Proceedings of the Java™ Virtual Machine Research and Technology Symposium (JVM '01)

Monterey, California, USA
April 23–24, 2001

Mostly Accurate Stack Scanning

Katherine Barabash
IBM Haifa Research Laboratory
kathy@il.ibm.com

Niv Buchbinder
IBM Haifa Research Laboratory
nivb@il.ibm.com

Tamar Domani
IBM Haifa Research Laboratory
tamar@il.ibm.com

Elliot K. Kolodner
IBM Haifa Research Laboratory
kolodner@il.ibm.com

Yoav Ossia
IBM Haifa Research Laboratory
yossia@il.ibm.com

Shlomit S. Pinter
IBM Haifa Research Laboratory
shlomit@il.ibm.com

Janice Shepherd
IBM T.J. Watson Research Laboratory
janshep@us.ibm.com

Ron Sivan
IBM Haifa Research Laboratory
rsivan@il.ibm.com

Victor Umansky
IBM Haifa Research Laboratory
victoru@il.ibm.com

Abstract: The Java Virtual Machine (Jvm) needs, for the purpose of garbage collection (GC), to determine the data type stored in every memory location. Jvms that can do this reliably are said to be *type-accurate* (TA). Full type-accuracy usually exacts a price in performance due to the need to scan stacks and registers accurately. The mostly accurate approach presented here can reduce the TA overhead significantly by sacrificing accuracy for the small minority of memory locations that add the most to the cost. Performance results show that mostly accurate stack scanning performs as well as conservative stack scanning and that relatively few objects are identified conservatively.
In addition our implementation is designed to support and generate type maps for any verifiable bytecode stream (including combinations that are unlikely to be produced by a compiler) without requiring rewriting of the bytecode. We introduce a new compression technique for type maps that uses a program-friendly format for the maps; yet, achieves good compression and provides fast opening of compressed maps. We show how to apply systematic testing techniques and test coverage tools to an accurate stack scanner.

Keywords: Jvm, type accuracy, garbage collection.

1. INTRODUCTION

For Java Virtual Machines (Jvms) the unequivocal identification of references at run time may be difficult. This is mainly a problem for the task of Garbage Collection (GC), mandated by the Jvm Specification [29], which needs to make such determinations.

In a commonly used solution, known as conservative GC [10], any value that could possibly signify a reference is treated as one; this includes both values in objects and values on the stack. While avoiding the danger of ever missing a reference, conservative GC runs the risk of considering values that are actually integers or irrelevant, thus retaining garbage. Moreover, this approach is incompatible with schemes that move objects, since when a value used as a reference is not guaranteed to be one, updating it (as would be necessary if the object it points to is relocated) could have disastrous effect if in fact the value was not a reference.

Another approach, often described as conservative with respect to the roots [8], scans the stacks and registers conservatively, but uses type information associated with the objects to scan them accurately. This approach is used widely in Java Virtual Machines, where the run time is implemented in C, e.g., the Sun Classic JVM [12] and IBM's Developer Kit [25]. In these Jvms scanning the stacks accurately is difficult, as C compilers do not typically produce type information for the stack frames they generate. However, type information is readily available for object slots. The conservative with respect to the roots approach, while retaining less garbage and allowing object relocation, complicates the use of copying GC [11], compaction, and algorithms such as generational scavenging [27, 40, 32] and Train [24, 36], which rely on copying. Nevertheless, there is a range of GC techniques available that do not require that all objects be relocatable, e.g., [8, 9, 33, 15, 13, 16, 35].

A third approach is type-accuracy (TA) [14], where the location of all references, both in the roots (stacks and registers) and objects, can be determined with certainty. TA for stacks and registers is generally obtained by producing type maps for methods, and using these maps to interpret stack frames as they are scanned by GC. This approach is also widely used in Jvm implementations [23, 5, 17, 26]. It is often considered the most technologically advanced of the three approaches since it allows all objects to be relocated and retains the least amount of garbage.

As appealing as TA sounds, it comes with a performance and a complexity price, especially for Java in server environments where applications are highly multithreaded and may make frequent transitions into native code, e.g., to enable access to legacy code. In a multithreaded environment all threads must be stopped for GC at *safe points*, execution points where type information for volatile structures such as stack frames and registers is available. GC safe points are typically placed at invocation sites and on backward branches in loops. Thread stopping is usually done through polling or code patching. Agesen [2] found that polling has a high performance overhead and that code patching is more efficient for single-threaded programs. However, code patching is complex and its performance is not likely to scale in a highly multithreaded environment, where every thread may potentially need to be patched. Stichnot et al. [39] report that the amount of space required to save type information for every compiler generated instruction for Java code can be kept manageable using appropriate compression techniques. This would avoid stopping overhead by allowing a thread to stop at any point in Java code. However, it remains to be shown that this approach performs well, as the compression/decompression algorithms are likely to be time consuming, and that it extends to handle native code where no maps are available.

Transitions from Java methods into native code require the saving of non-volatile registers. This is because compilers for native code, e.g., for C, do not generally keep track of type information or where registers are saved in stack frames. Frequent saving of non-volatile registers may also hurt performance. Dealing with registers may also present other problems. In particular, not all platforms allow the registers of a stopped thread to be updated by a non-privileged thread. This may be required in case registers contain references to objects that have been relocated.

1.1 Our Contribution

We propose a fourth approach, that is mostly accurate with respect to the roots. It is a pragmatic approach that has the potential to deliver a high degree of type accuracy, thereby reducing retained garbage and the number of non-relocatable objects, while incurring little or no performance overhead as compared to conservative stack scanning. It is also significantly less complex than full accuracy, as it does not require complicated schemes for stopping threads or for identifying all frames on the stack.

In principle, mostly accurate stack scanning applies accurate scanning only to frames where it is easy and fast to do so, and scans the remaining frames conservatively. Thus, threads are no longer required to be stopped at GC safe points but may be stopped anywhere, at the price of a conservative scan of the most recent frame on the thread's stack. Most other frames are expected to be at invocation points and therefore remain candidates for accurate scanning. Of those, frames whose type information is complicated to obtain (e.g., native code frames) are also scanned conservatively. Our implementation also does not pay the cost for saving non-volatile registers before transitions into native code. This does not incur an additional loss of accuracy since native code frames are scanned conservatively in any case.

We implemented a mostly accurate scanner, replacing the conservative stack scanner in a prototype version of IBM's Developer Kit for Windows NT that is compatible with Java 2. The GC in this Jvm is a stop-the-world parallel mark and sweep collector, which compacts only when necessary. We compared the performance of the mostly accurate scanner with the conservative scanner. Our results confirm that mostly accurate scanning performs as fast as conservative scanning and reduces the number of objects that are identified conservatively.

We also used several interesting techniques applicable to other implementations of type-accuracy. Our implementation of accurate stack scanning is designed to support and generate type maps for any verifiable bytecode stream (including combinations that are unlikely to be produced by a compiler) without requiring rewriting of the bytecode. It is based on the scheme first used by Jalapeño [6, 5]. In addition, we introduce a new compression technique for type maps that uses a program-friendly format for the maps; yet, achieves good compression and provides fast opening of compressed maps. Finally, we show how to apply systematic testing techniques and test coverage tools to an accurate stack scanner.

1.2 Outline

Section 2 presents our overall approach to scanning stack frames accurately and describes which frames we scan accurately and which frames we scan conservatively. Section 3 discusses map generation for interpreted Java methods. Section 4 describes how we store and compress the type maps, and provides performance results for our compression scheme. Section 5 shows how systematic testing techniques can be applied to accurate stack scanning. Section 6 presents performance results for our prototype implementation of mostly-accurate stack scanning. Finally we conclude in Section 7.

2. MECHANICS OF TYPE ACCURACY

The task of running a Java program can be split in the Jvm into several subtasks that together do the job correctly and efficiently. Type accuracy is achieved by a combination of means and techniques which collectively cover the various subtasks and the many cases within them that need to be handled. First we briefly review the mechanics of TA in general. Then we describe which parts of the stack we scan accurately and which conservatively.

2.1 Type Accuracy

References to objects can occur within other objects or in volatile execution structures: stack frames and machine registers. References within an object are documented by the class of the object. GC uses this information recorded in the class to accurately scan object instances. Comparable descriptions of volatile structures must be generated specifically to support TA. These descriptions often take the form of *type maps* which indicate which stack slots and machine registers contain a reference at a given execution point of a given method.

The Jvm subtasks may be broadly categorized as follows:

- Interpretation of Java bytecode.

- Execution of Just-In-Time compiled Java code (JITted code).
- Execution of non-Java (native) code.
- Jvm services, such as object allocation, class loading, JIT compilation and GC.

Type maps for frames of interpreted bytecode are generated by analyzing the bytecode itself; we describe map generation in detail in Section 3. Type maps for compiled code are generated by the JIT compiler as part of its compilation process. Non-Java methods (native code) are expected to abide the Java Native Interface (JNI, [30, 31]) that closely controls the usage of references. References are generally kept outside the reach of native code, and JNI provides mechanisms for tracking them in the few exceptions where they are not. Finally there are Jvm services, which are routines written in the implementation language of the Jvm, in our case, C.

Scanning of volatile structures is done using the following elements:

1. stack scanner
2. frame traverser
3. type map repository
4. type map generator

GC calls upon the stack scanner to scan the stacks. The stack scanner resorts to the frame traverser to walk the stack one frame at a time and to determine its particulars: which of the subtasks above owns the frame, what method is it executing and where is the execution of the method stopped. Based on the method and the point within its execution, the stack scanner attempts to obtain a type map for the frame from the repository. Type maps are normally available for compiled methods since the JIT compiler deposits maps for every method it compiles. If the method is interpreted and its maps are not yet available in the repository, the stack scanner invokes the map generator to create maps for the method and then saves them in the repository for possible future use. Note that in the case of interpreted code, this arrangement results in maps being calculated only for methods that are active at the time of GC. Once a map is available, the stack scanner can identify the slots containing references and report them to GC.

2.2 Mostly Accurate Stack Scanning

The frame traverser is able to recognize stack frames for Java methods, whether interpreted or JITted. Accordingly, our mostly accurate stack scanner scans all frames belonging to interpreted and JITted methods accurately, except for the most recent frame on the stack of each stopped thread. All other frames, which are just the regions on the stack between groups of Java frames, are scanned conservatively. Thus, the stack scanner does not differentiate between frames belonging to native methods and Jvm services.

3. GENERATING TYPE MAPS

The type map generator for the interpreter analyzes the bytecode of a method and produces maps for any execution point at which GC could occur. Below we provide an overview of the type map generator. We adopted the approach used by Jalapeño [5], which does not require rewriting of bytecode. Then we briefly discuss maps for JITted methods.

Since the use of slots in a stack frame could change during the course of execution of a method, maps depend on the point of execution. Conceivably, the layout of the frame need not be unique even at a specific execution point, but could depend on the history of execution. However, due to the *Gosling property* [19, 4], which holds for Java bytecode, except for `jsr` subroutines (discussed below), and is verified at class load time, the stack frame structure is normally independent of the execution path. This allows the use of a basic algorithm, similar to the one used for verification[29, Section 4.9], consisting of three steps:

1. The bytecode for a method is split into basic blocks.
2. An iterative algorithm computes the type map for the start of each basic block.
3. The maps for the GC points (e.g., invocation points, allocation points) within each basic block are calculated.

3.1 `jsr` Subroutines

The `jsr` and the accompanying `ret` bytecodes are a source of difficulty for generating type maps since their semantics permits a violation of the Gosling property. This bytecode pair is used to declare an inner subroutine within a method, which does not have its own frame. Java compilers use `jsr` subroutines to control the flow from a sequence with many exits into a sequence that must be executed regardless of the exit taken, e.g., a `finally` clause, or the release of the lock at the end of a `synchronized` method. Semantically, `jsr` is an intra-method branch instruction just like the `goto` bytecode, except that it also pushes a return address (the offset of the bytecode following the `jsr`) onto the operand stack. If that address is subsequently stored in some local variable, the `ret` bytecode can be used to jump back to the point from which the subroutine was called.

Various restrictions apply to `jsr` subroutines: they may be entered only via a `jsr` and may not call themselves recursively. However, there is no restriction on the contents of local variables that the subroutine does *not* reference. As a result, the same local could contain a reference at some invocations of the `jsr` subroutine and a non-reference at others (see Figure 1). This will not disturb the execution of the subroutine, but if it is stopped for GC and the types of its frame slots are needed, the state of that local could be indeterminate.

3.2 Solution For `jsr`

A solution, adopted here from the work done on the Jalapeño project [5], postpones the generation of the map until run time, when the actual path the execution has taken is known. The generator prepares two kinds of maps in advance: *final maps* and *change maps*. A final map is a bit array where each bit is associated with a stack frame slot. Slots expected to contain a reference (`ref`) have their bits set. A change map consists of two bit arrays that together represent the

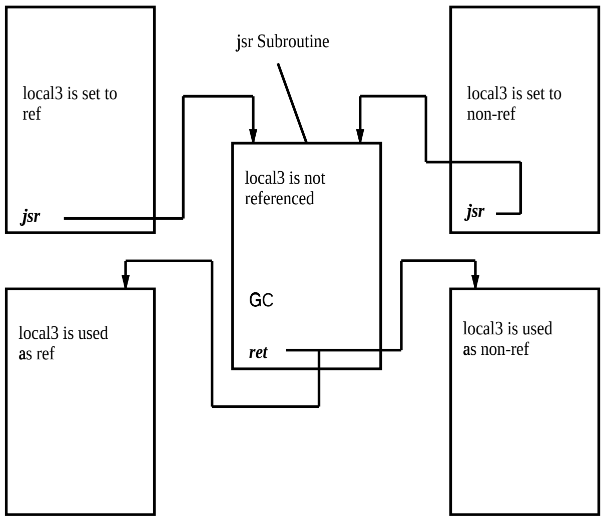

Figure 1: `jsr` subroutine where the Gosling property could be violated: the contents of `local3` at the time of GC may depend on the history of execution.

merge a change map	**with a final map**		**with another change map**		
	non-ref	ref	to-non-ref	to-ref	neither
to-non-ref	non-ref	non-ref	to-non-ref	to-ref	to-non-ref
to-ref	ref	ref	to-non-ref	to-ref	to-ref
neither	non-ref	ref	to-non-ref	to-ref	neither

(a) **(b)**

Table 1: Rules for merging a change map with (a) a final map and (b) with another change map

difference between two states of the stack frame, and hence between two final maps. One array, `to-ref`, indicates the locations that have changed to contain a reference, and the other, `to-non-ref`, indicates changes to a non-reference. A change map C can be *merged* with a final map F to obtain a new final map F', representing the state of a stack frame after applying the change described in C to the state described in F (see Table 1 **(a)**). Two change maps can also be merged, producing a new change map which describes the result of applying them consecutively (see Table 1 **(b)**. Note that this merge operation is not commutative.)

Within `jsr` subroutines the map generator calculates change maps, which reflect the changes from the beginning of the subroutine to the point for which the map was made. In addition, the map contains the location at which the return address provided by the `jsr` is kept. At run time, the Jvm retrieves the return address and uses its actual value to determine which `jsr` actually invoked the present subroutine, and obtains another map from there. If the latter map is a final map, merging the two maps yields a final map which correctly describes the present frame. If the map at the site of the `jsr` is a change map as well, (probably due to nesting of `jsr` subroutines), merging the maps produces yet another change map. The procedure is repeated recursively until a final map is obtained.

Finding the return address is not always simple. Normally, a `jsr` subroutine begins by storing the return address in some local and using that local in the `ret` at the end. However, verifiable code in general need not be this simple, and the return address could at times be rather elusive:

- A subroutine may store the return address in a local, but not at the very beginning of its execution. Consequently, the return address may still be on the operand stack when a map is required.
- The subroutine may `dup` the return address and use the duplicate, rather than the original value, for returning.
- The subroutine may discard the return address altogether, but then it may no longer return, and in effect it is no longer a subroutine.
- In case of nested subroutines, there are two return addresses in effect at the same time. The inner subroutine may swap the two values if they are adjacent on the operand stack. This could confuse the recursion process described above, as the map for the outer subroutine could specify a return address location, which

is not the right one after the swap. This is an esoteric case that was discovered during the course of map testing, which is described in Section 5.

Much of the complexity in implementing the map generator centered on the tracking of the return addresses for `jsr` subroutines.

3.3 Maps for Compiled Methods

Maps are generated for compiled methods as well. Unlike interpreted methods that are produced on demand, maps for compiled methods are created by the JIT compiler as a byproduct of the compilation. The maps are then kept in the repository until they are needed. Producing them on demand would essentially require recompiling the method and would be too costly.

Compiled methods also have to deal with various aspects of the architecture that are shielded by the interpreter. Nonvolatile registers (NVRs) are one such aspect. To abide by the calling convention, a method must save an NVR in its frame before its first use of the NVR and restore the NVR before returning. However, the method saving the NVR cannot know the type of its value. To overcome this problem, each map contains information both about what values its method places in NVRs on the one hand, and which NVRs it saves and where on the other. The stack scanner can intersect content information from one map with storage information from another as it traverses the stack, and so determine which slots used for saving NVRs actually contain references.

Another issue is that of *untidy references*, i.e., references which do not point to the top of objects as canonical, or *tidy*, references do. Untidy references are generated by the compiler when optimizing access to arrays, for example. Each type map contains a list of all such references if any exist in the frame, together with how they are dependent on the corresponding tidy references (see [14]). The stack scanner is again responsible for updating any untidy references whenever any of the tidy ones they depend on get modified in the process of GC. Our current implementation does not handle untidy references yet and instead scans them conservatively.

When compiled methods are concerned, the normal relationship of one frame per each active method may no longer hold. For example, there are cases where additional frames are injected between those of otherwise adjacent methods. This is done when incompatible calling conventions require reformatting of the arguments, or when a method switches from being interpreted to being compiled. These extra frames may also contain references, but for the most part are currently scanned conservatively.

3.4 Related Work

The challenge of generating type maps in the presence of `jsr` was first described by Ageson *et al.* [3, 4]. They solve the problem by splitting the problematic variables in two, one for the reference type and the other for the non-reference type. This requires rewriting of bytecode. Since there are size restrictions on a method's stack and the bytecode, this solution will not always work.

Another solution [1] maintains tags for ambiguous variables that indicate their current type at run time. This requires code to initialize and maintain the tags, and incurs an overhead upon method invocation and with each update of the affected variables.

A third approach [39] is similar to the one used in Jalapeño [5], which was described above. It also makes compile-time information available to the Jvm, which can then apply runtime knowledge to determine the true type of ambiguous variables.

4. REPOSITORY FOR MAPS

Generation of type maps for stack frames raises the need for map management. Although maps could be generated on demand each time one is required, such an approach would be wasteful and would hurt performance. A map, once created, is kept in a repository for possible future use. This section describes the repository and the compression technique it uses.

4.1 Requirements

Maps for interpreted methods are calculated on demand. Maps for JITted methods are produced as a byproduct of the compilation process. Each method may have multiple maps associated with it that are all produced at one time.

Type maps, in addition to indicating which slots of the frame contain references, provide information necessary for proper interpretation of the frame at run time. Some fields of the map fit in fixed sized fields, such as the number of local variables or operand stack entries, while others, such as the list of untidy references in JITted maps, are of variable length. The maps consist of a fixed size header, followed by a variable length data area, whose components and size are derivable from information stored in the header.

When interpreting a frame, only one of the maps of the corresponding method is ever needed. Over time, however, several or even all of the method's maps may be called for, particularly for methods that are called often. On the other hand, methods such as "main", that are called shortly after the run starts and remain active for a long time, may have their frames examined several times, requiring the same map over and over again.

In summary, the characteristics of map usage make the following demands on the repository:

- Maps should be saved for future use, avoiding the overhead of recalculating them. There are maps for all JITted methods and for all interpreted methods for which a map was ever requested. This could add up to a large number of maps, so compression, efficient in both time and space, is required.
- Maps are called for in random order, but only one map per method at a time. This requirement of fast retrieval of randomly selected maps favors compression schemes where each map is compressed separately from the others.
- Maps should not require a lot of bit operations for extracting information. This simplifies the code of the

stack scanner. This means that maps will be wider, but we need a compression scheme in any case.

We expect the maps to have the following characteristics:

1. Methods may have any number of maps, including none.
2. Most of the methods will have only a small number of maps.
3. Maps of the same method should be similar to each other, yet there is no part of all the maps of a method that is guaranteed to be the same.
4. For methods with many maps, the similarity should increase with proximity (within the bytecode stream), i.e., maps of neighboring bytecodes should be more similar than maps associated with bytecodes that are further apart.
5. Most of the maps are not big, less than 50 bytes on average.
6. All of the above holds true for both JITted and interpreted methods.

Examination of actual maps corroborated most of the assumptions above. Assumptions 2 and 4 did not quite hold for interpreted methods in short-lived applications. The reason could be that since maps for interpreted methods are created on demand, longer methods which invoke many additional methods are more likely to be encountered on the stack and therefore over-represented among the maps. Indeed, the failure of these assumptions was less pronounced in longer-running applications, where eventually maps were created for more of the methods.

4.2 Compression Techniques

In order to meet the requirements above, and with consideration of the characteristics of maps, we chose to represent every map as the difference from a specific (fixed) map of the same method. The latter was called the *pivot map*. A variation on the run-length encoding scheme [18], which we call *dirty runs*, is used to record the difference between a map and the pivot map.

A compressed map consists of a sequence of runs, $\{R_1 \ldots R_n\}$. Each run R_i has the structure $\{S_i, D_i, B_1 \ldots B_{D_i}\}$. S_i is the number of similar bytes, D_i is the number of different bytes that follow, and $\{B_1 \ldots B_{D_i}\}$ are the actual D_i different bytes. Maps are assumed to start with similar bytes (if this is not the case, S_0 is zero), and any tail part of the compressed map that is not accounted for in the runs is assumed to be similar to the tail of the pivot map.

Considering the expected size of maps, the S and D fields are almost always less than 128 and thus representable in 7 bits. The sign bit is therefore free to indicate a further optimization: if $S_i \leq 15$ and $D_i \leq 7$ then both values are packed into a single byte whose sign bit is set to indicate that such compression was done.

Maps turn out to contain many zero bytes, so the pivot map itself is saved internally as compressed against a map of similar length containing only zeros. This proved to be a good practice, even for methods having only one map.

The maps relating to one method are combined into a stream of bytes, consisting of:

- A small header indicating the number of maps and the index of the pivot map.
- A "table of contents" (TOC) listing the compressed size of each map. TOC entries are all the same size, but may differ from method to method, and are large enough to accommodate the largest value for the method.
- The dirty runs of all the maps of the method

4.3 Selecting the Pivot Map

The representation of a map is shorter the more similar the map is to the pivot. Thus, choosing a good pivot map is central to the quality of the compression on the whole.

Given a set of N maps, an exhaustive search for the best pivot map among them requires N^2 different compression attempts. We used a heuristic to select a good pivot map in N steps.

Let M_p be the pivot map and M_c a map to be compressed. Define $Cost(M_p, M_c)$ as the length of the compressed representation of M_c with respect to M_p. (With dirty runs scheme, the cost is the number of S and D bytes plus the number of bytes that are actually different between the maps). Define further $TotalCost(M_p)$ as the total length of the compressed representation of all the maps in the method:

$$TotalCost(M_p) = \sum_{i=1}^{N} Cost(M_p, M_i) \qquad (1)$$

where N is the number of maps in the set. The purpose is to find a p such that $TotalCost(M_p)$ is minimal. The algorithm is based on the heuristic assumption that if:

$$Cost(M_p, M_1) - Cost(M_p, M_2) = \delta \qquad (2)$$

then

$$Cost(M_2, M_1) \approx \delta \qquad (3)$$

In other words, a subset of maps that are different from a pivot map by similar degrees will probably be compressed at a lower cost if one of them is chosen as the pivot map instead. It is therefore possible to predict the results of compression with one pivot map based on the actual result of compressing it with another.

The algorithm for choosing a good pivot map is as follows:

1. A map in the set is arbitrarily selected as the initial pivot map M_{init}. (Cost: $O(1)$.)
2. All maps of the set are compressed using M_{init} as pivot. (Cost: $O(N)$.)
3. A histogram H of the costs of all the compressions of maps of the set is created. $H(\lambda)$ is the number of maps whose length after compression is λ, i.e.,

$$H(\lambda) = \| \{i \mid Cost(M_i, M_{init}) = \lambda\} \| \qquad (4)$$

For each λ for which $H(\lambda) > 0$, a pointer to a map whose cost is λ is kept. The histogram itself is of size K, where K is the longest compressed representation of any of the maps in the set. With dirty runs, K is usually much less than the maximal uncompressed string length. (Cost: $O(N)$ in time and $O(K)$ in space.)

4. A λ_{opt} is extracted from the histogram H. In our implementation, the expected λ is used:

$$\lambda_{opt} = \frac{\sum_{\lambda=1}^{K}(\lambda \times H(\lambda))}{N} \tag{5}$$

(Cost: $O(K)$.)

5. A map M_{opt} associated with λ_{opt} is selected as the pivot map. (Cost: $O(1)$ since no search is involved.)

The compression scheme allows for more than one pivot map. This capability may be used to improve the overall compression ratio. Each map is then compressed using the pivot it is most similar to, thus reducing the space for its representation. This approach is useful when the map population is big and aggregated in a small number of clusters. There is a cost involved in encoding the pivot used for compressing each map into its representation. We chose to allow a maximum of two pivot maps, extended the header to include the actual serial numbers of those pivot maps, and used a bit in every field of the TOC to record the logical index $\in [0..1]$ of the pivot map used in that compression.

To determine whether using a second pivot map is advantageous for a given map set, as well as selecting such a second pivot appropriately without repeating the expensive step 2 above, the algorithm is modified as follows:

- Step 4': The λ_{opt} is selected differently in this case to be at the center of the largest cluster in H. (Cost: $O(K)$.)
- Step 5': Same as 5 above.
- Step 6': A second histogram H_2, estimating the effect of using M_{opt} as pivot, is calculated:

$$H_2(\lambda) = H(\lambda_{opt} - \lambda) + H(\lambda_{opt} + \lambda) \tag{6}$$

  One of the maps that was associated with either $\lambda_{opt} - \lambda$ or $\lambda_{opt} + \lambda$ is now made to represent λ. H_2 is smaller in size than H. (cost: $O(K)$.)

Steps 4 and 5 above are repeated using H_2 to obtain a secondary (and subsequent) pivot maps.

4.4 Compression Results

The algorithm above was tested with SPECjvm98 [38]. The tests were run in two modes:

1. Interpreter only mode, where the JIT compiler is not active.
2. Mixed mode, where methods are interpreted initially, and are JIT-compiled only if they are invoked sufficiently often.

Compression statistics collected from tests in the two modes above are summarized in tables 2 and 4, respectively. The numbers in Table 4 represent the sum of the resources taken by the Jvm while running the test (e.g., "total maps size" is the total size of all maps generated, either for interpreted or for JITted methods; maps of same method may appear in both.).

There are a few observations that can be made about these results:

- The compression ratio is computed as the ratio between the space taken by the maps (after compression) and the size of the method's code (bytecode if interpreted, object code if JITted). We see that in all practical cases the space required for maps is less than that taken by the method's code.
- Comparing corresponding entries in Tables 2 and 4, it can be seen that the compression ratio is improved when JITting is done. This is not due to smaller map size but larger code size: bytecode is a rather terse program representation, and is typically much smaller than the object code created by the JIT compiler for the same method.
- The last two columns in Tables 2 and 4 indicate the improvement in compression ratio (in percentage points) that could be achieved if one of the alternatives described above were used instead. The column labeled "optimal pivot" applies to compression using the very best map of the set as a pivot. Note that finding this optimal pivot requires $O(N^2)$ compression tests for a set of N maps. The column labeled "two pivots" applies to the second option, which provides two good pivots without recalculating the histogram.

4.5 Related Work

Space for type maps is an issue every system that makes use of them must deal with. Agesen *et al.* [4] measured their type map overhead to be 57% of the size of the bytecode, compared with our 105% average (see Table 2). They chose a concise representation for the type information, which they do not compress, whereas we have chosen a wide, programmer-oriented representation and relied on compression to compensate for the wasted space. Also, they compute maps for all methods at class load time, while our scheme limits map generation to methods caught on the stack at GC time. When all methods are considered, the map space versus bytecode size ratio is lower (70%; see Table 3). We found that methods encountered on the stack at GC time tend to have more invocations than normal, a property which increases both their number of maps as well as their likelyhood to be found on the stack. Finally, the solution of Agesen *et al.* to the `jsr` problem is based on variable splitting and rewriting of the bytecode. The way we handle `jsr`s does not affect the bytecode, but does increase the size of the maps.

Stichnot *et al.* [39] describe a system totally dependent on JIT compilation where a map can be generated at any instruction. They use a two-level scheme: first, the data itself is encoded efficiently using domain-specific knowledge. Subsequently, Huffman compression is applied to the encoded

test name	number of methods	total size of code (C)	maps total raw size	maps total compressed size (M)	M/C	improvement optimal pivot	improvement two pivots
compress	17	2689	10552	3293	1.22	6.0%	10.9%
jess	42	5682	20212	6233	1.10	5.3%	9.1%
db	22	3204	11672	3655	1.14	6.5%	9.7%
javac	121	21599	64376	21696	1.00	8.1%	7.5%
mpegaudio	12	2114	8592	2615	1.24	3.8%	11.5%
mtrt	27	4793	19120	5972	1.25	8.8%	8.4%
jack	59	10957	36424	10281	0.94	6.1%	8.4%
total	300	51038	170948	53745	1.05	7.0%	8.5%

Table 2: Compression results for SPECjvm tests using bytecode interpretation only.

test name	number of methods	total size of code (C)	maps total raw size	maps total compressed size (M)	M/C
compress	222	16757	54352	12054	0.72
jess	841	50996	175900	51392	1.01
db	212	16988	60456	13802	0.81
javac	1309	102703	255204	77430	0.75
mpegaudio	489	65257	82296	20109	0.31
mtrt	354	25671	90528	23106	0.90
jack	490	48819	123892	31427	0.64
total	3917	327191	842628	229320	0.70

Table 3: Compression results for SPECjvm tests using bytecode interpretation only, when maps are generated for all methods (rather than only for those caught on the stack at GC time).

test name	number of methods	total size of code (C)	maps total raw size	maps total compressed size (M)	M/C	improvement optimal pivot	improvement two pivots
compress	32	6446	17000	4665	0.72	3.7%	4.9%
jess	139	41395	111288	26512	0.64	3.6%	3.0%
db	56	8513	26424	7090	0.83	4.6%	4.4%
javac	517	189342	365320	81113	0.43	2.8%	1.6%
mpegaudio	97	11274	36448	9206	0.82	3.1%	2.9%
mtrt	93	33448	72496	15269	0.46	2.9%	2.4%
jack	288	117225	243376	53097	0.45	3.1%	4.1%
total	1222	407643	872352	196952	0.48	3.1%	2.8%

Table 4: Compression results for SPECjvm tests using both bytecode interpretation and compilation (JIT-ting).

data, also using compression parameters that are collected during offline training runs. They report 20%–30% overhead when compared with the size of the compiled code. Their approach is quite different from the one presented here, making a direct comparison difficult, but for reasons explained above (Section 4.4), compression rates improve as more methods are JITted. Also, Stichnot *et al.* do not compress maps separately, so the time to extract a map using their scheme could be larger.

5. CORRECTNESS OF TYPE MAPS

The generation of stack frame type maps presents a problem for testers. The map generation algorithm is far from trivial and any implementation of it requires careful testing and debugging. On the other hand, testing is not easy. Few of the maps generated are actually used under normal circumstances. Moreover, even an incorrect map is surprisingly unlikely to cause a program to fail.

Map errors could either cause retention of garbage, by identifying a slot as a reference when it is not, or allow the collection of a live object, by failing to identify a slot with a reference. Errors of the former type do not cause program to fail. Those of the latter type can do so only when the missed reference is the sole reference to its object or the referenced object has moved, and even then it is very dependent on program behavior if a noticeable failure actually occurs. This was born out by the tests described below: they uncovered rather fundamental errors in the implementation of the map generation algorithm, even after it had passed without error a variety of benchmarks and a battery of specifically designed test programs.

The test strategy consisted of three components:

1. The Jvm was instrumented to allow for direct verification of map content, independent of whether any map errors could cause program failures.
2. An automated tool for generating test program was employed to cover the more difficult aspects of map generation.
3. A tool measuring the extent to which the map generation code had been *covered* by the tests was used. Additional test programs were handcrafted to expand the coverage to those parts of the code left unexercised.

5.1 Jvm Instrumentation

In order to verify maps directly, the Jvm interpreter was instrumented to place tags on all stack slots, indicating which of them contain object references. (Such tags could in principle solve the type accuracy problem in general, but their maintenance at runtime is too time-consuming. They are therefore useful only in tests where performance is not an issue.)

The tags are not kept on the stack itself but in a separate data structure which parallels it, the *shadow stack*. Tag information is initialized upon entry to a method with the types of the arguments (kept in local variables). It is updated after the execution of any bytecode that affects the stack according to the semantics of the executed bytecode.

Whenever the execution of a method reaches a point for which a map exists, the map is retrieved from the repository and compared with the tag information for the frame. Two kinds of disagreement could occur:

1. the map indicates a reference in a slot tagged as a non-reference, and
2. the map indicates a non-reference in a slot tagged as a reference.

The first kind is clearly an error and is flagged immediately, but the second kind might not be: it is possible that the slot in question is holding some layover value that no execution path will subsequently use. Although the map generator does not perform full liveness analysis, it can detect cases where conflicting types get stored in the same slot on different execution paths. Such slots are declared unusable until they are reinitialized; code which indeed never uses them can pass verification. The instrumentation code, therefore, postpones flagging an error on this kind of mismatch until the slot is actually used. It is interesting to note that although unusable slots among the local variables are expressly permitted [29, p. 146], the verifier rule for merging operand stacks in the JVM Specification [ibid] seems to preclude them from the operand stack. In reality, however, code in which unusable slots occur on the operand stack do pass verification. (The only bytecode which can legally apply to an unusable operand stack slot is `pop`, so this does not seem like a security loophole.)

5.2 Automatic Generation of Test Programs

Standard benchmarks, the JCK 1.3, and some handcrafted tests were used for testing the instrumented Jvm. However, due to the complexity of handling `jsr` bytecode in the course of map generation, test programs that use `jsr` are particularly important.

The Java compiler uses the `jsr` bytecode predictably when compiling a `try {body} finally {handler}` construct (and also for `synchronized` blocks). It is therefore possible to use test programs in Java instead of resorting to programs generated directly in bytecode.

A test-program generation tool, GOTCHA-TCBeans [22] was employed for this purpose. To use this tool, the desired behavior of the code to be tested must first be modeled using a special modeling language. The tool then analyzes the model and generates tests in an abstract language, which is later converted to actual test code; in this case, Java classes.

The model built in this case is based on the specification of the `try` statement in the Java Language Specification[20, Section 14.18, pp. 290–294]. The model includes variables, such as whether to use `catch`, `finally` or both, the nesting depth of `finally` blocks (and consequently of `jsr` subroutines), and the type of exceptions generated: intentional (caused by a `throw`), or accidental (e.g., divide by zero). GOTCHA subsequently creates tests for every permissible combination of the variables. In our case, 5783 test programs were generated.

As discussed in Section 3.1, however, many interesting situations involving `jsr` subroutines may never be generated by a compiler. To cover these cases, the test suite was augmented with programs written directly in bytecode, using a bytecode assembler/disassembler [21].

5.3 Measuring Test Coverage

To measure the extent to which the map generating code is exercised by the test programs, the code-coverage tool ATAC [7] was used. By using an ATAC-provided C compiler, which saves compile-time information and instruments the tested program to produce runtime statistics, one can determine the lines of code that have not been exercised. ATAC provides several types of coverage statistics, but the results below pertain only to statement coverage, which regards any line of code that has been executed at least once as covered.

The ATAC-provided visualization tool can also prioritize lines of code considered uncovered according to how many additional lines are dependent on them: lines whose execution will cause the greatest number of additional lines to be covered are ranked the highest. A human test writer can then concentrate his or her effort where it is likely to make the greatest difference. The existence of unusable slots on the OPS (see Section 5.1 above) was discovered by one of the tests that was generated specifically to improve the coverage results.

5.4 Testing Results

Coverage was measured for the map generation code only, which consists of 6914 lines of code making 2649 basic blocks.

The generation of maps was exercised using tests from the following sources:

- JCK 1.3
- SPECjvm98
- Tests automatically generated by GOTCHA
- `jsr` tests, handcrafted in bytecode
- Tests in Java and bytecode, manually generated based on ATAC recommendations.

The coverage results are summarized in Table 5. As expected, the greatest contribution comes from the JCK. It is interesting to note that some JCK tests are so small that they contain no maps at all and do not contribute to the coverage. It is not surprising to see that the SPECjvm98 test suite does not add any coverage beyond what is provided by the JCK. The effectiveness of the hints ATAC provides the test writer are also evident from the data.

The 83% represents close to complete statement coverage. Most of the code in the remaining 17% is in one of the following categories:

1. Error handling code to handle situations that should never occur.

test	coverage	
suite	added	total
JCK 1.3		78.4%
SPECjvm98	0.0%	78.4%
GOTCHA	0.7%	79.1%
`jsr` tests	0.3%	79.4%
ATAC-directed	3.4%	82.8%

Table 5: Code-coverage test results.

2. Trivial transformations for quick bytecodes [28, Chapter 9] not exercised due to the testing methodology. The instrumented Jvm generates a method's maps when it is first invoked before any of its bytecodes have been converted to their quick equivalents.

3. Separate map generation for small and large methods. A small method is one with fewer than 16 slots of local variable and operand stack entries combined. Every possible bytecode is treated in either case, but the `wide` bytecode variants never occur in short methods.

6. PERFORMANCE

Our implementation of mostly accurate stack scanning was compared with conservative stack scanning on an IBM prototype of J2SE v1.3 for Windows [25]. The garbage collector was essentially the same in both cases: a stop-the-world parallel mark and sweep collector, which compacts only when necessary. The only differences between the two collectors were minor changes to compaction designed to take advantage of accurately scanned stack slots. We compared both the performance of the Jvm and the efficiency of the GC. All tests were run on IBM Netfinity 7000, a PC server with 4 550MHz Pentium III Xeon^{TM} processors and 2GB of RAM running Windows NT 4.0. The programs used were taken from SPECjvm98 [38], a suite of client benchmarks, and SPECjbb2000 [37], a multithreaded server benchmark.

Table 6 presents performance data collected while running SPECjvm with a 12MB heap. We ran the Jvm in two modes:

- with interpretation only, and
- in mixed mode, where methods are interpreted initially, and are JIT-compiled only if they are invoked sufficiently often.

The best of 10 running times in both cases was selected. The results corroborate our expectation that the overhead incurred for mostly accurate stack scanning is small.

Tables 7, 8 and 9 show how much memory is found in conservatively referenced objects. Tables 7 and 8 refer to SPECjvm using 12MB and 24MB heaps, respectively, while Table 9 describes SPECjbb running in a heap of 512MB. The tables provide object counts as well as total object sizes. The numbers represent averages of measurements taken at the end of each GC cycle. The results indicate that mostly accurate stack scanning removes a significant part of the conservatively referenced memory. In particular, the number of non-relocatable objects was reduced by 37% to 94%

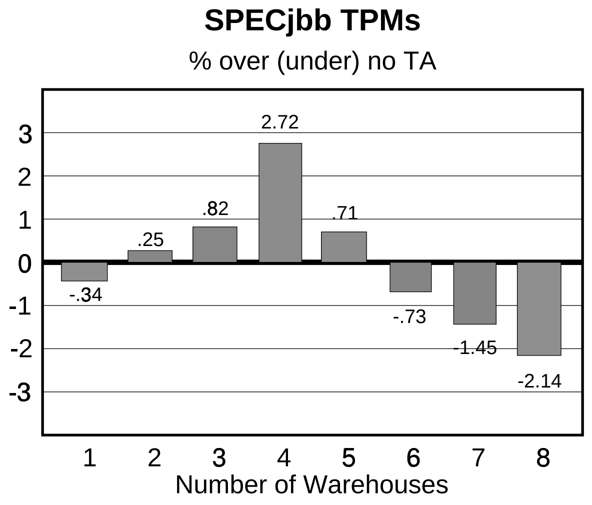

Figure 2: SPECjbb 2000 performance with TA compared with results without TA.

and the space they occupy was reduced by 40% to 99.7%. The reduction was particularly noticeable in benchmarks with the largest number of such objects. Moreover, most of the varability in test results is due to variations in the behavior of the non-TA Jvm; the number and size of immovable objects with mostly accurate stack scanning seems to be independent of heap size and number of GC cycles in the run.

SPECjbb is a benchmark which simulates a 3-tier transaction processor. Unlike other benchmarks, SPECjbb lets the application reach a steady state and then measures the rate at which its simulated server responds to its simulated clients, reported as transactions per minute (TPMs). This form of measurement makes it difficult to separate the overhead incurred from the benefits obtained when mostly accurate stack scanning is applied. Figure 2 shows the percent performance change of mostly accurate scanning with respect to the base, conservative scanner. Performance is improved up to a peek of 2.7% over base (at 4 warehouses), but at larger numbers of warehouses performance deteriorates until it drops 2.1% below base. From studying the GC trace output (not shown) it seems that although scanning the stacks accurately takes about 80% longer than it does when done conservatively, the overall time spent doing GC is practically not affected. The difference in the TPM figures must stem from other sources, currently under investigation.

7. CONCLUSIONS AND FUTURE WORK

We have tried to make a case for mostly accurate stack scanning as a viable alternative to full accuracy in cases where performance outweighs GC efficiency, and also where the complexity involved in implementing full accuracy is not justified. We have shown that a mostly accurate Jvm can reduce the amount of conservatively referenced memory to very low levels while making a very small impact on performance. There is room to explore other frame types that could also be accurately scanned without hurting performance. A better understanding of the characteristics of conservative and mostly accurate stack scanning and the tradeoffs between them could provide leads to additional improvements.

We have consciously chosen a programmer-friendly format for the type maps, in which the data is redundantly encoded. A more concise representation could improve repository compression rates, but at the same time increase the overhead involved in uncompressing maps at GC time.

We have applied a comprehensive testing methodology to map generation for interpreted bytecodes. It would be interesting to extend this methodology to map generation for compiled (JITted) code. In particular, extending the shadow stack mechanism would not be trivial.

Acknowledgments: We would like to thank the many people who have contributed to this project: Alain Azagury, Victor Leikehman, Yossi Levanoni, Erez Petrank, Dafna Sheinwald, Boaz Shmueli and Sagi Snir of HRL, Toshio Nakatani, Toshio Suganuma, Tamiya Onodera, Takeshi Ogasawara and many others at TRL, Kean Kuiper of Austin Java Performance Group, and Dionis Hristov of Toronto. Special thanks go to our colleagues at the JTC: Martin Trotter, whose proposal of a shadow stack to test the maps with is but one of his many contributions to this effort, and to Sam Borman, who together have nurtured this project from its inception.

8. REFERENCES

[1] Ali-Reza Adl-Tabatai, Michal Cierniak, Guei-Yuan Leuh, Vihesh M. Parikh, and James M. Stichnoth. Fast effective code generation in a Just-In-Time Java compiler. In PLDI [34].

[2] Ole Agesen. GC points in a threaded environment. Technical report, Sun Microsystems, Inc., 1999. TR-98-70.

test	without JIT			with JIT		
name	w/o TA	w/ TA	change	w/o TA	w/ TA	change
compress	291.4	288.5	-1.01%	15.6	15.6	0.0%
jess	77.7	77.3	-0.52%	13.9	13.9	-0.0%
db	158.6	155.6	-1.93%	29.3	29.2	-0.3%
javac	108.1	106.9	-1.12%	38.4	38.7	0.8%
jack	64.9	64.5	-0.62%	22.6	22.5	-0.4%
mpeg	257.1	257.2	0.04%	12.6	12.5	-0.8%
mtrt	80.2	80.2	-0.07%	8.8	8.8	0.1%
total	1038.0	1030.2	-0.76%	141.2	141.2	-0.0%

Table 6: Running times of SPECjvm98 tests with and without mostly accurate stack scanning. Tests were run with a fixed heap size of 12MB.

	GC cycles		Average object count				Average memory size (bytes)			
test	w/o	w/	total	immovable			total	immovable		
name	TA	TA		w/o TA	w/ TA	change		w/o TA	w/ TA	change
compress	20	20	5715	51.7	26.0	49.7%	3413692	1880	880	53.2%
jess	35	35	25532	345.6	25.3	92.7%	1553343	21930	862	96.1%
db	28	28	275351	69.2	25.1	63.7%	8399507	78767	858	98.9%
javac	97	84	238768	224.3	40.5	81.9%	8632638	15079	1227	91.9%
jack	18	18	11244	82.6	26.0	68.5%	859647	3290	880	73.3%
mpeg	2	2	6217	41.0	26.0	36.6%	590608	1472	880	40.2%
mtrt	26	26	253945	199.8	28.4	85.8%	7151880	7385	1029	86.1%

Table 7: Amount of memory in conservatively referenced objects. Tests were run with a fixed heap size of 12MB. The numbers reflect the average of measurements made at the end of each GC cycle.

	GC cycles		Average object count				Average memory size (bytes)			
test	w/o	w/	total	immovable			total	immovable		
name	TA	TA		w/o TA	w/ TA	change		w/o TA	w/ TA	change
compress	7	7	5532	50.3	26.0	48.3%	2129904	332428	880	99.7%
jess	16	16	24045	201.1	25.3	87.4%	1470944	11883	863	92.7%
db	8	8	223300	53.5	25.3	52.7%	6864361	63524	862	98.6%
javac	26	26	196537	178.7	40.4	77.4%	7194786	9189	1225	86.7%
jack	10	10	14392	75.1	26.0	65.4%	980317	3251	880	72.9%
mpeg	2	2	6189	41.0	26.0	36.6%	582660	1472	880	40.2%
mtrt	9	9	210157	130.2	27.9	78.6%	5989970	4600	1000	78.3%

Table 8: Amount of memory in conservatively referenced objects. Tests were run with a fixed heap size of 24MB. The numbers reflect the average of measurements made at the end of each GC cycle.

	GC cycles		Average object count				Average memory size (bytes)			
test	w/o	w/	total	immovable			total	immovable		
name	TA	TA		w/o TA	w/ TA	change		w/o TA	w/ TA	change
SPECjbb	126	127	727379	268	17	93.7%	83798673	17236	859	95.0%

Table 9: Amount of memory in conservatively referenced objects used by SPECjbb. Tests were run with a fixed heap size of 512MB. The numbers reflect the average of 3 runs.

[3] Ole Agesen and David Detlefs. Finding references in Java stacks. In Peter Dickman and Paul R. Wilson, editors, *OOPSLA '97 Workshop on Garbage Collection and Memory Management*, October 1997.

[4] Ole Agesen, David Detlefs, and J. Eliot B. Moss. Garbage collection and local variable type-precision and liveness in Java Virtual Machines. In PLDI [34], pages 269–279.

[5] 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 System Journal*, 39(1):211–238, 2000.

[6] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, A. Cocchi, S. F. Hummel, D. Lieber, T. Ngo, M. Mergen, J. C. Shepherd, and S. E. Smith. Implementing Jalapeño in Java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA '99)*, volume 34, 1999.

[7] ATAC: Automatic test analysis for C, 1998. Available at http://xsuds.argreenhouse.com.

[8] Joel F. Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88/2, DEC Western Research Laboratory, Palo Alto, CA, February 1988. Also in Lisp Pointers 1, 6 (April–June 1988), 2–12.

[9] Joel F. Bartlett. Mostly-Copying garbage collection picks up generations and C++. Technical note, DEC Western Research Laboratory, Palo Alto, CA, October 1989. Sources available in ftp://gatekeeper.dec.com/pub/DEC/CCgc.

[10] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[11] C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–8, November 1970.

[12] Java Virtual Machine by Sun Microsystems, Inc. Available at http://www.sun.com/software/sales.

[13] Robert Dimpsey, Rajiv Arora, and Kean Kuiper. Java server performance: A case study of building efficient, scalable Jvms. *IBM System Journal*, 39(1):151–174, 2000.

[14] Amer Diwan, J. Eliot B. Moss, and Richard L. Hudson. Compiler support for garbage collection in a statically typed language. In *Proceedings of SIGPLAN'92 Conference on Programming Languages Design and Implementation*, volume 27 of *ACM SIGPLAN Notices*, pages 273–282, San Francisco, CA, June 1992. ACM Press.

[15] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Conference Record of the Twenty-first Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices. ACM Press, January 1994.

[16] Tamar Domany, Elliot Kolodner, and Erez Petrank. A generational on-the-fly garbage collector for Java. In *Proceedings of SIGPLAN 2000 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Vancouver, June 2000. ACM Press.

[17] EVM: an exact Java Virtual Machine by Sun Microsystems, Inc. Available (as part of J2SE) at http://www.sun.com/solaris/java.

[18] S.W. Golomb. Run-length encodings. *IEEE Transactions on Information Theory*, 12(3):399–401, July 1966.

[19] James Gosling. Java intermediate bytecodes. In *Proceedings of the ACM SIGPLAN Workshop on Intermediate Representations*, pages 111–118, January 1995.

[20] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* The Java Series. Addison-Wesley, 1997.

[21] Matt Greenwood and Sara Porat. TOAD - an environment for monitoring understanding and optimizing Java. In *OOPSLA '99 Companion*, 1999. Available at http://www.alphaworks.ibm.com/tech/toad.

[22] Alan Hartman and Kenneth Nagin. TCBeans Software Test Toolkit. In *Proceedings of the 12th International Software Quality Week*, May 1999. Available at http://www.haifa.il.ibm.com/projects/gtcb.

[23] Java HotSpot Technology by Sun Microsystems, Inc. Available at http://java.sun.com/products/hotspot.

[24] Richard L. Hudson and J. Eliot B. Moss. Incremental garbage collection for mature objects. In Yves Bekkers and Jacques Cohen, editors, *Proceedings of International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, University of Massachusetts, USA, 16–18 September 1992. Springer-Verlag.

[25] IBM 1.3 Java Developer Kit for Windows. Available at http://www.software.ibm.com/download.

[26] Open runtime platform by Intel, Inc. Available at http://intel.com/research/mrl/orp.

[27] Henry Lieberman and Carl E. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983. Also report TM–184, Laboratory for Computer Science, MIT, Cambridge, MA, July 1980 and AI Lab Memo 569, 1981.

[28] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* The Java Series. Addison-Wesley, 1st edition edition, 1997.

[29] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* The Java Series. Addison-Wesley, 2nd edition edition, 1999.

[30] SUN Microsystems. Java Native Interface Specification. Technical report, Sun Microsystems, Inc., http://java.sun.com/j2se/1.3/docs/guide/jni/, 1997.

[31] SUN Microsystems. JNI enhancements in JDK 1.2. Technical report, Sun Microsystems, Inc., http://java.sun.com/j2se/1.3/docs/guide/jni/jni-12.html, 1998.

[32] David A. Moon. Garbage collection in a large LISP system. In Guy L. Steele, editor, *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 235–245, Austin, TX, August 1984. ACM Press.

[33] Tamiya Onodera. A generational and conservative copying collector for hybrid object-oriented languages. *Software Practice and Experience*, 23(10):1077–1093, October 1993.

[34] *Proceedings of SIGPLAN'98 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Montreal, June 1998. ACM Press.

[35] Tony Printezis and David Detlefs. A generational mostly concurrent garbage collector. In *Proceedings of the International Symposium on Memory Management (ISMM00)*, pages 143–154, Minneapolis, MN, October 2000.

[36] Jacob Seligmann and Steffen Grarup. Incremental mature garbage collection using the train algorithm. In O. Nierstras, editor, *Proceedings of 1995 European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, University of Aarhus, August 1995. Springer-Verlag.

[37] SPECjbb2000 Java Business Benchmark. Standard Performance Evaluation Corporation (SPEC), Fairfax, VA, 1998. Available at http://www.spec.org/osg/jbb2000/.

[38] SPECjvm98 Benchmarks. Standard Performance Evaluation Corporation (SPEC), Fairfax, VA, 1998. Available at http://www.spec.org/osg/jvm98/.

[39] James M. Stichnoth, Guei-Yuan Lueh, and Michal Cierniak. Support for garbage collection at every instruction in a Java compiler. In *Proceedings of SIGPLAN'99 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 118–127, Atlanta, May 1999. ACM Press.

[40] David M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984. Also published as ACM Software Engineering Notes 9, 3 (May 1984) — Proceedings of the ACM/SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 157–167, April 1984.