SableVM: A Research Framework for the Efficient Execution of Java Bytecode*

Etienne M. Gagnon and Laurie J. Hendren
Sable Research Group
School of Computer Science
McGill University
[gagnon,hendren]@sable.mcgill.ca

Abstract

SableVM is an open-source virtual machine for Java intended as a research framework for efficient execution of Java bytecode[1]. The framework is essentially composed of an extensible bytecode interpreter using state-of-the-art and innovative techniques. Written in the C programming language, and assuming minimal system dependencies, the interpreter emphasizes high-level techniques to support efficient execution.

In particular, we introduce a *biderectional layout* for object instances that groups reference fields sequentially to allow efficient garbage collection. We also introduce a *sparse interface virtual table layout* that reduces the cost of interface method calls to that of normal virtual calls. Finally, we present a technique to improve thin locks[13] by eliminating busy-wait in presence of contention.

1 Introduction & Motivation

Over the last few years, Java[21] has rapidly become one of the most popular general purpose object-oriented (OO) programming languages. Java programs are compiled into *class files* which include type information and platform independent bytecode instructions. On a specific platform, a runtime system called a *virtual machine*[24] loads and links class files then executes bytecode instructions. The virtual machine collaborates with the standard class libraries to provide key services to Java programs, including threads and synchronization, automatic memory management (garbage collection), safety features (array bound checks, null pointer detection, code verification), reflection, dynamic class loading, and more.[2]

Early Java virtual machines were simple bytecode interpreters. Soon, the quest for efficiency led to the addition of *Just-In-Time compilers* (JIT) to virtual machines, an idea formerly developed for other OO runtime systems like Smalltalk-80[17] and Self-91[15]. In a few words, a just-in-time compiler works by compiling bytecodes to machine specific code on the first invocation of a method. JITs range from the very naive, that use templates to replace each bytecode with a fixed sequence of native code instructions (early versions of Kaffe[5] did this), to the very sophisticated that perform register allocation, instruction scheduling and other scalar optimizations (e.g. [8, 23, 29, 32]).

JITs face two major problems. First, they strive to generate good code in very little time, as compile time is lost to the running application. Second, the code of compiled method resides in memory; this augments the pressure on the memory manager and garbage collector. Recent virtual machines try to overcome these problems. The main trend is to use dynamic strategies to find *hot* execution paths, and only optimize these areas (e.g. [4, 10, 16]). HotSpot[4], for example, is a mixed interpreter and compiler environment. It only compiles and optimizes *hot spots*. Jalapeno[10, 11], on the other hand, always compiles methods (naively at first),

*This research is partly supported by FCAR, NSERC, and Hydro-Québec.

[1] In this document, the term *Java* means: *the Java programming language*.

[2] There exist static compilers that directly compile Java programs to machine code (e.g. [2, 3, 7]). The constraints of static and dynamic environments are quite different. Our research focuses on dynamic Java execution environments.

then uses adaptive online feedback to recompile and optimize hot methods. These techniques are particularly suited to virtual machines executing long running programs in server environments. The optimizer can be relatively slow and consist of a fully fledged optimizing compiler using intermediate representations and performing *costly* aggressive optimizations, as compile time will be amortized on the overall running time.

Our research complements these approaches by exploring opportunities for making the virtual machine execute efficiently. Rather than looking at fine grain techniques, like register allocation and instruction scheduling, we address the fundamental problem of data layout in a dynamic Java environment. While Java shares many properties with other object-oriented languages, the set of runtime constraints enforced by the verifier and the basic services provided for each object (hash code, locking) are unique. This leads us to revisit traditional data structures used in object-oriented runtime environments, and adapt them to fully take advantage of the properties of the Java runtime environment.

As a testbed for evaluating our proposed data structures and algorithms, we are designing and implementing SableVM, a standards conforming open-source virtual machine. Written in the C programming language, and depending on the POSIX application programming interface (API), it is meant as a small and portable interpreter[3]. It can be used as an experimental framework for extending the bytecode language. It can also be used as an efficient virtual machine for embedded systems, or as a profiling interpreter in a hybrid interpreter/just-in-time optimizing-compiler environment.

The remaining part of this document is structured as follows. Section 2, we state the contributions of this paper. In section 3, we give an overview of the SableVM framework. In section 4, we describe SableVM's threaded interpreter. In section 5, we introduce our classification of virtual machine memory. In section 6, we introduce our new layouts for object instances and virtual tables, and our improved thin locks. In section 7, we discuss our proposed experiments. Finally, in section 8, we present our conclusions.

[3]SableVM depends on the open-source *GNU Classpath[1]* class library for providing standard library services.

2 Contribution

The specific contributions of this paper are as follows.

- Introduction of a bidirectional object instance layout that groups reference fields sequentially, enabling simpler and faster garbage collection tracing.
- Introduction of a sparse interface virtual table layout that enables constant time interface method lookup in presence of dynamic loading.
- Improvement of the bimodal field thin lock algorithm[13, 26] to eliminate busy-wait, without overhead in the object instance layout.
- Categorization of virtual machine memory into separate conceptual areas exhibiting different management needs.

3 Framework Overview

As shown in Figure 1, the SableVM experimental framework is a virtual machine composed of five main components: interpreter, memory manager, verifier, class loader, and native interface. In addition, the virtual machine implements various services required by the class library (e.g.: synchronization and threads).

SableVM is entirely[4] written in portable C. Thus, its source code is readable and simple to modify. This makes an ideal framework for testing new high-level implementation features or bytecode language extensions. For example, adding a new arithmetic bytecode instruction entails making a minor modification to the class loader, adding a few rules to the verifier, and finally adding the related interpreter code. This is pretty easy to do in SableVM, as compared to a virtual machine written in assembly language, or a virtual machine with an embedded compiler (e.g. JIT).

The current implementation of SableVM targets the Linux operating System on Intel x86 processors. It

[4]Exceptions: We assume a POSIX system library, we use *label as values* (see Figure 2(b)), and there is a single line of assembly code (*compare-and-swap*).

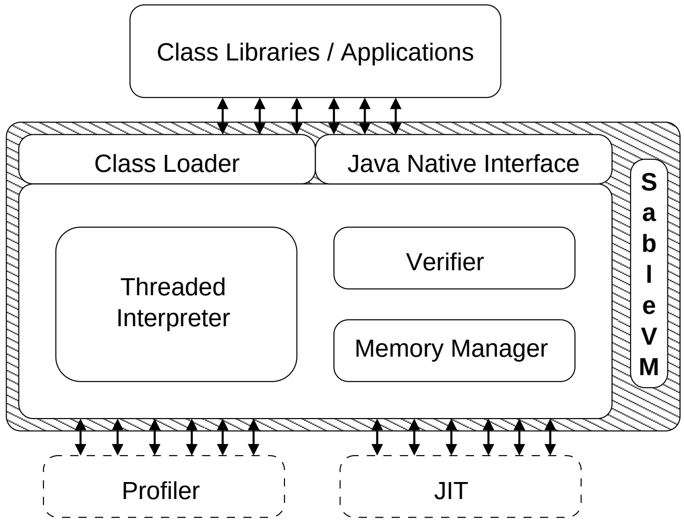

Figure 1: The SableVM experimental framework

uses the GNU libc implementation of POSIX threads to provide preemptive operating system level threads.

4 Threaded Interpreter

SableVM's interpreter is a threaded interpreter. Pure bytecode interpreters suffer from expensive dispatch costs: on every iteration, the dispatch loop fetches the next bytecode, looks up the associated implementation address in a table (explicitly, or through a *switch* statement), then transfers the control to that address. *Direct threading*[20] reduces this overhead: in the executable code stream, each bytecode is replaced by the address of its associated implementation. In addition, each bytecode implementation ends with the code required to dispatch the next opcode. This is illustrated in figure 2. This technique eliminates the table lookup and the central dispatch loop (thus eliminating a branch instruction to the head of the loop). As these operations are expensive on modern processors, this technique has been shown to be quite effective[20, 27].

Method bodies are translated to threaded code on their first invocation. We take advantage of this translation to do some optimizations. For example, we precompute absolute branch destinations, we translate overloaded bytecodes like the `GET_FIELD` instruction to separate implementation addresses (`GET_FIELD_INT`, `GET_FIELD_FLOAT`, ...), and we inline *constant pool* references to direct operand values.

This one pass translation is much simpler than the translation done by even the most naive just-in-time compiler, as each bytecode maps to an address, not a variable sized implementation. However, unlike a JIT, the threaded interpreter still pays the cost of an instruction dispatch for each bytecode. Piumarta[27] has shown a technique to eliminate this overhead within a *basic block* using selective inlining in a portable manner, at the cost of additional memory[5]. SableVM implements this technique optionally through a compile-time flag, as it might not be appropriate for systems with little memory.

5 Memory Management

Memory management is a central issue in the design of SableVM. Most of the high-level performance enhancements introduced in this research are related to memory management.

In this section, we classify the memory of the Java virtual machine according to the control on its management, and its allocation and release behavior. We define four categories (system, shared, thread specific, and class loader specific), and discuss how SableVM takes advantage of them.

[5]On some processors, this technique requires one line of assembly code to synchronize the instruction and data caches.

```
/* code */

char code[] = {
  ICONST_2, ICONST_2,
  ICONST_1, IADD, ...
}
char *pc = code;

/* dispatch loop */

while(true) {
  switch(*pc++) {
  case ICONST_1: *++sp = 1; break;
  case ICONST_2: *++sp = 2; break;
  case IADD:
    sp[-1] += *sp; --sp; break;
  ...
}}
```

(a) Pure bytecode interpreter

```
/* code */

void *code[] = {
  &&ICONST_2, &&ICONST_2,
  &&ICONST_1, &&IADD, ...
}
void **pc = code;

/* implementations */

goto **(pc++);

ICONST_1: *++sp = 1; goto **(pc++);
ICONST_2: *++sp = 2; goto **(pc++);
IADD:
  sp[-1] += *sp; --sp; goto **(pc++);
...
```

(b) Threaded Interpreter

Figure 2: Pure and threaded interpreters

5.1 System Memory

System memory is the portion of memory on which we, as C developers, have essentially no direct control. It consists of the memory used to store executable machine code, native C stacks, the C heap (`malloc()` and `free()`), dynamically linked native libraries, and any other uncontrollable memory.

5.2 Shared Memory

Shared memory is managed by the virtual machine and potentially allocated and modified by many threads executing methods of various class loaders.

This memory consists primarily of the Java heap (which is garbage collected), and global JNI references. The allocation and release behavior of such memory is highly application dependent, with no general allocation or release pattern.

5.3 Thread Specific Memory

Thread specific memory is also managed by the virtual machine, but it is allocated specifically for internal management of each Java thread.

This memory consists primarily of Java stacks, JNI local reference frames for each stack, and internal structures storing thread specific data like stack information, JNI virtual table, and exception status.

This memory exhibits precise allocation and release patterns. Thread specific structures have a life time similar to their related thread. So, this memory can be allocated and freed (or recycled) at the time of respective creation and death of the underlying thread. Also, stacks have a regular pattern: they grow and shrink on one side only. This property is shared by JNI local reference frames.

5.4 Class Loader Specific Memory

Class loader specific memory is managed by the virtual machine and is allocated for internal management of each class loader.

This memory consists primarily of the internal data structures used to store class loader, class, and method data structures. This includes method bodies in their various forms like bytecode, direct threaded code, inlined threaded code, and potentially compiled code (in the presence of a JIT). It also includes normal and interface virtual tables.

This memory exhibits precise allocation and release patterns. This memory is allocated at class loading time, and at various preparation, verification, and resolution execution points. This memory differs significantly from stacks and the shared garbage collected heap in that once it is allocated, it must stay at a fixed location, and it is unlikely to be released soon. The Java virtual machine specification allows for potential unloading of all classes of a class loader as a group, if no direct or indirect references to the class loader, its classes, and theirs instances remain. In such a case, and if a virtual machine supports class unloading, all memory used by a class loader and its classes can be released at once.

5.5 SableVM Implementation

In SableVM, thread specific memory is managed independently from shared memory. SableVM allocates thread structures on thread creation but does not release them at thread death. Instead, it manages a free list to recycle this memory on future thread creation.

Java stacks are growing structures; a memory block is allocated at thread creation, and if later the stack proves too small, the memory block is expanded, possibly moving it to another location to keep the stack contiguous and avoid fragmentation.

SableVM also manages class loader specific memory independently from other memory. Each class loader has it own memory manager that allocates memory (from the system) in relatively big chunks, then redistributes this memory is smaller fragments. This has many advantages.

It allows the allocation of many small memory blocks without the usual memory space overhead, as `malloc()` would use additional memory space to store the size of each allocated block in prevision of future `free()` calls. In the class loader specific memory category, smaller fragments will only be returned to the system as a group (in case of class unloading), so we need not keep track of individual fragment sizes.

As a corollary, class unloading is more efficient using a dedicated memory manager than using regular `malloc()` and `free()` calls, as there is no need to incrementally aggregate small memory segments, as would happen with a sequence of `free()` calls.

Using dedicated memory managers allows class parsing and decoding in one pass without memory overhead, by allocating many small memory blocks. This is usually not feasible, as it is not possible to estimate the memory requirement for storing internal class information before the end of the first pass.

Finally, and importantly, a dedicated memory manager allows for irregular memory management strategies: it is possible to return sub-areas of an allocated block to the memory manager, if these sub-areas are known not to be used. We take advantage of this to improve the representation of interface method lookup tables[6].

6 Performance Enhancements

In this section, we introduce new layouts for object instances and interface virtual tables, as well as improvements to the thin lock algorithm, leading to high-level performance enhancements in the areas of garbage collection, interface method invocation, and synchronization.

We say *high-level* enhancements, because these techniques are applicable to any Java virtual machine, independently from its form: interpreter, just-in-time compiler, adaptive online feedback based systems, etc.

6.1 Bidirectional Object Layout

In this subsection, we propose a new object layout that optimizes the placement of reference fields to allow efficient gc tracing.

The Java heap is by definition a garbage collected area. A Java programmer has no control on the deallocation of an object. Garbage collectors can be divided into two major classes: tracing and non-tracing collectors. Non-tracing collectors (mainly *reference counting*) cannot reclaim cyclic data structures, are a poor fit for concurrent programming models, and have a high reference count maintenance overhead. For this reason, Java virtual machine designers usually opt for a tracing collector.

There exist many tracing collectors[22]. The simplest models are mark-and-sweep, copying, and mark-compact. The common point to all tracing collectors (including advanced generational, conservative and incremental techniques) is that they must trace a subset of the heap, starting from a root set, looking for *reachable* objects. Tracing is often one of the most expensive steps of garbage collection[22]. For every root, the *garbage collector* (gc) looks up the type of the object to find the offset of its reference fields, then it recursively visits the objects referenced by these fields.

[6] See section 6.2.

To provide efficient field access, it is desirable to place fields at a constant offset from the object header, regardless of inheritance. This is easily achieved in Java as instance fields can only be declared in classes (not in interfaces), and classes are restricted to single inheritance. Fields are laid out consecutively after the object header, starting with super class fields then subclass fields, as shown in Figure 3(a). When tracing such an object, the garbage collector must access the object's class information to discover the offset of its reference fields, then access the superclass information to obtain the offset of its reference fields, and so on. As this process must be repeated for each traced object, it is quite expensive.

There are three improvements that are usually applied to this naive representation. Firstly, reference fields are grouped together in the layout of each class. Secondly, each class stores an array of offsets and counts of reference fields for itself and all its super classes. Thirdly, a *one word* bit array is used in the virtual table to represent the layout of reference fields in small objects (each bit being set if the object instance word, at the same index, is a reference). This is shown in Figure 3(b). For big objects, the number of memory accesses needed to trace an object is $n + 3 + (2 * arraysize)$, where n is the number of references. Two nested loops (and loop variables) are required: one to traverse the array, and one for each array element (accessing the related number of references). For smaller objects, the gc needs to access the virtual table to retrieve the bit field word, then it needs to perform a set of shift and mask operations to find the offset of reference fields. Overall, using this layout, tracing an object is a relatively complex operation.

Tracing reference fields could be much simpler if they were simply grouped consecutively. The difficulty is to group them while keeping the *constant offset* property in presence of inheritance.

We introduce a *bidirectional object instance layout* that groups reference fields while maintaining the *constant offset* property. The left part of Figure 4 illustrates this new layout. In the bidirectional object instance layout, the instance *starting point* is possibly a reference field. The instance grows both ways from the *object header*, which is located in the middle of the instance. References are placed before the header, and other fields are placed after it. The right part of Figure 4 illustrates the layout of array instances. Array element are placed in front or after the array instance header, depending on whether the element type is a reference or a non-reference type, respectively.

The object header contains two words (three for arrays). The first is a *lock word* and the second is a virtual table pointer. We use a few low-order bits of the lockword encode the following information:

- We set the last (lowest order) bit to one, to differentiate the lock word from the preceding reference fields (which are pointers to aligned objects, thus have their last bit set to zero).
- We use another bit to encode whether the instance is an object or an array.
- If it is an array, we use 4 bits to encode its element type (boolean, byte, short, char, int, long, float, double, or reference).
- If it is an object, we use a few bits to encode (1) the number of references and (2) the number of non-reference field words of the object, (or special overflow values, if the object is too big).

We also use two words of the virtual table (see Figure 5) to encode the number of reference and non-reference field words of the object if the object is too big to encode this information in the header.

At this point, we must distinguish the two ways in which an object instance can be reached by a tracing collector. The first way is through an object reference that points to the object header (which is in the middle of the object). The second way is through its starting point, in the *sweep* phase of a mark-and-sweep gc, or in the *tospace* scanning of a copying gc. In both cases, our bidirectional layout allows the implementation of simple and elegant tracing algorithms.

In the first case, the gc accesses the lock word to get the number of references n (one shift, one mask). If n is the overflow value (big object), then n is retrieved from the virtual table. Finally, the gc simply traces n references in front of the object header.

In the second case, the object instance is reached from its starting point in memory, which might be either a reference field or the object header (if there are no reference fields in this instance). At this point, the gc must find out whether the initial word

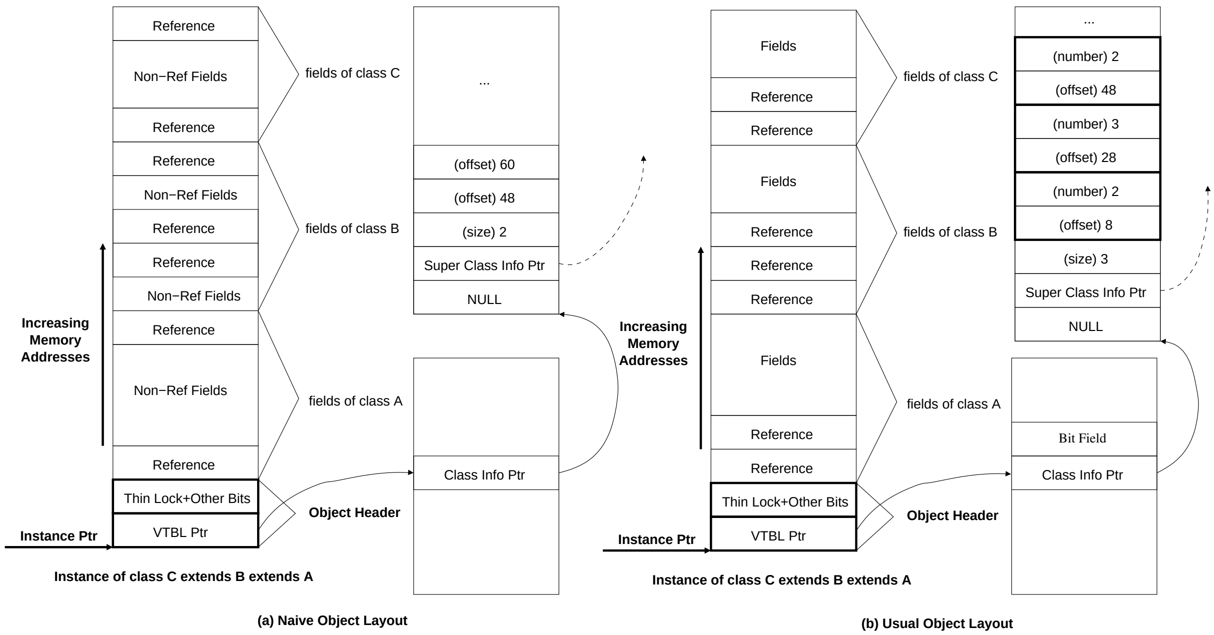

Figure 3: Traditional layout

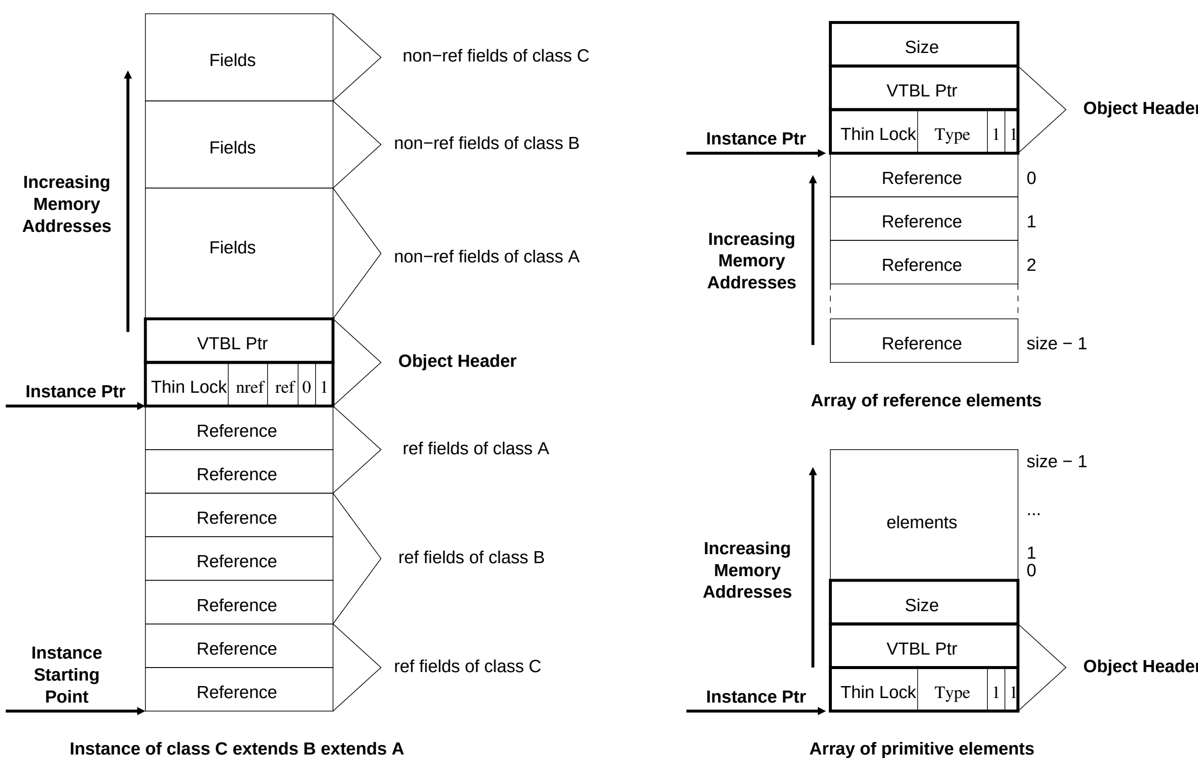

Figure 4: Bidirectional layout

is a reference or a lock word. But, this is easy to find. The gc needs simply check the state of the last bit of the word. If it is one, then the word is a lock word. If it is zero, then the word is a reference.

So, for example, a copying collector, while scanning the *tospace* needs only read words consecutively, checking the last bit. When set to zero, the discovered reference is traced, when set to 1, the number of non-reference field words (encoded in the lock word itself, or in the virtual table on overflow) is used to find the starting point of the next instance.

In summary, using our bidirectional layout, a gc only accesses the following memory locations while tracing: reference fields and lock word, for all instances (objects and arrays), and at most three additional accesses for objects with many fields (virtual table pointer and two words in the virtual table itself).

While our work on bidirectional objects for grouping references is new, we mention some previous related work. The idea of using a bidirectional object layout (without grouping references) has been investigated[25, 28] as a mean to provide efficient access to instance data and dispatch information in languages supporting multiple inheritance (most specifically C++). In [14], Bartlett proposed a garbage collector which required grouping pointers at the head of structures; this was not achieved using bidirectional structs.

6.2 Sparse Interface Virtual Tables

In this subsection, we present a virtual table layout that eliminates the overhead of interface method lookup over normal virtual method lookup.

This enhancement addresses a problem raised by multiple inheritance of interfaces in Java. The virtual machine instruction set contains an *invokeinterface* instruction, used to invoke interface methods. A common technique to implement this instruction is to prepare multiple virtual tables for each class: a main virtual table used for normal virtual method invocation, and one additional virtual table for each interface directly or indirectly implemented by the class[5]. Each method declared in an interface is given an index within its virtual table. After preparation, each *invokeinterface* has two arguments: an interface number, and a method index. On execution, the *invokeinterface* instruction operates its method lookup in two steps. It first lookups up the appropriate virtual table (using linear, binary, or hashed search), then it retrieves the method pointer in a single operation from the virtual table entry located at the given method index. This interface lookup procedure has the following overhead over normal virtual method lookup: it needs to do a search to find the appropriate virtual table. It would be possible to implement a constant time lookup using *compact encoding*[31], but unfortunately, dynamic class loading requires updating this information dynamically, which is difficult to do in a multi-threaded Java environment. Our approach is simple, and does not require dynamic recomputation of tables or code rewrite.

The idea of maintaining multiple virtual tables in case of multiple inheritance is reminiscent of C++ implementations[19]. But, Java's multiple inheritance has a major semantic difference: it only applies to interfaces which may only declare method signatures without providing an implementation. Furthermore, if a Java class implements two distinct interfaces which declare the same method signature, this class satisfies both interfaces by providing a single implementation of this method. C++ allows the inheritance of distinct implementations of the same method signature.

We take advantage of this important difference to rethink the appropriate data structure needed for efficient interface method lookup. Our ideas originate from previous work on efficient method lookup in dynamically typed OO languages using of *selector-indexed dispatch tables*[12, 18, 30]. We assign a globally unique increasing index[7] to each method signature declared in an interface. A method signature declared in multiple interfaces has a single index. When the virtual table of a class is created, we also create an *interface virtual table* that grows down from the normal virtual table. This interface virtual table has a size equal to the highest index of all methods declared in the direct and indirect super interfaces of the class. For every declared super interface method, the entry at its index is filled with the address of its implementation. Interface invocation is then encoded with the *invokeinterface* instruction, and a single interface method index. The execution of *invokeinterface* can then proceed at the *exact same* cost as an *invokevirtual*.

[7] In reality, we use a decreasing index, starting at at -1, to allow direct indexing in the *interface virtual table*.

The interface virtual table is a sparse array of method pointers. As more interfaces are loaded, with many interface method signatures, the amount of free space in interface virtual tables grows. The traditional approach has been to use table compression techniques to reduce the amount of free space. However, these techniques are poorly adapted to concurrent and dynamic class loading environments like the Java virtual machine, as they require dynamic recompilation.

Our approach differs. Instead of compressing interface virtual tables, we simply return the free space in them to the related class loader memory manager (see section 5.4). This memory is then used to store all kinds of other class loader related data structures. In other words, we simply recycle the free space of sparse interface virtual tables within a class loader. The layout of interface virtual tables is illustrated in Figure 5.

As interface usage in most Java programs range from very low to moderate, we could argue that it is unlikely that the free space returned by interface virtual tables will grow faster than the rate at which it is recycled. However, in order to handle pathological cases, we also provide a very simple technique, which incurs no runtime overhead, to limit the maximal growth of interface virtual tables. To limit this growth to N entries, we stop allocating new interface method indices as soon as index N is given. Then, new interface method signatures are encoded using traditional techniques. The trick to make this work is to encode interface calls differently, based on whether the invoked method signature has been assigned an index or not. The traditional technique used to handle overflow can safely ignore all interface methods which have already been assigned an index.

6.3 Improved Thin Locks

Our final enhancement improves upon Onodera's bimodal field locking algorithm[26], a modified version of Bacon's thin lock algorithm[13], but without busy-wait transition from light to heavy mode.

Bacon's thin lock algorithm can be summarized as follows. Each object instance has a one lock word in its header[8]. To acquire the lock of an object, a thread uses the *compare-and-swap* atomic operation to compare the current lock value to zero, and replace it with its thread identifier. If the lock value isn't zero, this means that either the lock is already inflated, in which case a normal locking procedure is applied, or the lock is thin and is already acquired by some thread. In the latter case, if the owning thread is the current one, a nesting count (in the lock word) is increased. If the owning thread is not the current one, then there is contention, and Bacon's version of the algorithm busy-waits, spinning until it acquires the lock. When it is finally acquired, it is inflated. Unlocking non-inflated locks is simple. On each unlock operation, the nesting count is decreased. When it reaches 0, the lock byte is replaced by zero, releasing the lock.

The advantages of this algorithm are that a single atomic operation is needed to acquire a thin lock in absence of contention, and more importantly, no atomic operation is required to unlock an object[9].

Onodera eliminates the busy wait in case of contention on a thin lock, using a single additional bit in each object instance. The role of this *contention bit* is to indicate that some other thread is waiting to acquire the current thin lock. Onodera's algorithm differs from the previous algorithm at two points. First, when a thread fails to acquire a thin lock (because of contention), it acquires a fat monitor for the object, sets the contention bit, checks that the thin lock was not released, then puts itself in a waiting state. Second, when a thin lock is released (e.g. lock word is replaced by zero), the releasing thread checks the contention bit. If it is set, it inflates the lock, and notifies all waiting threads[10].

The overhead of Onodera's algorithm over Bacon's is the contention bit test on unlocking, a fairly simple non-atomic operation, and the one bit per object. This bit has the following restriction: it must not reside in the lock word. This is a problem. It is important to keep the per object space overhead as low as possible, as Java programs tend allocate many small objects. It is now common practice to use 2 word headers in object instances; one word for the virtual pointer, and the second for the lock and other information. The contention bit cannot reside in either of these two words (putting it in the virtual table pointer word would add execution overhead to

[8] Only 24 bits of that word are used for locking on 32 bit systems. 8 bits remain free for other uses.

[9] Unlike Agesen's recent *meta-lock* algorithm[9] which requires an atomic operation for unlocking objects.

[10] This is a simplified description. Please refer to the original paper[26] for details.

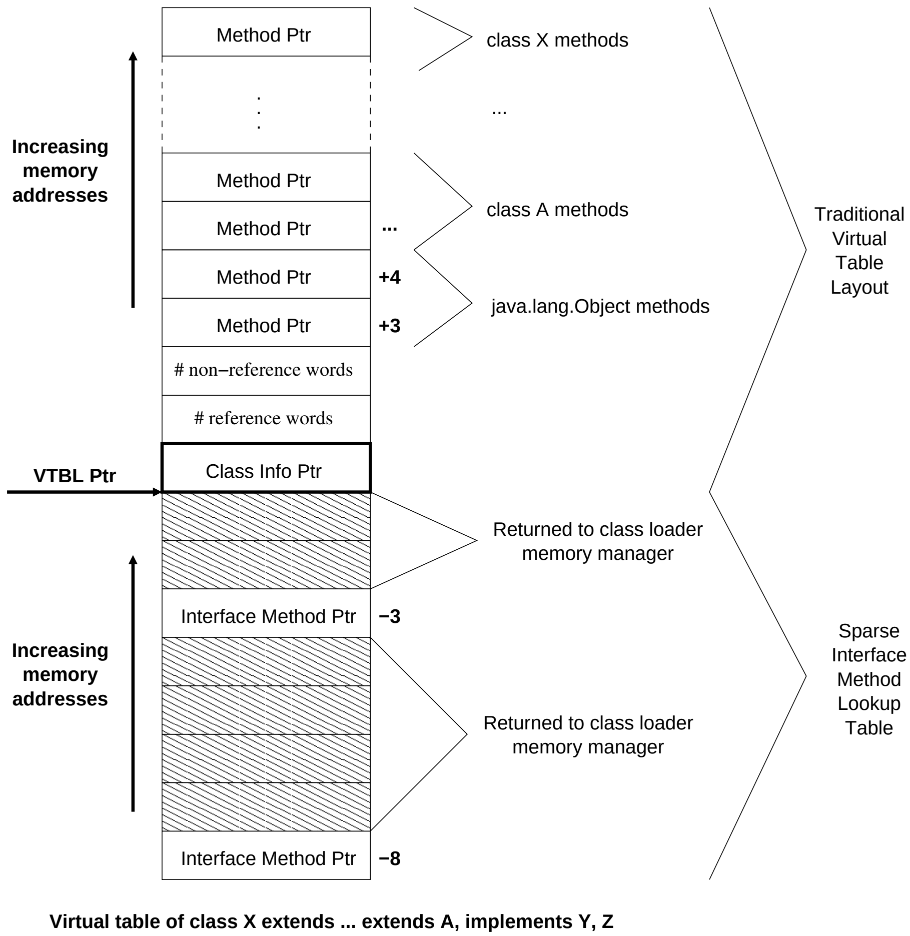

Figure 5: Virtual table layout

method invocation, field access, and any other operation dereferencing this pointer). As objects need to be aligned on a word multiple (for the atomic operation to work), this one bit overhead might well translate into a whole word overhead for small objects. Also, it is likely that the placement of this bit will be highly type dependent, which complicates the unlocking test.

Our solution to this problem is to put the *contention bit* in the thread structure, instead of in the object instance. This simple modification has the advantage of eliminating the per object overhead while maintaining the key properties of the algorithm, namely, fast thin lock acquisition with a single atomic operation, fast thin lock unlocking without atomic operations, and no busy-wait in case of contention.

We modify Onodera's algorithm as follows. In SableVM, each thread has a related data structure containing various information, like stack information and exception status. In this structure, we add the contention bit, a *contention lock*[11], and a linked list of (waiting thread, object) tuples. Then we modify the lock and unlock operation as described in the following two subsections.

6.3.1 Modifications to the lock operation

The lock operation is only modified in the case of contention on a thin lock.

When a thread x_t fails to acquire a thin lock on object z_o due to contention (because thread y_t already owns the thin lock), then (1) thread x_t acquires the contention lock of the owning thread (y_t), and (2) sets the contention bit of thread y_t, then (3) checks that the lock of object z_o is still thin and owned by

[11] The contention lock is a simple non-recursive mutex.

thread y_t. If the check fails, (4a) the contention bit is restored to its initial value, the contention lock is released and the lock operation is repeated. If the check succeeds, (4b) the tuple $(x_t,\ z_o)$ is added to the linked list of thread y_t, then thread x_t is put in the waiting state (temporarily releasing the contention lock of thread y_t, while it sleeps). Later, when thread x_t wakes up (because it was signaled), it releases the re-acquired contention lock and repeats the lock operation.

6.3.2 Modifications to the unlock operation

The unlock operation is modified to check the contention bit of the currently executing thread. This check is only done when a lock is actually released (as locks are recursive), after releasing the lock.

When the lock of object b_o is released by thread y_t, and if the contention bit of thread y_t is set, then (1) thread y_t acquires its own contention lock, and (2) iterates over all the elements of its tuple linked list. For each tuple $(x_t,\ z_o)$, if $(z_0 = b_o)$, thread x_t is simply signaled. If $(z_o \neq b_o)$, the lock of object z_o is inflated[12] (if it is thin), then thread x_t is signaled. Finally, (3) thread y_t empties its tuple linked list, clears its contention bit, and releases its contention lock.

7 Experimentation

We are conducting the following experiments, to evaluate the various memory management and performance enhancement strategies.

- Implementation of both standard and bidirectional instance layout, and comparison of the tracing speed of SableVM's copying collector on both layouts.
- Measure the memory overhead of *sparse interface method lookup tables* in application benchmarks. Test, using micro benchmarks, SableVM's behavior in presence of pathological cases.
- Measure the size of *class loader memory fragments* returned to the memory manager for recycling. Measure how much of this memory gets effectively reused. Explore the possibility of not managing these fragments if the storage they require is insignificant.
- Evaluate relative speed of SableVM compared to the speed of other Java virtual machines running on Linux, using a set of standard benchmarks, and some micro benchmarks.

The results of these experiments can be found on the following web site [6].

8 Conclusion and Future Work

In this paper, we have presented SableVM, a framework for testing high-level performance enhancements and extensions to the Java virtual machine. SableVM is written in portable C with minimal system dependecies.

The main goal of the SableVM project was the design and implementation of an open-source virtual machine suitable for research which is easy to modify, can simply handle language and bytecode extensions, and also provides a testbed for various implementation strategies.

Particularly, we have introduced in this paper new high-level techniques usable by any Java virtual machine (including JITs, and hybrid systems) to support efficient execution of Java bytecode.

More specifically, we have introduce a bidirectional object layout that groups reference fields, and we showed how this layout makes tracing objects and arrays simple and efficient.

We also introduced a sparse interface virtual table layout adapted to the dynamic class loading facility of Java, which reduces the cost of an invokeinterface instruction to that of an invokevirtual. We also demonstrated that the sparse representation need not waste memory, because unused holes in the interface table could be recycled and used by the class loader memory manager.

Our last performance enhancement technique was an improvement on thin locks. We introduced a simple algorithm and related data structures that

[12] Notice that thread y_t necessarily owns the lock of object z_o, as a only one lock (on object b_o) has been released by thread y_t since it last cleared its contention bit and emptied its tuple list.

eliminate busy-wait in case of contention on a thin lock. This strategy incurs no space overhead on object instances.

Other groups have expressed an interest in adding other components to the VM, including a JIT compiler. We encourage such collaboration. The SableVM source is publicly-available at: `http://www.sablevm.org/`.

References

[1] Classpath. www.classpath.org/.

[2] GCJ. sources.redhat.com/java/.

[3] Harissa. www.irisa.fr/compose/harissa/harissa.html.

[4] HotSpot. java.sun.com/products/hotspot/whitepaper.html.

[5] Kaffe. www.kaffe.org/.

[6] SableVM. www.sablevm.org/.

[7] Toba. www.cs.arizona.edu/sumatra/toba/.

[8] Ali-Reza Adl-Tabatabai, Michał Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast, effective code generation in a Just-in-Time Java compiler. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 280–290. ACM Press, 1998.

[9] Ole Agesen, David Detlefs, Alex Garthwaite, Ross Knippel, Y. S. Ramakrishna, and Derek White. An efficient meta-lock for implementing ubiquitous synchronization. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 207–222. ACM Press, November 1999.

[10] 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, October 2000.

[11] Bowen Alpern, Dick Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, and John J. Barton. Implementing Jalapeno in Java. In *Proceeings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34.10 of *ACM Sigplan Notices*, pages 314–324. ACM Press, November 1999.

[12] Cox B. *Object-Oriented Programming: An evolutionary Approach*. Addison-Wesley, 1987.

[13] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin locks: Featherweight synchronization for Java. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 258–268. ACM Press, June 1998.

[14] Joel F. Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88.2, Digital – Western Research Laboratory, 1988.

[15] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 24, pages 49–70. ACM Press, October 1989.

[16] Michał Cierniak, Guei-Yuan Lueh, and James N. Stichnoth. Practicing JUDO: Java under dynamic optimizations. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 13–26, Vancouver, British Columbia, June 2000. ACM Press.

[17] Peter Deutsch and Alan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 297–302. ACM Press, January 1984.

[18] Karel Driesen. Selector table indexing & sparse arrays. *SIGPLAN Notices: Proc. 8th Annual Conf. Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*, 28(10):259–270, September 1993.

[19] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, December 1990.

[20] Anton M. Ertl. A portable Forth engine. www.complang.tuwien.ac.at/forth/threaded-code.html.

[21] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, 2000.

[22] Richard Jones and Rafael Lins. *Garbage collection: algorithms for automatic dynamic memory management*. Wiley, 1996.

[23] Andreas Krall. Efficient JavaVM Just-in-Time compilation. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT '98)*, pages 205–212. IEEE Computer Society Press, October 1998.

[24] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, 1999.

[25] Andrew C. Myers. Bidirectional object layout for separate compilation. In *OOPSLA '95 Conference Proceedings: Object-Oriented Programming*

Systems, Languages, and Applications, pages 124–139. ACM Press, October 1995.

[26] Tamiya Onodera and Kiyokuni Kawachiya. A study of locking objects with bimodal fields. In *Proceeings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34.10 of *ACM Sigplan Notices*, pages 223–237. ACM Press, November 1999.

[27] Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 291–300. ACM Press, June 1998.

[28] William Pugh and Grant Weddell. Two-directional record layout for multiple inheritance. *ACM SIGPLAN Notices*, 25(6):85–91, June 1990.

[29] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal*, 39(1):175–193, 2000.

[30] Jan Vitek and R. Nigel Horspool. Taming message passing: Efficient method look-up for dynamically typed languages. In *Object-Oriented Programming, Proceedings of the 8th European Conference ECOOP'94*, volume 821 of *Lecture Notes in Computer Science*, pages 432–449. Springer, July 1994.

[31] Jan Vitek, R. Nigel Horspool, and Andreas Krall. Efficient type inclusion tests. In *OOPSLA '97 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications*, pages 142–157. ACM Press, October 1997.

[32] Byung-Sun Yang, Soo-Mook Moon, Seongbae Park, Junpyo Lee, SeungIl Lee, Jinpyo Park, Yoo C. Chung, Suhyun Kim, Kemal Ebcioğlu, and Erik Altman. LaTTe: A Java VM Just-in-Time compiler with fast and efficient register allocation. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT '99)*, pages 128–138. IEEE Computer Society Press, October 1999.