To appear in the European Conf. on Object-Oriented Programming, June, 2002, Málaga, Spain

Space- and Time-Efficient Implementation of the Java Object Model

David F. Bacon, Stephen J. Fink, and David Grove

IBM T.J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598, U.S.A.
dfb@watson.ibm.com sjfink@us.ibm.com groved@us.ibm.com

Abstract. While many object-oriented languages impose space overhead of only one word per object to support features like virtual method dispatch, Java's richer functionality has led to implementations that require two or three header words per object. This space overhead increases memory usage and attendant garbage collection costs, reduces cache locality, and constrains programmers who might naturally solve a problem by using large numbers of small objects.

In this paper, we show that with careful engineering, a high-performance virtual machine can instantiate most Java objects with only a single-word object header. The single header word provides fast access to the virtual method table, allowing for quick method invocation. The implementation represents other per-object data (lock state, hash code, and garbage collection flags) using heuristic compression techniques. The heuristic retains two-word headers, containing thin lock state, only for objects that have synchronized methods.

We describe the implementation of various object models in the IBM Jikes Research Virtual Machine, by introducing a pluggable object model abstraction into the virtual machine implementation. We compare an object model with a two-word header with three different object models with single-word headers. Experimental results show that the object header compression techniques give a mean space savings of 7%, with savings of up to 21%. Compared to the two-word headers, the compressed space-encodings result in application speedups ranging from -1.5% to $+2.2\%$. Performance on synthetic micro-benchmarks ranges from $+23\%$ due to benefits from reduced object size, to -12% on a stress test of virtual method invocation.

1 Introduction

The choice of *object model* plays a central role in the design of any object-oriented language implementation. The object model dictates how to represent objects in storage. The best object model will maximize efficiency of frequent language operations while minimizing storage overhead.

A fundamental property of object-oriented languages is that the operations performed on an object depend upon the object's *run-time* type, rather than its compile-time type. Therefore, in any object model, each object must at a minimum contain some sort of run-time type identifier, typically a pointer to a virtual method table.

Some modern object-oriented languages, like Java[1], require additional per-object state to support richer functionality including garbage collection, hashing, and synchronization. Conventional wisdom holds that since an object's virtual method table is accessed so frequently, any attempt to encode extra information into that header word would seriously degrade performance. Therefore, existing Java systems all require at least two words (and in some cases three words) of header space for each object.

This paper makes the following contributions regarding object models for Java and similar object-oriented languages:

- we describe a variety of composable header compression techniques;
- we show how these techniques can be composed into a variety of object models requiring only one word of space overhead per object;
- we show how the object models can all be implemented in one plug-compatible framework; and
- we show that the compressed object headers can improve mean run-time performance up to 2.3%, and even the most aggressive space compression leads to a mean run-time slowdown of only 1.6% while reducing space consumption by a mean of 7% (14% ignoring two programs that mainly manipulate very large arrays).

In summary, our work shows that in the presence of a high-quality JIT compiler, conventional wisdom is wrong: encoding the method table pointer, when engineered carefully, has a negligible run-time performance impact while saving significant space. This result is significant in that once adopted, it will encourage programmers to use large numbers of small objects more freely. This in turn will improve the quality and maintainability of code.

Our results should apply to other object-oriented languages, such as Smalltalk, Modula-3, SELF, Sather, and Oberon, with the obvious caveat that the more similar the language is to Java, the more directly translatable the results should be.

We have implemented the pluggable object model framework and four different object models in the Jikes[2] Research Virtual Machine [15] (formerly known as Jalapeño [6]), and present detailed measurements from this implementation. The Jikes RVM is an open-source Java virtual machine that includes a high-quality optimizing JIT compiler. The object model framework and implementations described in this paper are available in the open-source release of Jikes RVM version 2.0.4 and later.

The rest of the paper is organized as follows: Section 2 describes the abstract, plug-compatible object model. Section 3 describes the various header compression techniques that we use. Section 4 describes the four object models that we implemented: a standard object model with a two-word header, and three different object models with single-word headers. Section 5 presents our measurements in detail. Section 6 compares related work, and is followed by our conclusions.

[1] **Java**™ and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

[2] **Jikes**™ is a trademark or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

2 Object Model Abstraction

Any object model implementation must provide a basic set of functionality for other parts of the virtual machine. In this study, we will assume that any object model implementation must instantiate a common abstract object model. The object model provides access to the following abstract fields for each heap-allocated object:

TIB Pointer The TIB (Type Information Block) holds information that applies to all objects of a type. Each object points to a TIB, which could be a class object or some other related object. In the Jikes RVM, the TIB includes the virtual method table, a pointer to an object representing the type, and pointers to a few data structures to facilitate efficient interface invocation [4] and dynamic type checking [5].

Default Hash Code Each Java object has a default hash code.

Lock Each Java object has an associated lock state. This could be a pointer to a lock object or a direct representation of the lock.

Garbage Collection Information Each Java object has associated information used by the memory management system. Usually this consists of one or two mark bits, but this could also include some combination of a reference count, forwarding pointer, etc.

Additionally, each array object provides a *length* field, and in certain experimental configurations each object header may contain one or more fields for profiling.

This paper explores various implementations of this abstract object model.

For example, one could implement an object model where each object has a four-word header, with one word devoted to each abstract field. This implementation would support 2^{32} distinct values for each field. This is usually overkill, as other considerations restrict the range of distinct values for each field.

2.1 Pluggable Implementation

In order to facilitate an apples-to-apples comparison of different object models in a high performance virtual machine, we have modified the Jikes RVM to delegate all access to the object model through a (logically) abstract class, called `VM_ObjectModel`. The fact that the Jikes RVM is implemented in Java made introducing this new abstraction fairly straightforward. We select and plug in a `VM_ObjectModel` implementation at system build time, allowing the system-building compiler to specialize all components of the virtual machine for the chosen object model implementation.

The `VM_ObjectModel` class provides the following services to the rest of the the virtual machine:

Getters and setters for each field in the abstract object model The Jikes RVM runtime services (class loaders, garbage collectors, compilers, etc.) perform all accesses to the fields through these methods.

Compiler inline code generation stubs For high performance dynamic type checks and virtual method dispatch, the compilers must generate inline code to access an object's TIB. `VM_ObjectModel` provides small callbacks for the compilers to generate the appropriate inline code sequence.

Locking entry points Since the synchronization implementation depends on the object model, all locking calls delegate to `VM_ObjectModel`.

Allocator support `VM_ObjectModel` provides services to the memory management system to compute the size of an object and to initialize a chunk of raw memory as an object of a given type.

Note that the system-building compiler inlines the getter, setter, and other forwarding methods, so that abstracting the object model in this fashion has no runtime costs.

3 Header Compression Techniques

In this section we describe various compression techniques for each of the object header components.

3.1 TIB Pointer

The virtual machine uses the TIB pointer for virtual method calls, dynamic type checks, and other operations based on the object's run-time type. We examine three basic compression techniques that can be applied to the TIB pointer: *bit-stealing*, *indirection*, and *implicit type*.

Bit-stealing exploits the property that some bits of the TIB pointer always have the same value (usually 0) and allocates those bits for other uses. When bit-stealing, the virtual machine must perform a short sequence of ALU operations (on PowerPC[3], a single rotate-and-mask instruction) to extract the TIB pointer from the object header. Most commonly, the implementation can steal the low-order two or three bits, since TIBs are generally aligned on four- or eight-byte boundaries. The implementation may steal high order bits of the word as well, if the TIB always resides in a particular memory segment.

The bit-stealing technique has the advantages of low runtime overhead and not requiring any additional loads. The technique's main disadvantage is that it generally frees only a few bits for other uses.

A more general technique, indirection, represents the TIB pointer as an index into a table of TIB pointers. Usually, indirection can free more bits than bit-stealing, since the table can pack TIB pointers more densely than TIB objects can be packed in memory. Furthermore, the total number of types should be several orders of magnitude smaller than the number of objects.

The disadvantages of indirection are that it requires an extra load to access the TIB pointer, and the table both consumes space and imposes a fixed limit on the number of TIBs.

A third technique, the implicit type method [11], reserves a range of memory for objects that all share the same type. If the memory range is a page, then the TIB pointer can be computed by shifting the object address to obtain a page number that is used as an index into a table that maps pages to TIB pointers.

[3] **PowerPC**™ is a trademark or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

The main advantage of this approach is that it frees all 32 bits typically associated with the TIB pointer, potentially allowing objects with no header space overhead. Like indirection, it requires only a single ALU operation followed by a single load to obtain the TIB pointer. The disadvantage is that it can lead to significant fragmentation of the heap, since all the objects on a page must be of the same type.

3.2 Synchronization State

In previous work, Bacon et al. [7] showed that using an in-object "thin lock" for the common case when locking does not involve contention can yield application speedups of up to a factor of two for Java programs due to the thread-safe design of many core libraries. The thin lock consists of a partial word used by inlined code that attempts to lock the object using a compare-and-swap style atomic operation. Out-of-line code handles the uncommon case when the compare-and-swap fails.

While fast, a thin lock for every object introduces a space cost in each object's header. Furthermore, in most programs, the majority of objects are never locked. We observe that in most cases, a simple static heuristic can predict whether an object of a particular class is likely to be locked. The heuristic predicts that an object of a class C is likely to be locked if and only if C has at least one synchronized method, or if any of its methods contain synchronized(this) statements.

Put another way, we consider locks as instance variables that are implicitly defined in the first class in the hierarchy that has a synchronized method or statement. Objects without the implicit lock variable (in particular, arrays), will not have a synchronization instance variable and must always resort to a more heavyweight locking scheme, mediated through a hash table.

Note that all synchronized methods and synchronized statements whose argument types have lock instance variables can always have the offset of the lock variable generated as a compile-time constant, leading to highly efficient inline locking code. Also note that since the space overhead of lock words is eliminated from most objects, there is no need for locking schemes that attempt to use only a small number of bits until the object is locked, such as meta-locks [1].

This heuristic will not catch a common idiom where program uses an object of type java.lang.Object as a lock. For such cases, we provide a type Synchronizer with a dummy synchronized method that forces allocation of a thin lock in the object header. Naturally, legacy code using this idiom will still suffer performance degradation. One could provide tool support with static or dynamic analysis to help identify troublesome cases. Alternatively, as a subject for future work, an adaptive optimization system could detect excessive locking overhead dynamically and introduce a thin lock word for particular object instances as needed.

The Lock Nursery To implement the lock space optimization, we introduce the *lock nursery*, a data structure holding lock state for objects that do not have thin locks allocated in the object header. The implementation finds an object's entry in the lock nursery via its hash code.

Hash-table based locking schemes are notoriously slow. The original Sun Java Virtual Machine used a hash table scheme in which locking an object required acquiring

two locks: first on the hash table and then on the object itself. Performance suffered both due to long path length and to contention on the hash table lock.

So, for good performance, either lock nursery access must be infrequent, or we must address inherent inefficiencies in hash-based locking. First, is the lock prediction heuristic sufficiently accurate that we rarely result to the lock nursery? If so, a simple lock nursery implementation suffices, since performance will not be critical. Our measurements in Section 5 show that is indeed the case for the benchmarks considered.

However, even if applications arise for which synchronized blocks turn out to be a performance bottleneck, we present two techniques that can address the problem.

In a copying collector, we can *evacuate* an object from the lock nursery when it moves, reformatting the object header to allocate an inline lock word in the object. Especially in a generational collector, this technique will convert long-lived locks to thin locks in a relatively short time period. The only vulnerability of this approach is to programs that invoke a large number of synchronized blocks on a large number of short-lived objects. This seems unlikely.

In a non-copying collector, if the simple lock nursery does not perform well enough, we can employ a more sophisticated implementation. Essentially, we define each slot in the hash table as holding either a surrogate thin lock for a single object hashed to that slot, or in the case of collisions, as a pointer to a list of lock objects, one for each object that maps to that hash slot.

When no object in the lock nursery maps to a particular slot, then that entry holds 0. When a single object is in the thin-locked state, the hash table slot contains a triple

$$(threadId, objectId, lockCount).$$

When *lockCount* $\neq 0$, the system can recover the object pointer from this information by computing

$$(objectId \times tableSize) + slotNumber.$$

When *lockCount* $= 0$, the rest of the word contains a pointer to (or index of) a list of fat locks for that hash code.

Note that since the majority of objects should be synchronized via the in-object thin locks allocated by the static lock prediction heuristic, the hash table will be far less dense than previous hash table based approaches to synchronization for Java.

3.3 Default Hash Code

For non-moving collectors such as mark-and-sweep, the system can define the default hash code of an object to be its address, or more generally, a function thereof.

For moving collectors, we can use the tri-state encoding technique from Bacon et al. [7] (also developed independently by Agesen and used in the Sun EVM), where the states of an object are *unhashed*, *hashed*, and *hashed-and-moved*. For the first two states, the hash code of the object is the address. When the collector moves an object whose state is *hashed*, it changes its state to *hashed-and-moved* and copies the old address to the end of the new version of the object.

If space is available, the system can encode hash code state in two bits in the object header.

If space is not available in the header, we can encode the state by using different TIB pointers which point to blocks that have different hashCode() method implementations. The additional TIB blocks are generated on demand, since most classes are never hashed. Therefore, the space cost for the extra TIB blocks should be minimal.

Note that if fast class equality tests are implemented by comparing TIB pointers, the class equality test must be modified so that if the TIBs do not match then the class pointers are compared before a false result is returned.

3.4 Garbage Collector State

Usually, garbage collectors require the ability to mark objects in one or more ways. For instance, one bit may indicate that the object has been reached from the roots by the garbage collector, and another bit may indicate that the object has been placed in a write buffer by a generational collector (as a way of avoiding duplicate entries).

Because the amount and type of garbage collector state depends heavily on the collector implementation, for this paper, we assume that the collector requires two bits of state information for each object, and provide those bits in all but one of the object models that we implement. We consider the one exception of a garbage collector that uses an alternative method to represent the marks. There are many well-known alternatives, particularly bit maps, that often benefit locality in addition to removing state from the object header [16].

3.5 Forwarding Pointers in Copying Collectors

In a copying collector, it is generally necessary for the object in from-space to temporarily contain a forwarding pointer to its replica in to-space. In object models with multi-word headers, the forwarding pointer is usually placed in the word that normally contains the lock state and hash code. However, when a single-word object model is used, there is generally only one choice: to use the space normally occupied by the TIB pointer.

As a result, the type of the object is not directly available during collection and must be obtained by following the forwarding pointer. This is generally not a problem if the run-time system is written in another language, like C or C++. However, if the run-time system is itself written in Java, as is the case for Jikes RVM, then the presence of forwarding pointers complicates matters considerably.

In particular, an object that is being used by one processor to perform the task of collection may be forwarded by another processor which copies it into to-space. As a result, the TIB of the first processor's object will become a forwarding pointer and virtual method dispatch and other run-time operations will fail.

There are a number of ways to address this problem. The simplest is to always check whether the word contains a forwarding pointer, but this is prohibitively expensive. We chose instead to mark those classes used by the garbage collector, and generate the checks only for methods in those classes and only in object models that require it (namely copying collectors with one-word headers). The result is a slight slow-down in operations on some virtual machine classes, but unimpeded performance for user classes.

3.6 Atomicity

The previous techniques all describe methods to pack sub-word fields into a single word. However, in a multi-processor system, the techniques must also consider atomicity of access to these fields. Mainstream CPUs provide atomic memory access at the word or sometimes byte granularity; they do not provide atomic access at the granularity of a single bit. Thus, if we pack two logically distinct fields into the same byte or word, we must be sure to guard updates to the individual fields to ensure that concurrent writes are not lost.

Note that if the TIB field remains constant after object initialization, stealing bits from the TIB field does not cause a race condition. The other fields may be subject to concurrent writes, and so require care in their layout and access.

4 The Object Models

In this section we describe the various object models implemented for this paper. Figure 1 illustrates the various models.[4] For all object models, arrays include an extra header word which contains the length of the array.

All object models we implemented define the hash code to be an object's address, shifted right by two. For the copying semispace collector, we use the tri-state hashcode encoding described in Section 3.3.

4.1 Two-word Header Object Model

Each scalar object in this object model has a two-word header. The first word contains the TIB pointer, and the second word contains all the other per-object state: lock state (24 bits) and garbage collection state (2 bits). Additionally, for the semispace collector, the second word holds two bits for the tri-state hash code information. The remaining bits are unused.

The advantages of the two-word header object model are that it is relatively simple, allows direct access to the TIB with a single *load* instruction, and only requires two words of overhead for non-arrays and three words of overhead for arrays.

If a copying collector changes GC bits during mutation, this model faces an atomicity issue since the GC state bits and the hash code bits reside in the same byte. The Jikes RVM generational collectors face this problem, since the write barriers may mutate the GC bit for objects in mature space. One option would be to always update the hash bits and GC state bits with atomic operations; but this solution hampers allocators which must set the GC state inside of a write barrier.

[4] The diagrams show the data as always following the header, but in fact in the Jikes RVM the data precedes the header for non-arrays and follows the header for arrays. This allows trap-based null checking and optimized array access [6].

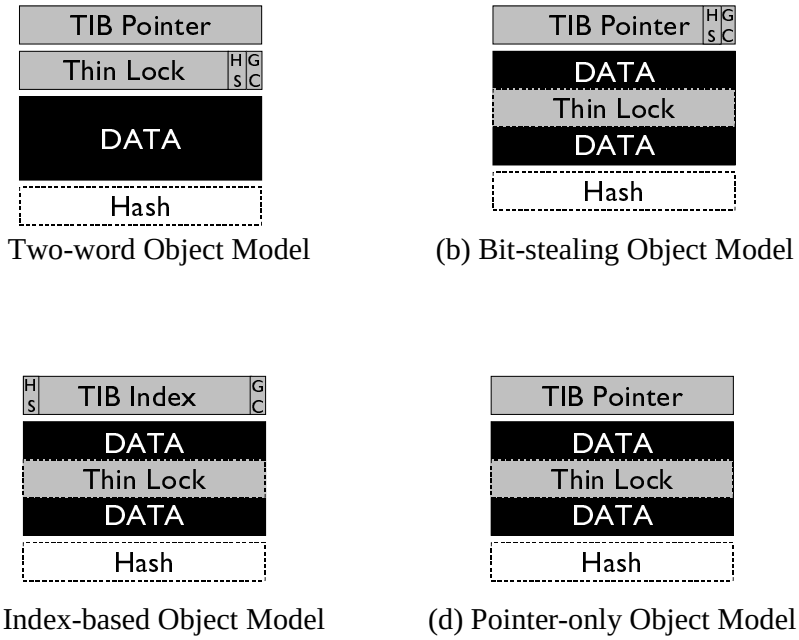

Fig. 1. Object models compared in this paper. Fields in dashed lines are optional and usually absent, including the thin lock in models (b), (c), and (d), in which it is treated as an instance variable. GC and HS are two-bit fields encoding garbage collection and hash code state.

4.2 Single-Word Header Object Models

The remaining object models all use the techniques described in Section 3.2 to eliminate synchronization state from objects that do not have synchronized methods. If the program invokes synchronized blocks upon these objects, the synchronization proceeds via a lock nursery.

These object models also use a combination of compression techniques to remove the need for a second header word to hold the hash code and garbage collection state. Therefore, the per-object overhead is two words for arrays and objects with synchronized methods, and one word for all other objects. Since most objects have no synchronized methods, this amounts to a savings of almost one word for every allocated object.

Bit-stealing Object Model The bit-stealing object model steals the low-order two bits from the TIB pointer and uses them for the garbage collection state. These bits must be masked from the word before it can be used as a TIB pointer.

In addition, for the copying collectors, we need to encode the hash code state. In our implementations, we did this by aligning all TIBs on 16-byte boundaries, making the 4

low-order bits of TIB pointers available to the object model. Since TIBs are of moderate size and occur infrequently relative to instance objects, the space cost is negligible.

Note that for a generational copying collector, if the hash code and GC bits reside in the same byte, this model faces the same atomicity issue between GC bits and hash code state bits as described in Section 4.1. However, if the high-order bit of the TIB address is guaranteed to be fixed (e.g. 0), and the TIB address is fixed, then the system can update this high-order bit with non-atomic byte operations, without introducing a race condition with the hash bits. The current Jikes RVM admits this solution by allocating all TIBs in a non-moving space in a low memory segment.

Indexed Object Model The indexed object model uses a TIB index instead of a TIB pointer, requiring an extra *load* instruction to obtain the TIB pointer indirectly.

The advantages of this object model are that more state can be packed into the header word. Furthermore, in our implementation for the generational copying collector, we use the *high-order* bits for hash code state and the *low-order* bits for GC state. Since the two logical fields do not inhabit the same byte of memory, there are no issues with atomic updates during mutation.

Since the TIB index does not change, the JIT compiler could fold the TIB index as a compile-time constant, further improving the speed of dynamic type tests.

TIB Pointer Only The TIB-only object model provides a minimalist object model that applies when both the GC and hash code state can be eliminated from the object. An example is a mark-and-sweep collector that uses side-arrays of bits to mark objects, rather than marking the objects themselves.

Since a mark-and-sweep collector does not move objects, it can use a function of the object address as the hash code, eliminating the need for the hash code state.

The Jikes RVM includes just such a collector; in fact it uses a side array of bytes rather than bits; this allows parallelization of the marking phase without the need for atomic operations to update the mark bits (since the target architecture is byte-atomic). In this case, the increased parallel garbage collector performance justified the cost of an extra byte per object.

5 Measurements

To evaluate the different object models, we implemented them in a common modular framework in the Jikes RVM version 2.0.4 [15]. All measurements are for the Jikes JIT compiler at optimization level 2, and are run in a non-adaptive configuration to remove non-deterministic effects due to the adaptive sampling which drives re-compilation. Other than the object model, all other facets of the implementation and the machine environment were held constant.

All performance results reported below were obtained on an IBM RS/6000 Enterprise Server model F80 running AIX[5] 4.3.3. The machine has 4GB of main memory and six 500MHz PowerPC RS64 III processors, each of which has a 4MB L2 cache.

[5] **AIX**™ is a trademark or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

Program	Description	Applic. Size	Allocation (MB)	Avg. Obj. Size	Lock Operations
jBYTEmark	Synthetic benchmark	71 KB	35.5	36.2	336,933
CaffeineMark	Synthetic benchmark	33 KB	966.1	307.0	1,606,773
201.compress	Compression	18 KB	105.7	2,973.3	23,856
202.jess	Expert system shell	11 KB	231.7	34.5	4,811,737
209.db	Database	10 KB	62.5	24.2	45,236,755
213.javac	Bytecode compiler	688 KB	198.6	31.7	15,109,145
222.mpegaudio	MPEG coder/decoder	120 KB	1.1	34.6	17936
227.mtrt	Multithreaded raytracer	57 KB	80.4	22.4	1,357,819
228.jack	Parser generator	131 KB	279.8	39.4	9,672,868
SPECjbb2000	TPC-C style workload	821 KB	18,514.3	26.4	1,376,606,250

Table 1. The benchmarks used to evaluate the object models. The benchmarks include the complete SPECjvm98 suite and represent a wide range of application characteristics. The average object size reported assumes a two-word header for scalars and a three-word header for arrays.

Table 1 lists the benchmarks used to evaluate the various object models. They include the full SPECjvm98 [25] benchmark suite, the `SPECjbb2000` [26] benchmark which is commonly used to measure multi-processor transactional workloads, and two commonly cited synthetic benchmarks. The results reported do *not* conform to the official SPEC run rules, so our results do not directly or indirectly represent a SPEC metric. Note that the average object size ranges between 22 and 36 bytes, except for two codes that mostly create large arrays (`compress` and `CaffeineMark`).

With the exception of `SPECjbb2000`, the lock, space usage, and hash code statistics reported below were gathered using a measurement harness that ran the benchmark multiple times, clearing the counters after each run. This enables us to remove distortions caused by Jikes RVM during the first run (in particular optimizing compilation) and accurately report statistics for only the benchmark itself. Since `SPECjbb2000` ran for over 30 minutes, the startup effects of compilation should not significantly impact the overall statistics. Since the Jikes RVM itself is implemented in Java, the statistics do include locks, allocations, and hash code operations performed by the JVM itself on behalf of application code.

5.1 Use of Lock Nursery

Figure 2 shows the distribution of lock operations, using a lock nursery for synchronized blocks that operate on objects that have no synchronized methods. The figure reports the percentage of dynamic lock operations by type: nursery lock, fat (contended) lock, slow thin lock (recursive, no contention), and fast thin lock (non-recursive, no contention).

Nursery locks account for about 0.5% of all lock operations on `SPECjbb2000`. For all other benchmarks, the number of nursery locks is so small as to be invisible on the graph. This indicates that the use of a lock nursery should have a negligible impact on performance.

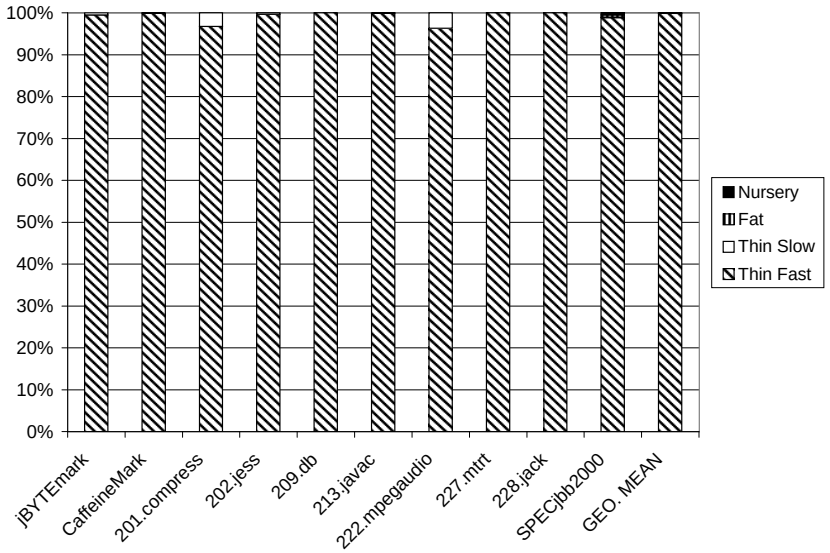

Fig. 2. Lock Usage. The lock nursery accounts for 0.5% of lock operations in `SPECjbb2000` and is insignificant for all other programs, indicating that performance impact will be minimal.

Based on these results, we relied on an exceedingly simple lock nursery implementation: a hash table guarded by a single global lock.

The other statistics for locking operations are consistent with those reported in our and other related work on fast locking: more than 95% of all lock operations are handled by the inlined fast path; virtually all of the remaining cases are handled by the slow path for thin locks that handles nested locking. Contention (and the correspondent use of fat locks) is negligible.

5.2 Space Savings

Figure 3 shows how well two single-word object models reduce space consumption. The first bar shows the percentage of allocated objects whose size we have reduced by a word: arrays and objects without synchronized methods. Clearly, the simple heuristic effectively saves header space. Although we almost never resort to the lock nursery, we have eliminated the space for the thin lock from 97.5% of all objects in the system.

The second bar reports the space savings, as a percentage of the total number of bytes allocated. The one-word object models reduce mean allocated space by about 7%, although on some benchmarks the savings was as high as 21%. Excluding the

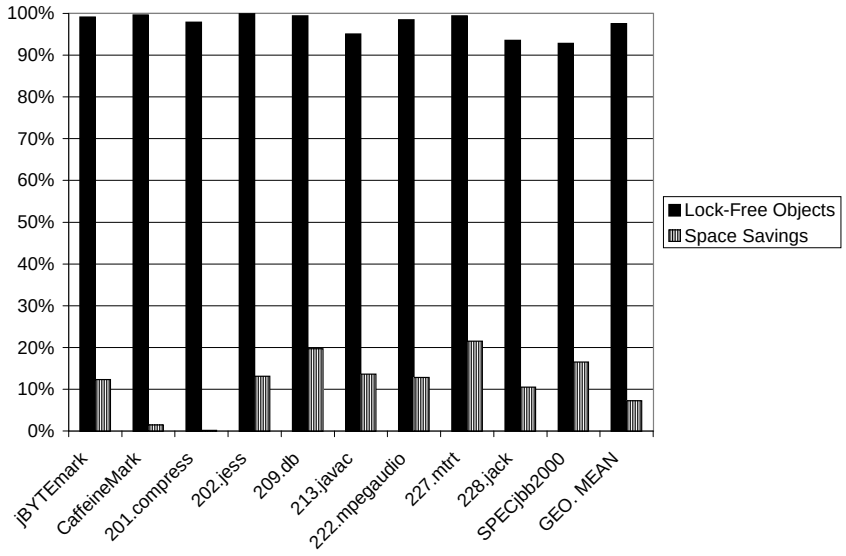

Fig. 3. Object compaction and its effects. Overall 97.5% of objects can be allocated without a lock word; the result is a mean space savings of about 7%. Excluding the two codes that mainly create large arrays, the mean space savings is 14.6%.

two array-based codes, `compress` and `CaffeineMark`, the (geometric) mean space savings is 14.6%; we believe this figure more accurately represents the expected savings from codes written in an object-oriented style.

5.3 Hash Codes

In a copying collector where the hash code can not simply be a function of the object's address, we use the two-bit encoding described in Section 3.3. Table 2 shows the effectiveness of this hash code compression technique for our benchmarks in a semispace copying collector.

First of all, each of the benchmarks we measured exercises the default hashCode() method very rarely: no code takes the default hash code on more than 1.3% of all objects, and the percentage is only that high in benchmarks that perform very little object allocation, while the virtual machine performs 469 hashCode() operations internally.

Of these codes, only `SPECjbb2000` incurred garbage collection during the measurement runs. The copying collector moved only $9,736$ (0.0011%) of the hashed objects on `SPECjbb2000`. This indicates that at least for `SPECjbb2000`, the extra

Program	Objects			hashCode()
	Allocated	Hashed		Operations
jBYTEmark	1,184,687	488	0.041%	2,636
CaffeineMark	3,483,406	469	0.013%	2,515
201.compress	37,419	469	1.253%	2,515
202.jess	7,959,679	469	0.006%	2,515
209.db	3,247,129	469	0.014%	2,515
213.javac	7,517,852	9,961	0.132%	1,961,835
222.mpegaudio	37,018	469	1.267%	2,515
227.mtrt	4,565,881	0	0%	0
228.jack	8,225,623	469	0.006%	2,515
SPECjbb2000	852,907,645	1,053,377	0.124%	21,363,400

Table 2. Hash Code Compression. Of all objects allocated, less than 1.3% ever have their default hash codes taken.

space overhead incurred by *hashed-and-moved* objects is negligible. Of course, these numbers would vary depending on heap size and any generational collection strategy.

These measurements provide some useful insights. These benchmarks use default hash codes rarely, indicating that the virtual machine should place a premium on a compact hashcode representation and a simple implementation. While the two-bit encoding scheme we measured works well, in some cases the extra two bits may not be available in the object header; in this case, it may be better to represent the hash code state with multiple TIBs as described in Section 3.3. Unfortunately this strategy would somewhat complicate other portions of the virtual machine.

For future language designs, it might be better to omit hashing as a feature of all objects in the system, and instead require an explicit declaration. Then hash codes could be treated like instance fields, as we do for lock words.

5.4 Run-time Performance: Non-moving Collector

Figure 4 shows the performance of the various object models, relative to the two-word header object model. These measurements are all taken in a non-moving mark-and-sweep collector.

Figure 4 shows that none of the single-word header object models introduces more than a 8% slowdown on any benchmark, as compared to the two-word header object model. However, we did observe slowdowns of 4% and 12% for the Bit-stealing and Index object models, respectively, on the `Method` component of the `CaffeineMark` micro-benchmark. This component represents a worst-case measurement of object-model induced virtual dispatch overhead. However, the overall performance results indicate that between compiler optimizations that eliminate redundant TIB loads and instruction-level parallelism in the hardware that can perform the extra ALU operations, packing the TIB pointer in with other information only has a minor performance impact on normal workloads. This goes against the common folklore of object-oriented run-time systems design.

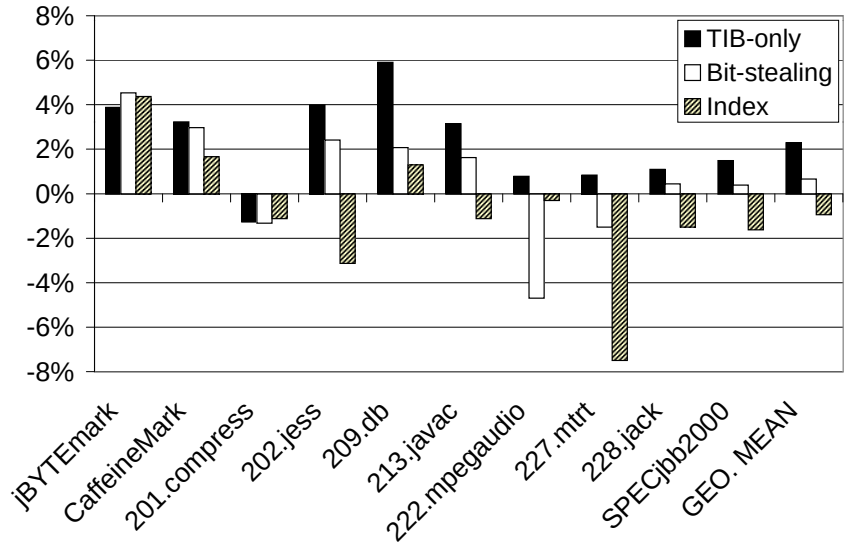

Fig. 4. Speedup obtained by the one-word object models with a non-copying mark-and-sweep collector, relative to the standard two-word Jikes RVM object model. Measurements are for optimized code.

Secondly, across these benchmarks, one-word headers improve performance more often than they degrade performance. This indicates that better cache locality and less frequent garbage collection can have a significant positive performance impact. The `jBYTEmark`, `202.jess`, and `209.db` benchmarks show significant speedups (over 4%) with the TIB-only object model.

In some sections of the micro-benchmarks, we saw large performance increases (23% on `Float` in `CaffeineMark` and 14% on `LU Decomposition` in `jBYTEmark` for TIB-only). This probably results from increased cache locality and more effective use of memory bandwidth for codes that make heavy use of small objects.

Krall and Tomsich [18] have argued that the header size of Java objects should not impose a significant performance penalty on Java floating point codes, since they will mostly use arrays. However, our measurements indicate that in some cases exactly the opposite holds: header size matters most for floating point codes that manipulate large numbers of small arrays or objects (such as complex numbers).

Overall, the TIB-only object model leads to a mean speedup of 2.3% (not surprising since it has the benefit of small headers but pays no extra cost for TIB lookup), while the bit-stealing object model achieves a mean speedup of 0.6%. Taken together with the

space savings, these object models are a clear improvement over the baseline two-word object header.

The indexed object model suffers a 0.9% mean run-time slowdown, and the space saving may justify this implementation choice in some scenarios. However, on codes where the TIB field is frequently accessed, the indexed object model pays a much higher price than the bit-stealing object model.

For `mtrt`, where the indexed model suffers almost 8% slowdown, the TIB accesses are primarily due to guarded inlining performed with the method test [9]; on IA32 Jikes RVM inlines these methods with a code patching guard [14] and thus does not access the TIB. Alternatively, the inlining could be performed with a type test which simply compared the TIB indexes instead of a method test, which would also eliminate the additional cost.

The 1.2% slowdown on `201.compress` for all one-word object models and the 4.7% slowdown on `222.mpegaudio` for the bit-stealing object model are due to cache effects.

Even though it makes non-trivial use of the lock nursery (0.5% of all locking operations), the performance of `SPECjbb2000` still improves with the TIB-only and bit-stealing object models. The space saved by the one word object models enables a 13% decrease in the number of garbage collections.

5.5 Run-time Performance: Copying Collector

Figure 5 shows the performance of the Bit-stealing and Index object models, relative to the object model with a two-word header. These measurements are all taken in a non-generational semispace copying collector.

We did not implement the TIB-only object model in the copying collector because the copying collector does not naturally have mark arrays external to the objects, although this could certainly be done.

Overall, the performance trends are similar to those obtained with the mark-and-sweep collector. The bit-stealing object model improved mean performance by 0.4%, while the indexed object model degraded performance by 1.6%. The difference in relative performance, when compared to the same object models under the non-moving collector, may be due to the cost of checking for forwarding pointers on TIB access within the virtual machine run-time system classes, or may simply be the result of variations in locality and similar effects.

The performance inversions on `201.compress` and `222.mpegaudio` observed with the mark-and-sweep collector did not occur with the semispace collector, but `227.mtrt` once again payed a high cost for TIB lookup with guarded inlining.

6 Related Work

To our knowledge this work is the first comprehensive study to explore alternative implementations of the Java object model. Previous work has been implicit in work on other parts of the system, primarily locking and garbage collection.

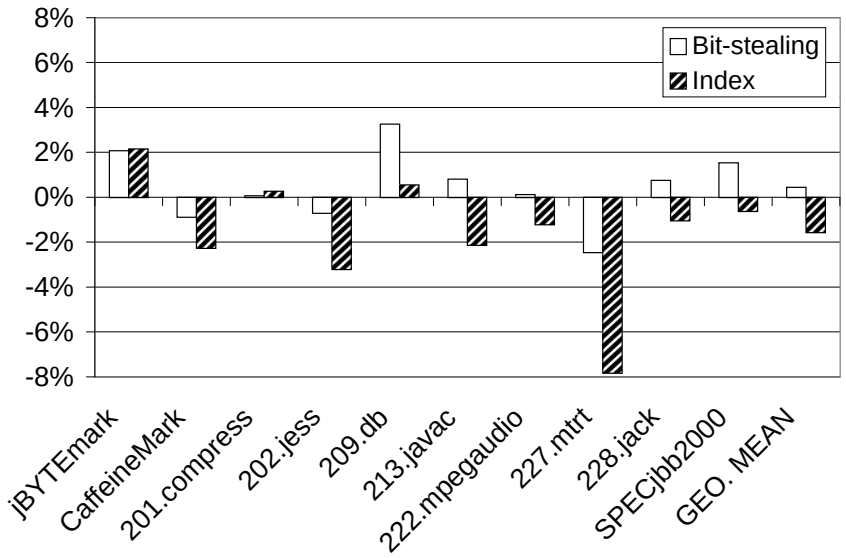

Fig. 5. Speedup obtained by the one-word object models, with a semi-space copying collector, relative to the standard two-word Jikes RVM object model. Measurements are for optimized code.

Most closely related to this paper is the work of Shuf et al. [22], in which they studied the opportunities for space savings made possible by the classification of certain types as *prolific*. Prolific types are assigned a short (4-bit) type index, which is stored in a compressed header word along with lock, hash, and garbage collection state. If the type index is 0, an extra header word is the full 32-bit TIB pointer. They presented measurements of potential space savings that agree with ours, but did not implement the scheme.

One way of understanding the difference between our work and that of Shuf et al. is that they are usually compressing the TIB pointer, while we are usually eliminating the thin lock word. The main disadvantage of their scheme is that if the number of prolific types exceeds 16, performance in both space and time will be adversely affected.

Dieckmann and Hölzle [10] performed a detailed study of the allocation behavior of the SPECjvm98 benchmarks. Their measurements were all in terms of an idealized, abstract object model in which each instance object had a class pointer and each array object had a class pointer and a length — note that this results in virtually the same object space overheads as we achieve in our actual implementation. They found that

instance objects have a median size of 12-24 bytes, and char arrays have a median size of 10-72 bytes, indicating that most space savings will come from these object types.

The initial Java virtual machine from Sun used *handles* (a level of indirection) to avoid the need to recompute object references in the copying collector, and used an additional word for the hash code and garbage collection state. Locking was always performed via a hash table with a global lock. The result was an object model which imposed three words of overhead per object (handle, class pointer, and hash code/GC state) and had poor performance for synchronization, which turned out to be ubiquitous in many Java programs. The use of handles led each object to reside in two cache lines, significantly reducing effective cache size.

6.1 Run-time Monitor Optimizations

Krall and Probst [17] were the first to attack the synchronization overhead in a systematic manner in the CACAO JIT compiler. Since their locks were represented as three-word structures, it was unacceptable to place them inline in the object header. Instead, the system accessed the locks via an external hash table, severely limiting the potential speedup.

With *thin locks*, Bacon et al. [7] attacked the synchronization overhead by allocating 24 bits within each object for lock state, and using atomic compare-and-swap operations to handle the most common cases: locking unlocked objects and locking objects already owned by the current thread. In these cases, the lock was in its thin state, and the 24 bits represented the thread identifier of the lock owner and the lock nesting level. In rare cases (deeply nested locking, contention between threads, or use of wait and notify operations), the high bit of the lock state is set to 1, indicating that the lock is a fat lock, and the other 23 bits are an index into a table of fat locks.

To make room for the thin lock, the system employed the two-bit hash code state encoding described in Section 3.3, and the remaining 6 bits of the word were reserved for garbage collection state. The implementation was done in a Sun-derived JVM modified by the IBM Tokyo Research Lab to eliminate handles. The resultant system had an object model with two words of space overhead per object and good lock performance.

Thin locks have the potential drawbacks of excessive space consumption due to lack of deflation and excessive spinning due to busy-waiting on inflation. Subsequent work by Onodera and Kawachiya [20] attempted to ameliorate these problems, at the expense of complicating the locking protocol and requiring an extra bit in the object header in a separate word from the lock. Subsequently, Gagnon and Hendren [13] showed how the extra bit could be stored on a per-thread rather than a per-object basis.

The alternative approach to lightweight Java locking that is most similar to thin locks is that of Yang et al. [28] in the LaTTe virtual machine, which used a 32-bit lock word in each object. Unlike thin locks, the format of the lock word never changes, and includes a count of waiting threads. If the count is non-zero, the queue is found via a hash table.

In the *meta-lock* approach of Agesen et al. [1] in the Sun Exact VM, one 32-bit word in the header normally contains a 25 bit hash code, a five bit age (for generational collection), and a two bit lock (which is 0 in the normal state). When the lock is non-zero, the other 30 bits point to an auxiliary structure, either a lock structure or a thread

structure of the thread that is changing the lock state. In the latter case the object is considered meta-locked, and the per-object lock, hash code, and garbage collection state are temporarily unavailable. Meta-locked objects are always in the process of making short transitions between other states. When the object is locked, the auxiliary structure contains the relocated hash code and garbage collection state.

Fitzgerald et al. [12] describe the Marmot static compiler for Java, which uses a two word object header that includes a class pointer and a word containing synchronization state and hash code.

All of these various techniques are compatible with the object compression techniques described in this paper, which could be used to generate single-word object models that use different synchronization techniques.

6.2 Compile-time Monitor Optimizations

There has also been significant work on elimination of synchronization via compile-time analysis [2, 3, 8, 21]. This work can in some cases be highly effective at reducing or (in the case of single-threaded programs) eliminating synchronization operations. Therefore these optimizations are complementary to the run-time techniques described above.

While none of the cited work investigated the potential for reducing object size, this could certainly be done, complementing the work presented in this paper.

6.3 Other Languages

Previous work [23, 19] describes object models for C++, primarily addressing complications due to multiple inheritance. C++ does not support synchronization, hashing, and garbage collection at the language level.

Tip and Sweeney [27] describe how C++ class hierarchy specialization can improve the efficiency of object layout. In a subsequent study [24], they show that a significant number of data members are never actually used at run-time, and can be identified and removed at compile-time.

7 Conclusions

We have shown that a fundamental piece of folklore about the design of object-oriented run-time systems is wrong: when engineered carefully in conjunction with a high-quality optimizing JIT compiler, encoding the method table pointer does not lead to any significant performance penalty. Our techniques do not depend on any particular implementation of lightweight synchronization or garbage collection, and so long as a high-quality compiler is used, should apply to other object-oriented languages.

Experimental results show that header compression can yield significant space savings while simultaneously either improving run-time performance by a mean of 2.3% or at worst degrading it by 1.6% (depending on the details of the implemented object model). Programs that use large numbers of small objects show significant performance

improvement. The availability of space-efficient Java run-time systems should encourage programmers to more freely apply object-oriented abstraction to small objects.

We have also shown that treating the lock as an instance field, implicitly defined by synchronized methods, is a highly effective way of eliminating the space overhead of in-object locks, while retaining their performance benefits in 99% of the cases.

As an added benefit, we have shown how to engineer a high performance virtual machine with a pluggable, parameterized object model, without any loss in run-time efficiency. This feature allows an implementer to substitute various object models according to their fit with the rest of the system design, especially the garbage collector. The object model plug-ins are available via the IBM Jikes RVM open-source release to encourage future experimental studies of object model variants.

References

[1] AGESEN, O., DETLEFS, D., GARTHWAITE, A., KNIPPEL, R., RAMAKRISHNA, Y. S., AND WHITE, D. An efficient meta-lock for implementing ubiquitous synchronization. In *OOPSLA'99 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications* (Denver, Colorado, Oct. 1999). *SIGPLAN Notices, 34*, 10, 207–222.

[2] ALDRICH, J., CHAMBERS, C., SIRER, E. G., AND EGGERS, S. J. Static analyses for eliminating unnecessary synchronization from Java programs. In *Static Analysis: Sixth International Symposium* (Venice, Italy, Sept. 1999), A. Cortesi and G. Filé, Eds., vol. 1694 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 19–38.

[3] ALDRICH, J., SIRER, E. G., CHAMBERS, C., AND EGGERS, S. Comprehensive synchronization elimination for Java. Tech. Rep. UW-CSE-00-10-01, Department of Computer Science, University of Washington, 2000.

[4] ALPERN, B., COCCHI, A., FINK, S., GROVE, D., AND LIEBER, D. Efficient implementation of Java interfaces: invokeinterface considered harmless. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Tampa, Florida, Oct. 2001). *SIGPLAN Notices, 36*, 10, 108–124.

[5] ALPERN, B., COCCHI, A., AND GROVE, D. Dynamic type checking in Jalapeño. In *Proceedings of the Java Virtual Machine Research and Technology Symposium* (Monterey, California, Apr. 2001), pp. 41–52.

[6] ALPERN, B., ET AL. The Jalapeño virtual machine. *IBM Syst. J. 39*, 1 (2000), 211–238.

[7] BACON, D. F., KONURU, R., MURTHY, C., AND SERRANO, M. Thin locks: Featherweight synchronization for Java. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation* (Montreal, Canada, June 1998). *SIGPLAN Notices, 33*, 6, 258–268.

[8] BOGDA, J., AND HÖLZLE, U. Removing unnecessary synchronization in Java. In *Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications* (Denver, Colorado, Oct. 1999). *SIGPLAN Notices, 34*, 10, 35–46.

[9] DETLEFS, D., AND AGESEN, O. Inlining of virtual methods. In *Thirteenth European Conference on Object-Oriented Programming* (1999), vol. 1628 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 258–278.

[10] DIECKMANN, S., AND HÖLZLE, U. A study of the allocation behavior of the SPECjvm98 Java benchmarks. In *Proceedings of the Thirteenth European Conference on Object-Oriented Programming* (Lisbon, Portugal, 1999), R. Guerraoui, Ed., vol. 1628 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 92–115.

[11] DYBVIG, R. K., EBY, D., AND BRUGGEMAN, C. Don't stop the BIBOP: Flexible and efficient storage management for dynamically-typed languages. Tech. Rep. 400, Indiana University Computer Science Department, 1994.

[12] FITZGERALD, R. P., KNOBLOCK, T. B., RUF, E., STEENSGAARD, B., AND TARDITI, D. Marmot: an optimizing compiler for Java. *Software – Practice and Experience 30*, 3 (2000), 199–232.

[13] GAGNON, E., AND HENDREN, L. SableVM: A research framework for the efficient execution of Java bytecode. In *Proceedings of the Java Virtual Machine Research and Technology Symposium* (Monterey, California, Apr. 2001), pp. 27–40.

[14] ISHIZAKI, K., KAWAHITO, M., YASUE, T., KOMATSU, H., AND NAKATANI, T. A study of devirtualization techniques for a Java Just-In-Time compiler. In *OOPSLA'2000 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications* (Tampa, Florida, Oct. 2000). *SIGPLAN Notices, 35*, 10, 294–310.

[15] Jikes RVM 2.0.4. `http://www.ibm.com/developerworks/oss/jikesrvm`.

[16] JONES, R. E., AND LINS, R. D. *Garbage Collection*. John Wiley and Sons, 1996.

[17] KRALL, A., AND PROBST, M. Monitors and exceptions: how to implement Java efficiently. *Concurrency: Practice and Experience 10*, 11–13 (1998), 837–850.

[18] KRALL, A., AND TOMSICH, P. Java for large-scale scientific computations? In *Proceedings of the Third International Conference on Large-Scale Scientific Computing* (Sozopol, Bulgaria, June 2001), S. Margenov, J. Wasniewski, and P. Y. Yalamov, Eds., vol. 2179 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 228–235.

[19] MYERS, A. C. Bidirectional object layout for separate compilation. In *OOPSLA'95 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications* (Oct. 1995). *SIGPLAN Notices, 30*, 10, 124–139.

[20] ONODERA, T., AND KAWACHIYA, K. A study of locking objects with bimodal fields. In *OOPSLA'99 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications* (Denver, Colorado, Oct. 1999). *SIGPLAN Notices, 34*, 10, 223–237.

[21] RUF, E. Effective synchronization removal for Java. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, June 2000). *SIGPLAN Notices, 35*, 5, 208–218.

[22] SHUF, Y., GUPTA, M., BORDAWEKAR, R., AND SINGH, J. P. Exploiting prolific types for memory management and optimizations. In *Conference Record of the ACM Conference on Principles of Programming Languages* (Portland, Oregon, Jan. 2002), pp. 295–306.

[23] STROUSTROUP, B. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990. Chapter 10, section 10.

[24] SWEENEY, P. F., AND TIP, F. A study of dead data members in C++ applications. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation* (Montreal, Canada, June 1998). *SIGPLAN Notices, 33*, 6, 324–332.

[25] THE STANDARD PERFORMANCE EVALUATION CORPORATION. SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98, 1998.

[26] THE STANDARD PERFORMANCE EVALUATION CORPORATION. SPEC JBB 2000 Benchmark. http://www.spec.org/osg/jbb2000, 2000.

[27] TIP, F., AND SWEENEY, P. F. Class hierarchy specialization. In *OOPSLA'97 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications* (Denver, Colorado, Oct. 1997). *SIGPLAN Notices, 32*, 10, 271–285.

[28] YANG, B.-S., LEE, J., PARK, J., MOON, S.-M., AND EBCIOĞLU, K. Lightweight monitor in Java virtual machine. In *Proceedings of the Third Workshop on Interaction between Compilers and Computer Architectures* (San Jose, California, Oct. 1998). *SIGARCH Computer Architecture News, 21*, 1, 35–38.