

Submitted to the OOPSLA'97 Workshop on Garbage Collection and Memory Management

Finding References in JavaTM Stacks

Ole Agesen, David Detlefs

Sun Microsystems Laboratories
2 Elizabeth Drive
Chelmsford, MA 01824, U.S.A.
ole.agesen@sun.com, david.detlefs@east.sun.com

August, 1997

Abstract. Exact garbage collection for the strongly-typed Java language may seem straightforward. Unfortunately, a single pair of bytecodes in the Java Virtual Machine instruction set greatly presents an obstacle that has thus far not been discussed in the literature. We explain the problem, outline the space of possible solutions, and present a solution utilizing bytecode-preprocessing to enable exact garbage collection while maintaining compatibility with existing compiled Java class files.

1 Introduction

All garbage collection algorithms determine reachability of objects from some set of *roots*. In most language implementations, *stacks* form one component of the root set. A stack is a region in which *stack frames* may be allocated and deallocated. Each method executing in a thread of control allocates a stack frame, and uses the *slots* of that stack to hold the values of local variables. Some of those variables may contain references to heap-allocated objects. Such objects must be considered reachable as long as the method is executing. The term *stack* is used because the stack frames obey a last-in/first-out allocation discipline within a given thread of control (at least in languages without closures). There is generally a stack associated with each thread of control.

A garbage collector may be exact or conservative in how it treats different sources of references, such as stacks. A *conservative* collector knows only that some region of memory may contain references, but doesn't know whether or not a given value in that region is a reference. If such a collector encounters a value that is a plausible reference value, it must keep the "referenced" object alive. Because of this uncertainty, the collector is constrained not to move the object, since that would require updating the "reference," which might actually be an unfortunately-valued integer or floating-point number. The main advantage of conservative collection is that it allows garbage collection to be used with systems not originally designed to support collection. For example, the Boehm-Weiser collector [2] and the Bartlett collector [1] use conservative techniques to support collection for C and C++.

In contrast, a collector is *exact* in its treatment of a memory region if it can accurately distinguish references from non-reference values in that region. Exactness has several advantages over conservatism. A conservative collector may retain garbage "referenced" by a non-reference value that an exact collector would reclaim. Perhaps more importantly, an exact collector is free to relocate objects referenced only by exactly identified references. In an *exact system*, one in which references and non-references can be distinguished everywhere, this enables a wide range of useful and efficient garbage-collection techniques that cannot (easily) be used in a conservative setting.

A drawback of exact systems is that they must somehow provide the information that makes them exact. This may introduce both performance and complexity overhead. One technique for providing this information is *tagging*, in which values are self-describing: one or more bits in each value is reserved to indicate whether the value is a reference. Tags may be supported by hardware, but more commonly require the generation of extra instructions to check, mask, and restore tags. If tagging is not used, then the system must associate data structures with each memory region allowing references and non-references to be distinguished in that region. For example, each object might start with a reference to a descriptor of the type of the object, which would include a *layout map* describing which fields of the object contain references. Stack frames are another kind of memory region that may contain references, so data structures describing which variables in a stack frame contain references would also have to be generated; we will call such a data structure a *stack map*. The subject of this paper is the generation of stack maps for stack frames in the Java virtual machine. We describe the general problems associated with designing and generating such stack maps, and

some particular problems with doing so for Java. We have implemented our approach, and describe that implementation.

2 Stack maps and gc points

In the scheme above, the layout map of an object was associated with its type; all objects of a given type have the same layout map, and the map of an object is the same for its entire lifetime. In contrast, the layout of a stack frame is usually allowed to change during its lifetime. A given stack frame slot may be uninitialized at the start of a method, hold a reference variable for one block in the method, then hold an integer variable for another. For example, a compiler may translate the following program fragment

```
if (b) {
    int i;
    ...
} else {
    Rattlesnake r;
    ...
}
```

in such a way that the integer variable `i` and the reference variable `r` are mapped to the same slot in the stack frame. In fact, one could imagine that whether a given stack frame slot contains a reference at a given point in a method execution might depend not only on the current point in the program execution but also on the control path leading to that point, as in the code fragment below.

```
if (b) v = an int
else v = a reference;
...gcPoint...
if (b) use v as an int
else use v as a reference
```

We will call the first kind of variation in stack layout a *control point dependency*, the second a *control path dependency*. One approach to generating stack maps would be to forbid either kind of dependency: the layout of stack frames allocated for invocations of method *m* would be the same at all points of all executions of *m*. This means that stack frame slots would be typed as reference slots or non-reference slots, and that all reference slots would be initialized (probably to the null reference) before execution of the method begins. This is perhaps a viable design choice, though it may cause methods to require larger stack frames, and may impose extra initialization overhead on method start-up. More to the point for this paper, however, is that it is not the design choice made by Java.

Another approach allows control point dependencies, but forbids control path dependencies. The stack map data structure would then have to be rich enough to be able to map execution points within a method to the layout map appropriate for that execution point. If collection may occur at any instruction in a method, these stack maps may be voluminous, complicated, and/or time-consuming to process during collection. Most of these problems can be avoided by requiring that collections can occur only at a small set of *gc points* within each method. Then the stack map data structure would need only to associate a layout map with each *gc point* of the method, decreasing the storage required, usually to a point where simpler and faster data structures can be used with reasonable storage costs.

We will not discuss in any detail how a system may ensure that each method of each thread is stopped at a *gc point* when collections occur, except to note that several techniques are known. We will discuss very briefly the criteria for choosing rules that define *gc points*. First, all methods executing in a thread except for the currently executing method will be stopped at calls, implying that method call instructions should probably be *gc points*. Second, we'd like to ensure that when some thread *A* attempts an allocation that fails because a collection is necessary, no thread *B* can delay *A* by requiring arbitrarily long to come to a *gc point*. In our system, we make every backwards branch a *gc point*, since such a branch may be part of a loop.

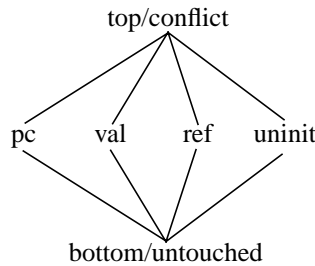
3 Generating stackmaps for Java

“Java” is actually two separable things that are sometimes conflated: the Java language [4], and the Java virtual machine [5]. We are interested in the Java virtual machine, an execution environment for Java class files containing methods expressed in the machine-independent Java “byte code” instruction set.

The Java virtual machine specification [4] lays out a set of rules defining legal Java class files, along with a description of a procedure (the Java “byte code verifier”) for checking adherence to those rules. We call one of these rules the *Gosling property*, since James Gosling first stated it explicitly [3]. The Gosling property requires that if two control paths lead to a given instruction I that uses some local (*i.e.*, stack-allocated) variable v , then I must be legal assuming that the type of v is the *merge* of its types along the two control paths. The merge of two types is the least general type that contains the two types (the least upper bound in the subtype lattice.) Such a type always exists in the Java type system when the types being merged are object types, since the class `java.lang.Object` is a superclass of all classes, including array classes. However, if one type is a reference type and the other is a primitive type such as `int`, there is no common supertype. To make the merge operation well-defined in such cases, a *top* type, called *conflict*, is added to the type lattice. Any use of a variable whose inferred type is *conflict* is illegal; the variable can be used only after an assignment has reset its type to a non-*conflict* type.

The Gosling property implies that stack maps for legal Java methods do not need to consider control path dependencies; the layout at a control point depends only on that control point, not how it was reached. Further, the bytecode verification process gives us a model for how to generate the stack maps. The part of verification that checks types is a kind of *abstract interpretation*. An abstract interpretation is an execution of a program that “approximates” the values of the program variables and the control flow. Values are abstracted as elements of some lattice. The interpretation keeps track of the set of possible values of every variables at every control point, represented as the least lattice element that contains all the possible values. All variables are initialized with the *bottom* element of the lattice at each control point, and the first instruction of the method is marked as *changed*. The abstract interpreter then executes a loop that picks a *changed* instruction as long as one exists, removes the *changed* mark, applies the operation represented by the instruction to yield a new vector of variable values, and element-wise merges this vector with the state vectors of the control points that can follow the current instruction. If the merge changes the state vector associated with any of these successor instructions, then those instructions are marked *changed*. Because the value space is a lattice, which has no infinite ascending chains, this process must terminate.

In bytecode verification, the lattice contains primitive types and object types, augmented, as we saw, with a *top* type *conflict* and a *bottom* type. For stack map generation, we can use an even coarser approximation, in which all object types are lumped together as *ref*, and all non-reference (primitive value) types are coalesced as *val*. We need an additional type *uninit* representing an uninitialized variable since uses of such variables must be rejected, and a type *pc* for program counters since the Java bytecode `jsr` pushes return addresses on the stack, yielding this lattice:



To start the abstract interpretation for a method, we set all variables at the entry point to *uninit*, since this corresponds to the starting condition during execution. At all other points in the method, the variables have the type *bottom* so that any other type flowing into such a point will result in the variable having that type. Performing the abstract interpretation over this lattice with the specified starting conditions yields stack layouts for all the instructions of the method. We can then choose those corresponding to gc points for permanent storage in a compact stack map data structure associated with the method.

4 The jsr problem

Unfortunately, we have not yet told the complete story. The Java virtual machine explicitly allows one exception to the Gosling property. The Java bytecode instruction set includes a pair of operations called `jsr` and `ret`. The `jsr` instruction jumps to an address specified in the instruction and pushes a return address value on the operand stack¹ of the current method. The `ret` instruction specifies a local variable that must contain a return address, and jumps to that return address.

The intended use of these bytecodes is in the implementation of the

```
try { body } finally { handler }
```

construct of the Java language, in which *handler* is executed no matter how *body* is exited. The *handler* would be translated as a *jsr subroutine*: a “mini-method” within the method. Every instruction that exits *body*, such as `return` or `throw` statements or “falling off the end”, would be preceded in the translation by a `jsr` to that subroutine, which would store the pushed return address in a local variable, perform the work of *handler*, then perform a `ret`. Although a `jsr` subroutine resembles a real method, there is a crucial difference: it executes in the same stack frame as its containing method and has access to all the local variables of this method.

The Java virtual machine specification for verification of `jsr` subroutines contains an explicit exception to the Gosling property [5] (p. 136): the bytecode verifier permits any local variable v that is neither read nor written in a `jsr` to retain its type across a `jsr` to that subroutine.

This seemingly reasonable weakening of the Gosling property causes serious difficulty for exact garbage collection. Consider a case in which there are two `jsr`'s to the same `jsr` subroutine. At one `jsr`, a local variable v is being used to hold an integer, and at the other, it holds a reference. Should a garbage collection occur while a thread is in the `jsr` subroutine, a control-point-dependent stack map is unable to determine if v contains a reference, since the stack layout is now control path dependent. Simply disallowing garbage collections for the duration of the `jsr` subroutine is not an option since `try-finally` handlers can perform arbitrary computation, including calling methods that may execute indefinitely and allocate an unbounded number of objects.

5 Possible solutions

There are several possible solutions to this problem. An obvious one is to change the Java virtual machine specification to remove the Gosling property exception for `jsr` subroutines. Not only would this eliminate the problem for garbage collectors, it would also significantly simplify the verification process. One of the main selling points of the Java system is safe execution of untrusted code; this guarantee relies on the correctness of the verifier. The simpler the verifier, the more confidence we can have in its correctness. On the other hand, there are some drawbacks to tightening the verifier specification to enforce the Gosling principle everywhere. Java compilers would have to be modified to obey the new rules. We believe that the original motivation for the exception was to keep the Gosling property from preventing “natural” reuse of local variable slots. For example, if a method m contains two sequential blocks, one that uses local variable slot s to hold an object of class A , and another that uses s to hold an object of class B , where A and B have no common supertype other than `Object`, then there was a fear that `jsr`'s in the two blocks to some common `jsr` subroutine would prevent the reuse of the slot s , increasing the number of local variable slots required by Java methods. If this fear is justified in practice, then removing the exception is problematical.

Probably the most important drawback to tightening the verifier specification, however, is that it may invalidate existing Java class files. There is already a very large installed base of compiled Java programs. Staging a change that potentially requires many existing programs to be recompiled with a new compiler is, to say the least, logistically difficult.

Another class of solution would be to “live with” the control path dependency introduced by the Gosling property exception. The stack map data structures and the analysis that creates them would be enhanced to track `jsr` return values, and use those return values to help determine stack layouts. We find this solution complicated, especially when one considers nested `jsr` subroutines, where it seems possible to construct pathological examples requiring exponential space in the stack map data structures. However, such cases probably are rare, and we see no reason that such solutions could not be made to work with sufficient skill and enthusiasm.

The remaining class of solutions rewrite existing class files to eliminate violations of the Gosling property (or at least violations relevant to garbage collection.) Such solutions may be used as a long-term strategy, or might be used to help stage the introduction of a tighter verifier specification, by allowing old-style class files to run with new-style vir-

1. For the present discussion, it is unnecessary to distinguish between stack frame slots holding operand stack values and stack frame slots holding local variables: we think of them as two separate sets of local variables, one being addressed from the stack frame base, the other being addressed from the top of stack pointer.

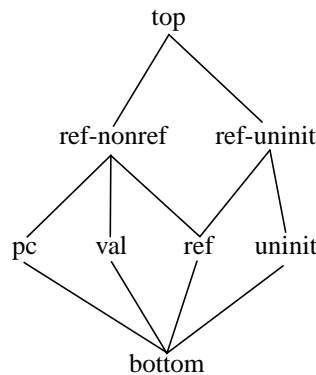
tual machines. These solutions come in two flavors. One approach eliminates conflicts caused by jsr subroutines by duplicating the subroutines. That is, if local variable v holds an integer at one jsr to a jsr subroutine, and a reference at another jsr to the same subroutine, then the conflict for v within the jsr subroutine can be eliminated by duplicating the subroutine, and retargeting one of the jsr instructions to the new copy of the subroutine. By extension, all jsr's at which v holds a reference would go to one copy, and all jsr's at which v holds a non-reference would go to the other copy.

We do not favor this approach, either; even in this simple case it increases code size. Worse, when one considers two local variables and four jsr instructions to a single jsr subroutines, it is not hard to see that four copies of the subroutine might be required, and so on exponentially.

Instead, the solution we have implemented rewrites conflicting local variables instead of duplicating code. It could be viewed as “splitting” a variable into reference-containing and non-reference-containing halves, which effectively undoes some of the local variable slot reuse the compiler was allowed by the exception to the Gosling principle. The next section describes our solution in detail.

6 Bytecode rewriting to split conflicting variables

Our first step was to refine the lattice used in the abstract interpretation that computes stack maps to record not only that a conflict occurs but also the kind of conflict.



We need this more detailed information because we resolve conflicts between references and uninitialized values (*ref-uninit* conflicts) differently from conflicts between references and non-reference values (*ref-nonref* conflicts):

- *ref-uninit* conflicts are eliminated by prepending code to the start of the method to initialize the variables to null.
- *ref-nonref* conflicts are eliminated by splitting the variable.
- *top* conflicts are resolved by a combination of the above two actions.

The stack map computation for a method m is augmented as follows. The abstract interpretation and conflict elimination is iterated until all conflicts have been eliminated. A variable *varsToInit* holds a set of reference-containing variables requiring initialization. It is initially empty. The abstract interpretation considers the variables in the set to hold initialized reference values at the start of the method. Each iteration initializes a variable *varsToSplit* to the empty set of variables. This set will hold variables that were found to hold a *ref-nonref* conflict when they were used. Finally, each iteration initializes a flag *conflictOccurred*, indicating whether a conflict occurred, to false.

The stack map computation then proceeds as described previously, except in its handling of conflict values. A use of a variable whose value in the abstract interpretation is the *ref-uninit* conflict value causes the variable to be added to *varsToInit*. A use of a variable holding the *ref-nonref* conflict value in the interpretation adds the variable to *varsToSplit*. Either kind of conflict use sets the *conflictOccurred* flag to true.

At the end of an iteration, *varsToSplit* is checked. If it is non-empty, then each variable in it is *split*. To split local variable n , we increase the number of local variables allocated in stack frames for method m by one; let mn be the number of newly allocated local variable. We then examine the bytecodes for the method m , modifying them so that instruc-

tions that use the original variable n to hold references are unchanged, but non-reference uses are changed to use variable nm instead (the other choice, where reference uses of n are changed to use nm and non-reference uses are unchanged, is equivalent). It is a happy property of the Java bytecode instruction set that instructions have sufficient type information encoded in their opcodes to determine locally whether a given instruction uses a local variable as a reference, making the rewriting fairly simple. There is one exception to this property: the `astore` instruction is usually used to pop a reference (and *address*, hence the prefix letter *a*) from the operand stack and store it in a local variable, but it may also be used to do the same with return addresses pushed on the stack by `jsr` instructions. Fortunately, the abstract interpretation already maintains sufficient state to determine whether the operand stack top at the point of the `astore` is such a return address, so this complication is easily circumvented.

There is one more complication. Rewriting instructions to reference different local variables may change the width of those instructions. The Java bytecode instruction set optimizes some important cases; operations that read or write local variables have two forms, one that takes a general variable number as an argument, and another, used for the first few (more precisely, four) local variables, that encodes the variable being referenced in the opcode of the instruction. Empirically, the first few variables are heavily used, so this optimization compacts the bytecodes and speeds their execution. However, when we change a one-byte `astore_2` instruction to use local variable 10 instead of local variable 2, the new instruction occupies two bytes. Such changes require relocation of all following instructions, recalculation of relative jumps that cross the modified instruction, and adjustment of instructions, such as `switch` instructions, with internal alignment constraints. We will not describe this relocation process in detail, except to alert readers to the potential complications. Our implementation includes a general bytecode relocater that solves these problems.

If any uses of conflict variables are detected, at least some are repaired by this variable-splitting process or by addition to the `varsToInit` set. The next iteration of the loop may still find some conflicts in the rewritten code (perhaps a variable has both ref-uninit and ref-value conflicts), causing another iteration, or it will detect no conflicts and successfully generate the stack maps.

The rewriting is not guaranteed to succeed. Allocating new local variables could exceed the limit on the number of locals in a method, which is imposed by the bytecode instruction set. Widening instructions could conceivably cause a method to exceed the maximum method size. In such cases, the virtual machine would have to somehow indicate an error akin to a verification error. In practice, however, we expect programs that violate these limits to be extremely rare.

In practice, the performance of the bytecode rewriting is not a crucial issue since with the most commonly used compiler, `javac`, very few methods need rewriting. For example, on a particular set of Java programs, comprising several thousand lines of code, only three methods were rewritten, as demonstrated by this “verbose” run of our virtual machine:

```
Rewrote ref-uninit conflict in method sun/io/CharToByteConverter.convertAll (115 bytes).
Rewrote ref-uninit conflict in method sun/io/ByteToCharConverter.convertAll (115 bytes).
Rewrote ref-uninit conflict in method sun/tools/java/Parser.parseClassBody (271 bytes).
```

The above three methods are too large to include here, but we have hand-written a short method with a ref-nonref conflict and show its bytecodes before and after rewriting:

Method before rewriting	Method after rewriting
Method void refInt1()	Method void refInt1()
0 iconst_3	0 iconst_3
1 istore_0	1 istore_2
2 jsr 15	2 jsr 15
5 iload_0	5 iload_2
6 pop	6 pop
7 aconst_null	7 aconst_null
8 astore_0	8 astore_0
9 jsr 15	9 jsr 15
12 aload_0	12 aload_0
13 pop	13 pop
14 return	14 return
15 astore_1	15 astore_1
16 ret 1	16 ret 1

7 Conclusions

In summary, we have seen that exact garbage collection requires the computation of stack maps, data structures that determine which slots in stack frames contain references and which do not. The Java virtual machine specification almost universally enforces a property we have called the Gosling property, which has the desirable consequence of preventing stack frame layouts from having control-path dependencies, allowing simpler stack map data structures. However, an exception to the Gosling property was made for jsr subroutines, which allows control-path dependencies to creep back in. The first contribution of this paper is simply to point out that this problem exists, since we don't believe its existence is widely appreciated.

The second contribution of this paper is to identify the space of possible solutions. The purest approach would be to disallow any exceptions to the Gosling property. This would simplify exact garbage collection and, furthermore, remove a source of complexity from the Java bytecode verifier. A simpler verifier decreases the probability of security holes, and perhaps even makes formal verification of their absence tractable. But if practical problems prevent this approach, then we have described a technique that rewrites methods that violate the Gosling property to a form that respects it. This technique may be used as a permanent solution, or, if the purer solution is adopted as the long term strategy, it may be used as a bridge to allow older class files to execute on newer virtual machines that enforce the Gosling property universally.

Acknowledgments. The greatly simplifying realization that *ref* and *nonref* uses of variables can be distinguished based only on the opcode of the using instruction, came about in a discussion with Boris Beylin, Ross Knippel, and Bob Wilson. John Rose explained to us the behavior of the javac compiler when translating the `try-finally` construct.

References

1. Bartlett, Joel F. *Mostly-Copying Collection Picks Up Generations and C++*. Technical Report TN-12, DEC Western Research Laboratory, October 1989.
2. Boehm, Hans Juergen and Weiser, Mark. Garbage Collection in an Uncooperative Environment. *Software—Practice & Experience*, 18(9), p. 807-820, September 1988.
3. Gosling, James. Java Intermediate Bytecodes. In *Proceedings of ACM SIGPLAN Workshop on Intermediate Representations (IR'95)*, p. 111-118, January 1995. Published as *ACM SIGPLAN Notices* 30(3), March 1995.
4. Gosling, James, Joy, Bill, and Steele, Guy. *The Java Language Specification*, The Java Series, Addison-Wesley, 1996.
5. Lindholm, Tim and Yellin, Frank. *The Java Virtual Machine Specification*. The Java Series, Addison-Wesley, 1996.

Sun, Sun Microsystems, and Java are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.