps. First, we remove the TIB pointer field. As will be discussed extensively later on this paper, the TIB pointer can then be calculated from the object's virtual address. This leaves us with a header consisting of an 8 byte status field. In a second step, we reduce this 8 byte status field to a 4 byte status field containing the lock, hash and GC elements. Whenever a forwarding pointer is needed, we overwrite the 4 byte status field as well as the first 4 bytes of the object's data. Note that the object's 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's data. This implies that the minimum size occupied by a TVA-enabled object is 8 bytes. (Obviously, this cannot be done for data structures belonging to the garbage collector itself—this is obviously only of concern whenever the garbage collector (or the entire VM) is written in Java.)

4.2 STVA type selection

As mentioned before, we do not apply TVA to all objects. Object types that are not allocated frequently 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 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.

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 preferably are 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 procentual volume of these header bytes in relation to the total number of allocated bytes. We then select object types for which this procentual 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 truely long-lived objects. For those objects that survive a garbage collection, we again compute the procentual volume of the header bytes in relation to the total number of allocated bytes. We then retain object types for which this procentual volume exceeds the *long-lived memory reduction threshold (LLMRT)*.

4.3 Implementation issues

We now detail on a number of issues of how to implement STVA. Although some of these issues are geared towards our specific 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.

The virtual memory address of a Java object in an STVA-enabled VM implementation is depicted in Figure 1. The five most significant bits are AIX-reserved bits that 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 where only TVA-enabled objects reside. 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

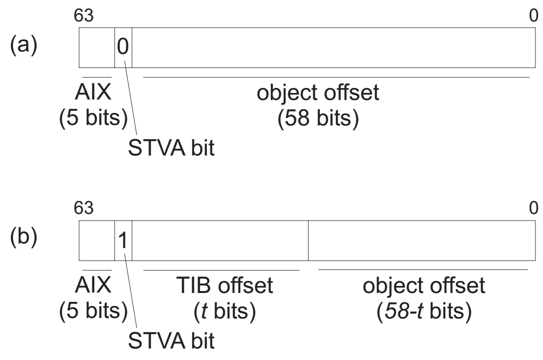

Figure 1. The 64-bit virtual address for a TVA-disabled object (a) and for a TVA-enabled object (b).

types. If the bit 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's type is a TVA-disabled type), the object falls under the default Java object model from section 2. In the latter case (a TVA-disabled object type), the least significant 58 bits determine the object's offset, see Figure 1(a). In case of a TVA-enabled object, see Figure 1(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.

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's 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 2. A TVA-disabled object follows the traditional way of getting to the TIB pointer. A load instruction reads the TIB pointer from the object's header. For a TVA-enabled object, the TIB pointer is computed from the object's vir-

```
if (bit 58 of virtual address is set) {
  /* TVA-enabled object*/
  mask the TIB offset from the object's
    virtual address;
  add the TIB offset to the base TIB pointer;
}
else {
  /* TVA-disabled object */
  read the TIB pointer from the object's header;
}
```

Figure 2. Computing an object's TIB pointer in an STVA-enabled VM implementation.

tual 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 that fall under TVA are mapped in a limited address space starting at the TIB base pointer. The TIB space's size is limited to 256MB in our implementation; this comes from the 25-bit TIB offset that we use, see Figure 1, along with a 3-bit shift left for 8-byte alignment. Note again that this limit is not hard and can be easily adjusted by changing the address organization from Figure 1 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, 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 conditional 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. 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.

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 live 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 3.

Jikes RVM however, works with contiguous spaces. Jikes RVM identifies a space by a SpaceDescriptor, a numerical value encrypting 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 by S, a mask by M, an address by A and the starting address of a space with B, 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 for example has the following bit pattern `'00..011..100..0'`, or `'11..100..0'`. A contiguous space now just is a special case in which the mask has a serie of `'1'`s followed by a serie of `'0'`s, *i.e.*, the mask looks as follows: `'11..100..0'`. A non-contiguous space has a mask of the form `'00..011..100..0'`, `'11..100..011.100..0'` or `'00..011..100..011..100..0'`.

It is also important to note that during garbage collection, different actions may be undertaken depending on whether an object resides in a nursery/mature TVA-disabled/enabled space. In our implementation, whenever it needs to be determined in what space an object resides, this is done through a nested-if construction. We optimized this nested-if construction in such a way that the most common case appears as the first if-statement. Nevertheless, this incurs additional overhead over the traditional garbage collector, since we have more if-branches.

It is also interesting to note 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

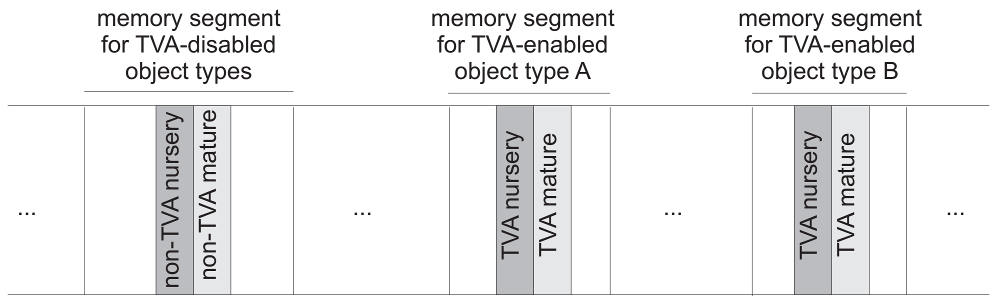
memory segment for TVA-disabled object types
memory segment for TVA-enabled object type A
memory segment for TVA-enabled object type B
...
non-TVA nursery
non-TVA mature
TVA nursery
TVA mature

Figure 3. Mapping the nursery and mature spaces in the virtual address space in a TVA-aware VM.

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.* [11] 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 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.

Jikes RVM. The Jikes RVM is an open-source virtual machine developed by IBM Research [2]. 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 garbage collector which is a generational collector that employs a SemiSpace copying strategy during full heap collections. Note that Jikes RVM is a rather unique VM since it is written in Java. This obviously affects the results presented in this paper because STVA also applies to Jikes RVM's objects. However, we believe that the memory consumption reduction would be even larger for other virtual machines that are not written in Java. The reason is that most Jikes RVM's objects are fairly large objects; the application's objects are generally smaller. As such, the relative decrease in memory consumption due to STVA would be even more significant for a VM not written in Java.

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 1. The SPECjvm98 benchmarks model client-side workloads. We use the `-s100` input

suite	benchmark
SPECjvm98	jess
	db
	javac
	mpegaudio
	mtrt
	jack
SPECjbb2000	pseudojbb
Java Grande Forum	search
	moldyn
	crypt
	FFT
	heapSort
	LUFact
	SOR
	sparse

Table 1. The benchmarks used in this paper.

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 a fixed amount of time, we used pseudojbb which runs for a fixed amount of work (35,000 transactions). During our profiling runs, pseudojbb processes 12,000 transactions. 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 SPECjbb2000 and JGF, the maximum heap size is 384MB.

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. Standard pages in the POWER4 are 4KB in size.

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.

Statistical analysis. In the evaluation section, we want to measure the impact on performance of an STVA-aware VM implementation. 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 STVA-aware VMs result in statistically significant or statistically insignificant performance gains or degradations. We use the unpaired or noncorresponding setup for comparing means, see [9] (pages 64–69).

6 Evaluation

We now evaluate our STVA-enhanced 64-bit Jikes RVM using the experimental setup detailed in the previous section.

6.1 Feasibility study

We first inspect the potential of Selective Typed Virtual Addressing by characterizing the profile input. As mentioned in section 4.2, we determine whether an object type is TVA-enabled or TVA-disabled based on two criteria. 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 4 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 4 is for the profile input, and only gives a rough indication of what is to be expected for the reference input. In addition, Figure 4 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 4 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 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 4 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 4 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; we do not remove the complete header though, but a large part of it. 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%.

6.2 Impact on performance

Figure 5 shows the performance speedup along with the 95% confidence intervals that we obtain through STVA. This is done as a function of the MRT and LLMRT thresholds. These results were 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 performance speedups. We observe that for some benchmarks, STVA results in a statistically significant performance degradation. This is the case for compress and SOR. However, the performance degradation is very small: 5% for compress and only 1% for SOR. This suggests that memory fragmentation due to 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%) and sparse (24%). For these benchmarks, the reduction in memory consumption has a larger impact on overall performance than the increased memory fragmentation which results in a significant performance speedup. For example, through a detailed analysis of the hardware

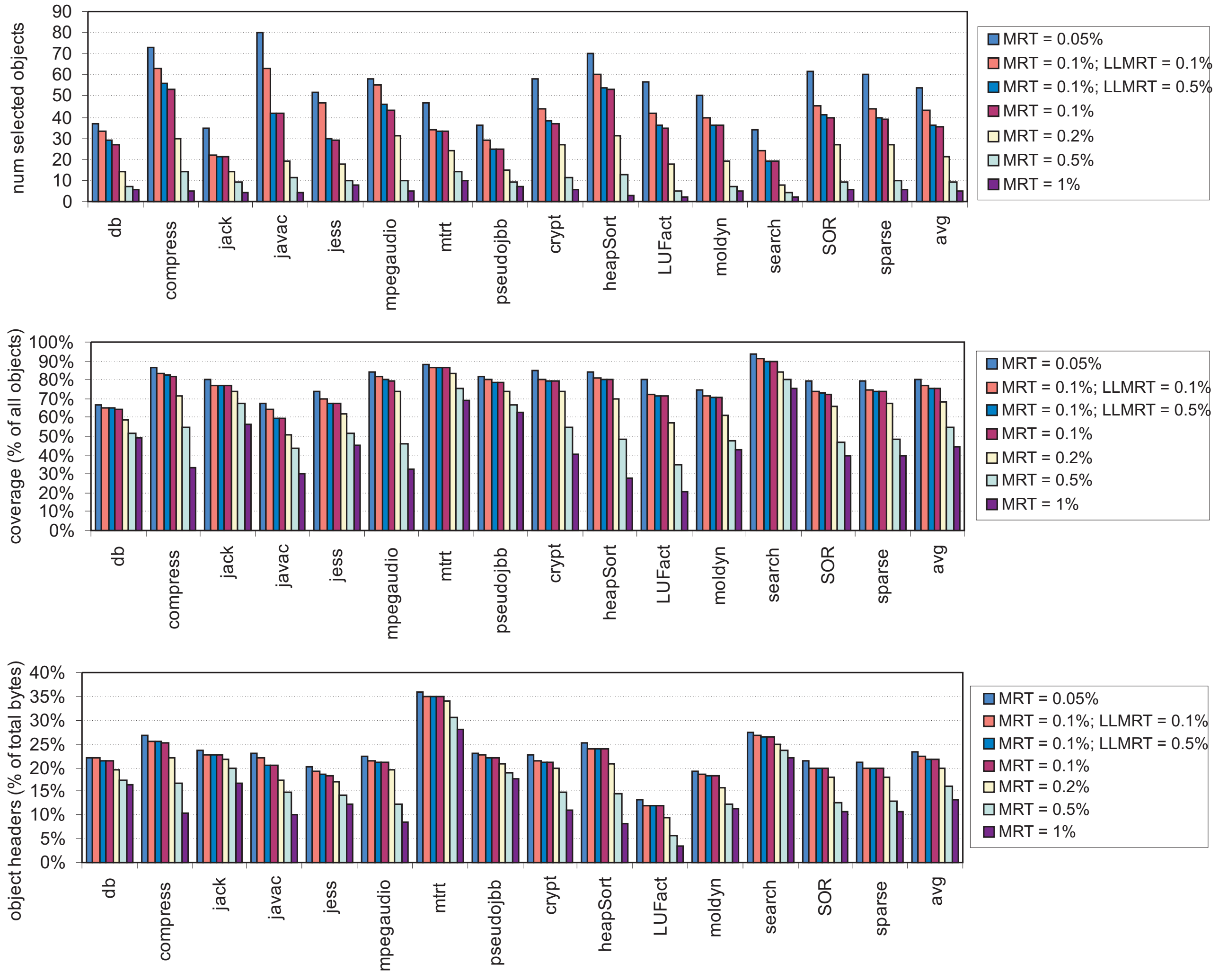

Figure 4. The top graph shows the number of selected object types as a function of the MRT and LLMRT thresholds. The middle graph shows the coverage by the selected object types as a percentage of the total number of allocated bytes. The bottom graph shows the percentage of allocated bytes in the headers of the selected object types.

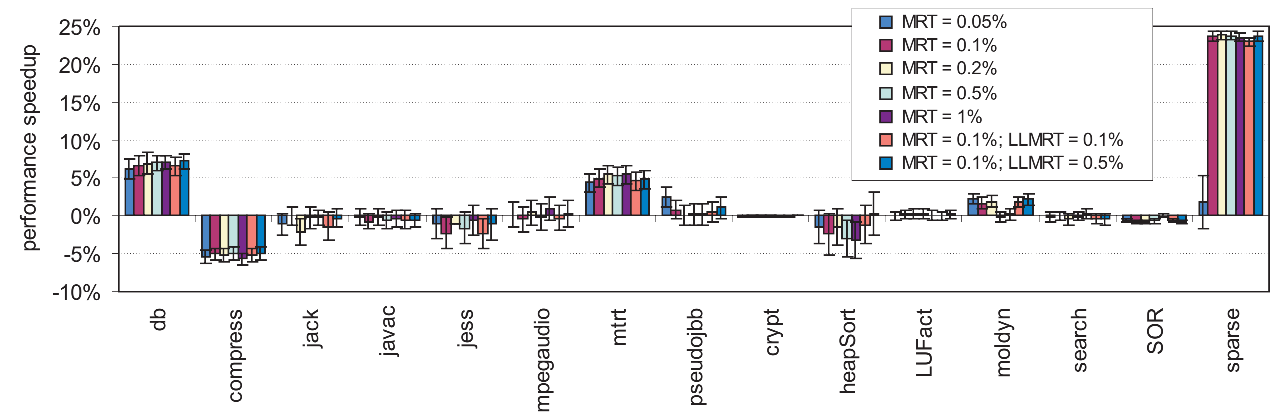

Figure 5. Performance speedups along with the 95% confidence intervals as a function of the MRT and LLMRT thresholds.

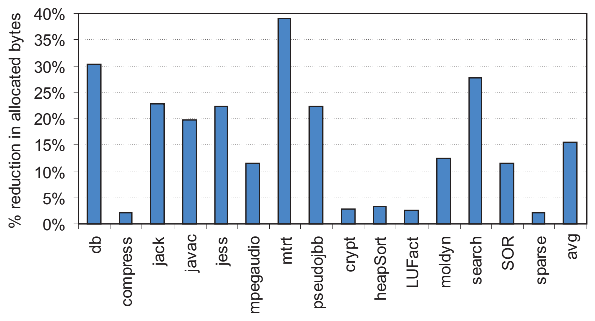

Figure 6. Reduction in the number of allocated bytes for MRT = LLMRT = 0.1%.

performance counters we observed that the L1 data cache miss rate was reduced by 18% for mtrt. For db and sparse, STVA decreases the number of L3 cache misses by as much as 50%. Two other benchmarks show small (but significant) performance improvements: moldyn (2.2%) and pseudojbb (2.4% for $MRT = 0.05\%$ and an infinite LLMRT; other MRT and LLMRT values yield statistically insignificant speedups). For all remaining benchmarks (7 out of the 15 benchmarks in total), STVA has no statistically significant impact on overall performance.

6.3 Impact on memory consumption

Figure 6 shows the reduction in allocated bytes for $MRT = LLMRT = 0.1\%$. Again, these numbers are for the reference input from a cross-validation setup. We observe an average reduction in allocated bytes of 15.5%. For some benchmarks we even observe a reduction in allocated bytes of 28% (search), 30% (db) and 39% (mtrt). There are two important notes that we would like to make related to this data compared to the data presented in Figure 4. First, the data in Figure 6 is for the reference runs whereas Figure 4 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 4. 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 4 shows potential memory reductions in the nursery and mature generations only, no data is included concerning the large object space (LOS). The data

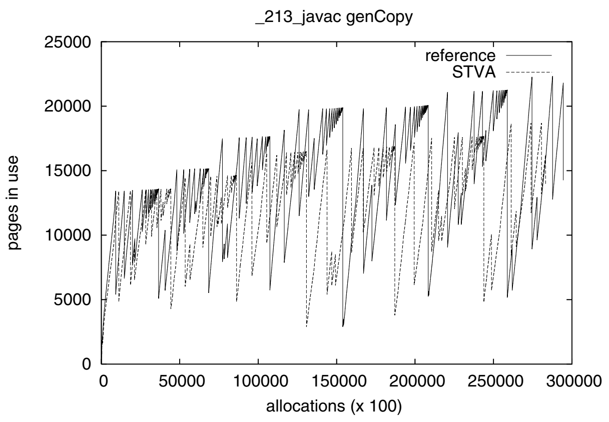

Figure 7. The heap size is shown as a function of the number of allocations for the original Jikes RVM implementation versus STVA for javac.

in Figure 6 shows the effective memory reduction.

Figure 7 shows 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; this is for the javac benchmark. 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 live data at that point. This explains the shape of 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. This is clearly observed in Figure 7 where the STVA curve is shifted to the right compared the original Jikes RVM curve. Second, when garbage is collected, the number of pages in use for STVA drops below the number of pages in use for the original Jikes RVM. The reason is that the amount of live data is smaller under STVA because of the space-efficient STVA object model.

6.4 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 8 where STVA is compared against TVA. TVA results in significant performance degradations for three benchmarks, namely compress (15%), javac (4.6%) and mpegaudio (5.3%). In addition, the performance improvement observed for STVA for sparse (24%) disappears for TVA. As such, we conclude that implicit typing on selected object types outperforms implicit typing on all object types.

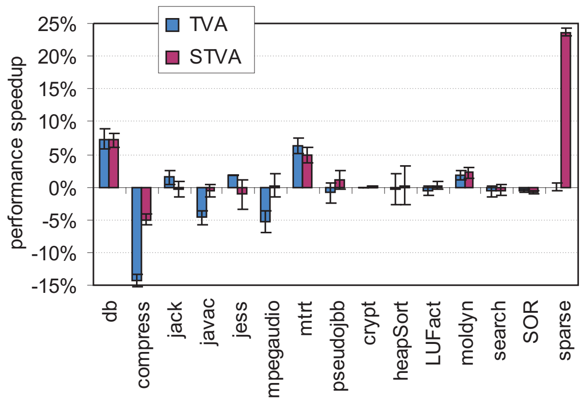

Figure 8. Comparing STVA versus TVA in terms of speedup.

7 Conclusion and future work

This paper proposed Selective Typed Virtual Addressing (STVA) which is an approach for space-efficient 64-bit Java objects. 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. In addition to applying TVA, we also propose an intelligent positioning of the forwarding pointer. This reduces the object's headers of non-array TVA-enabled objects from 16 bytes to only 4 bytes; for array TVA-enabled objects, TVA reduces the header size from 24 bytes to only 8 bytes. We apply TVA selectively, hence the name Selective TVA, on object types that are frequently allocated and/or tend to be long-lived. We evaluate STVA in a 64-bit Java VM implementation, namely Jikes RVM, on an AIX IBM POWER4 machine. The huge virtual address space in 64-bit mode facilitates the implementation of STVA. Our results show that STVA yields a reduction in the number of allocated bytes by 15.5% on average (up to 39%). In terms of performance, STVA yields statistically significant speedups for a number of benchmarks (up to 24%).

It is important to note that this paper is only a first step into making STVA a practical Java 64-bit VM solution. Our current approach requires a profiling step for determining what object types to make TVA-enabled. In future work we will look into making STVA a completely online technique. There are a couple of issues that need to be dealt with. First, making STVA an online technique requires a low-overhead mechanism for determining on what object types to apply implicit typing. Second, an online technique also requires that TVA can be enabled at the object level, not just the class level. Indeed, upgrading an object type from TVA-disabled to TVA-enabled requires that the VM supports that TVA-disabled and TVA-enabled objects from the same type temporarily co-exist at run time. (This is already supported in our current implementation.)

Acknowledgements

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

References

[1] A.-R. Adl-Tabatabai, J. Bharadwaj, M. Cierniak, M. Eng, J. Fang, B. T. Lewis, B. R. Murphy, and J. M. Stichnoth. Improving 64-bit Java IPF performance by compressing heap references. In *CGO-2*, page 100–110, Mar. 2004.

[2] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–238, Feb. 2000.

[3] A. W. Appel. A runtime system. Technical Report CS-TR-220-89, Princeton University, Computer Science Department, May 1989.

[4] D. F. Bacon, S. J. Fink, and D. Grove. Space- and time-efficient implementation of the Java object model. In *ECOOP-16*, pages 111–132, June 2002.

[5] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: Featherweight synchronization for java. In *PLDI*, pages 258–268, June 1998.

[6] S. Dieckmann and U. Hölzle. A study of the allocation behavior of the specjvm98 Java benchmarks. In *ECOOP-13*, pages 92–115, June 1999.

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

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

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

[10] S. T. Shebs and R. R. Kessler. Automatic design and implementation of language data types. In *PLDI*, pages 26–37, June 1987.

[11] Y. Shuf, M. Gupta, R. Bordawekar, and J. P. Singh. Exploiting prolific types for memory management and optimizations. In *POPL*, pages 295–306, Jan. 2002.

[12] G. L. Steele, Jr. Data representation in PDP-10 MACLISP. Technical Report AI Memo 420, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Sept. 1997.

[13] K. Venstermans, L. Eeckhout, and K. De Bosschere. 64-bit versus 32-bit virtual machines for Java. *Software—Practice and Experience*, 26(1):1–26, 2006.