

Support for Garbage Collection at Every Instruction in a Java™ Compiler

James M. Stichnoth[†]
stichnot@inktomi.com

Guei-Yuan Lueh
guei-yuan.lueh@intel.com

Michał Cierniak
michal.cierniak@intel.com

Intel Corporation
2200 Mission College Blvd
Santa Clara, CA 95052

ABSTRACT

A high-performance implementation of a Java Virtual Machine¹ requires a compiler to translate Java bytecodes into native instructions, as well as an advanced garbage collector (e.g., copying or generational). When the Java heap is exhausted and the garbage collector executes, the compiler must report to the garbage collector all live object references contained in physical registers and stack locations. Typical compilers only allow certain instructions (e.g., call instructions and backward branches) to be *GC-safe*; if GC happens at some other instruction, the compiler may need to advance execution to the next GC-safe point. Until now, no one has ever attempted to make *every* compiler-generated instruction GC-safe, due to the perception that recording this information would require too much space. This kind of support could improve the GC performance in multithreaded applications. We show how to use simple compression techniques to reduce the size of the GC map to about 20% of the generated code size, a result that is competitive with the best previously published results. In addition, we extend the work of Agesen, Detlefs, and Moss, regarding the so-called “JSR Problem” (the single exception to Java’s type safety property), in a way that eliminates the need for extra runtime overhead in the generated code.

Keywords

Compilers, garbage collection, Java.

1. INTRODUCTION

As the Java language[6] matures, more and more focus is placed on performance. One of the keys to a high-performance Java Virtual Machine[10] (JVM) implementation is replacing the interpreter with a compiler (often a Just-In-Time compiler), which

translates Java bytecodes into optimized native code. Most of the code generation issues are familiar to optimizing C/C++ compiler writers, due to Java’s similarity to C and C++. However, a tricky (and unfamiliar) implementation issue is the interaction between the compiler and the garbage collector.

When the Java heap is exhausted, the garbage collector locates and collects unreachable (dead) objects. All live objects are reachable from the root set. The root set consists of all objects pointed to by global pointers (e.g., static fields of classes) or by pointers in the active stack frames. If an active stack frame belongs to a compiled method, then only the compiler can accurately determine the live references in the stack frame. Therefore, at compile time, the compiler also generates an auxiliary table of information, called a *stack map* or a *GC map*, that allows it to produce the root set at certain points within the compiled method. The GC map translates points in the compiled code into registers and stack locations containing live references. The garbage collector can then walk the stack frames, and use the GC maps to construct the complete root set.

In a single-threaded application, GC usually happens only at a call to `new()`, when there is insufficient heap to complete the allocation. Therefore, in all stack frames, the instruction pointer is at a call site. For this reason, it is only necessary for the GC map to encode the root set for call sites; i.e., only call instructions need to be *GC-safe*.

The situation is trickier for multi-threaded applications, though. When one thread exhausts the heap, the other threads could be anywhere, not just at a GC-safe site. Nonetheless, they must stop and produce their root sets as well. The standard solution is to let the threads continue execution until they reach GC-safe sites. To bound this delay, backward branches are usually made GC-safe as well.

1.1 State of the art

The simplest GC support in a Java compiler is no support at all, by using a conservative garbage collector, such as the Boehm-Weiser collector[4]. A conservative GC scans the stack and the global area, with no help from the compiler, looking for words that might possibly be object pointers. However, modern copying and generational GC algorithms[7][9] must be precise, rather than conservative, since object pointers in the root set must be updated when objects are moved; hence the required compiler support.

The typical compiler support is to make call instructions and backward branches GC-safe, and to apply compression techniques to minimize the size of the GC map. The best example in the literature is the work by Diwan, Moss, and Hudson[5], who

[†] Author’s current affiliation is Inktomi Corporation, 1900 S. Norfolk St., Suite 310, San Mateo, CA 94403

¹ All third party trademarks, tradenames, and other brands are the property of their respective owners

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. SIGPLAN ’99 (PLDI) 5/99 Atlanta, GA, USA
© 1999 ACM 1-58113-083-X/99/0004...\$5.00

modified gcc to produce GC maps for C programs, making calls and backward branches GC-safe, and compressed the maps to about 16% of the generated Vax code size.

Why not make *every* instruction GC-safe, rather than just call sites and backward branches? To the best of our knowledge, no one has attempted this before. The perception seems to be that this would be impractical, that the resulting GC maps would be too large, and that there are just too many corner-cases to deal with. In this paper, we show that a few basic compression techniques lead to Java GC maps that compress to around 20% of the generated iA32[8] code size, where every instruction is GC-safe. We believe that this is the first implementation and description of such aggressive GC support.

1.2 Our work

In our implementation of GC support, we gather preliminary GC information through two dataflow analyses on our intermediate representation (IR). The first phase is a forward pass to assign type information to every use of a register or stack location. (Actually, in our implementation, this pass is implicit, since the type information is assigned as the IR is built.) The second phase is a backward pass to calculate liveness information. (Only live references need to be included in the root set; references that are *available* but dead need not be reported.) We make use of Java's type safety guarantees to simplify the type analysis problem.

Our IR has the invariant that at code emission time, every IR instruction results in exactly one native instruction. This invariant simplifies the task of generating GC maps for which every native instruction is GC-safe. We use simple compression techniques, primarily Huffman encoding, based on offline analysis of the static properties of the code produced by our compiler. These techniques succeed in reducing the GC maps to about 20% of the code size. We have gathered statistics over a set of 7 benchmarks, representing a fairly wide range of coding styles, and have found that the statistics are quite similar across all of the benchmarks. This leads us to believe that we will see similar compression results across a much wider range of programs.

As an additional result, we extend the work of Agesen, Detlefs, and Moss[2], regarding the so-called "JSR Problem", which is the single exception to Java's type safety property. Their work shows how to recover type safety at the cost of an extra initialization of each type-unsafe variable in every execution of the method. We show how to recover the type information of such variables at GC time, with no impact on the quality of the generated code.

1.3 Organization

Section 2 describes our framework for gathering and compressing the GC map information, with Section 2.4 discussing the specific statistics of our generated code, as they relate to GC map compression. Section 3 describes our solution to the JSR problem, in a way that does not affect the quality of the generated code. Section 4 summarizes our conclusions.

2. STATISTICS AND COMPRESSION TECHNIQUES

2.1 Requirements

To provide support for GC at every instruction, the GC map must provide two types of information for every instruction. First, it must provide the set of registers and stack locations that contain

live references at that instruction. Second, it must provide the "stack adjustment" for that instruction. The stack adjustment is the value by which the current stack pointer must be adjusted to return the stack pointer to its "canonical" value. For example, if the thread is stopped just after it has pushed two arguments for a function call, then the stack pointer needs to be adjusted back by two words to reach its canonical value. In addition, if the thread is stopped in the prolog or epilog, while the new stack frame is being set up or destroyed, the stack pointer might also need to be adjusted to a canonical value. (This adjustment may not be necessary if the compiler generates code using a frame pointer, but on the iA32 architecture[8], registers are at a premium, and we prefer to free up the frame pointer for more productive uses.)

Therefore, for each instruction, the GC map must record three items: which registers (if any) changed liveness, which stack locations (if any) changed liveness, and the instruction's effect (if any) on the stack pointer.

2.2 Compression overview

Our compression mechanism is a two-level scheme. At the higher level, we exploit the fact that within a basic block, each instruction can only have a limited effect on the stack pointer or on the set of live references. Thus we encode a snapshot of the live references and the stack adjustment at the beginning of each basic block, and we encode only the changes that each instruction makes to the previous state.

The changes can be one or more of the following: beginning or end of a stack reference's live range, beginning or end of a register reference's live range, and an operation (such as a push or call instruction) that affects the stack pointer. For a push operation, we also need to encode whether a reference was pushed, because then the top of the stack is part of the root set for that instruction.

At the lower level, we use Huffman encoding to compactly encode the delta of each instruction. The parameters of the Huffman encoding are based on offline analysis of statistics gathered over a set of Java benchmarks. Section 2.4 describes these statistics.

We write the data to memory using a "sequential bit stream" data structure. We pack variable-width fields (where field widths can be an arbitrary number of bits, with no byte-alignment requirements) into a memory buffer. In this way, no bits are wasted, except perhaps a few bits at the end of the bit stream.

Sometimes an instruction has no effect on the previous root set or stack adjustment. For example, an instruction might increment an integer variable, or it might load a non-reference field into a register or a stack location. When this happens, we coalesce the sequence of instructions with no delta, followed by the next instruction with some delta, and treat the entire sequence as one large "macro-instruction" that changes the state only at the end.

Similarly, for two basic blocks that are adjacent in the final code layout, if the state at the end of the first basic block is identical to the state at the beginning of the second basic block, then there is no need to include a snapshot of the second basic block's state. Instead, we treat the two blocks as a single basic block, with only one snapshot of the state.

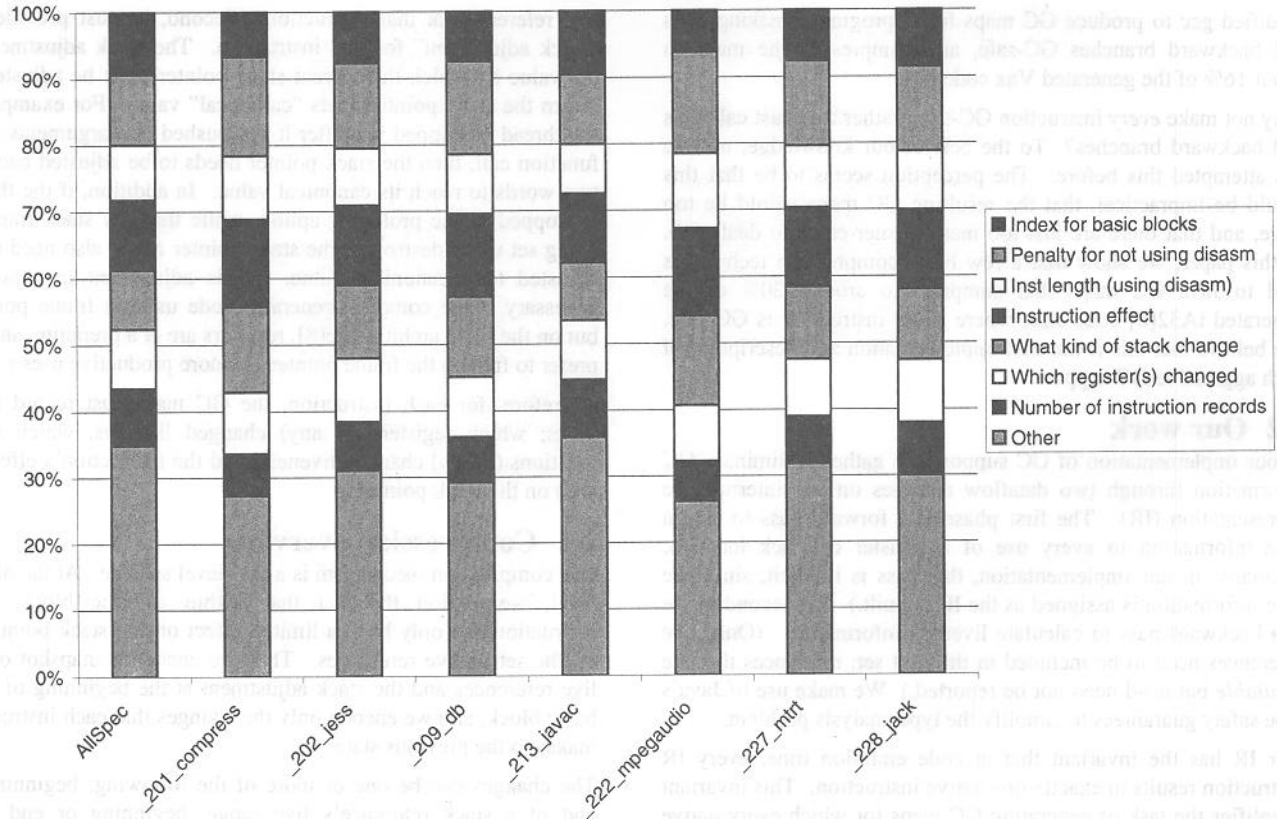


Figure 1: Total Composition

2.3 GC-unsafe instructions

There are certain instructions at which we cannot allow GC to occur, even though we still have complete root set information. These instructions involve *write barriers*. In an efficient implementation of a generational or incremental garbage collector[7][9], the GC needs to be notified whenever an object in an older generation points to an object in a younger generation. In our implementation, we first store the pointer, and then pass the two objects to the write barrier routine. If GC happens between the store and the write barrier call, the GC's state is likely to become corrupted sometime in the future, causing it to collect live objects. One possible solution is to generate a write barrier both before and after the store, but of course at a cost of double the runtime overhead. There are several other ways to deal with this problem without imposing this additional overhead (but beyond the scope of this paper), but all have one thing in common: the GC map must be able to identify such "GC-unsafe" instructions.

An additional issue is the handling of *interior pointers*. At GC time, a register might point not to the base of a live object, but to some location in the interior of the object. The compiler could generate interior pointers for loop optimizations, or for spill code. These interior pointers could be live across call sites and backward branches. The compiler must add information to the GC map describing the uses of interior pointers, and possibly how to recover the base pointer from the interior pointer. Our GC

implementation does not currently support interior pointers, so our compiler avoids generating such code.

2.4 Compression statistics

To demonstrate the results of our compression, we use a collection of programs from the SPECjvm98[11] benchmark suite. We look at statistics gathered over the execution of each individual benchmark, as well as the weighted average over all of the individual benchmarks. In addition, to factor out the effect of the system class libraries that all benchmarks share, we constructed a test harness, called "AllSpec", which simply executes each benchmark in sequence, within a single execution of the JVM. All benchmarks are compiled to native iA32 instructions.

In our measurements for this paper, we have used unmodified programs from SPECjvm98 with their full problem sizes. However, we did not fully follow the official run rules defined by the SPEC committee, so no conclusions about performance should be drawn from our experiments. The running times are presented in the Appendix. All experiments were performed on a PC with two 450MHz Intel® Pentium® II Xeon™ processors.

The table below shows the overall results of the compression for all of the tests. Code size and GC map size are reported in bytes. We evaluate the compression in terms of the ratio of the GC map size to the code size. As the table shows, our ratio is usually around 20-30%.

	Number of methods	Code size	GC map size	Ratio
AllSpec	1852	629948	172495	27.38%
_201_compress	211	149612	22920	15.32%
_202_jess	583	204645	43183	21.10%
_209_db	233	155730	25992	16.69%
_213_javac	920	330080	93385	28.29%
_222_mpegaudio	344	237470	34991	14.73%
_227_mtrt	330	184015	34691	18.85%
_228_jack	384	232761	45670	19.62%
Total	4857	2124261	473327	22.28%

Figure 1 shows the composition of the GC map for each benchmark, in terms of the proportion of the GC map occupied by the most space-intensive components of the map. The contributions of each component are the following. After describing the contributions, we provide a few examples of instructions and their effects.

- **Index for basic blocks.** Due to the nature of the encoding, decoding requires processing the GC map sequentially from the beginning, instruction by instruction, until the target instruction is reached. Depending on how often the GC map is accessed, this processing can become quite expensive. One solution for speeding up access is to embed an index into the map, allowing direct access to the start of any given basic block. The overhead for this index is shown as the top segment of the bar graph. (The table above reflects measurements without the use of the index.). An additional solution for higher performance, which can be used either with or without the index, is to implement a small cache that maps instruction addresses to root sets. This cache takes advantage of the tendency for deeper stack frames to remain unchanged from one collection to the next. Our implementation allows us to experiment with all configurations, and we find that when a cache is used, the additional performance from using the index is usually negligible.

• **Instruction length (using `disasm`).** The iA32 architecture has instructions with widely varying lengths, depending on the instructions and addressing modes used. We have to record the length of each instruction (or macro-instruction) in the GC map. Our statistics show that 91-96% of the instructions are encoded as single instructions, rather than

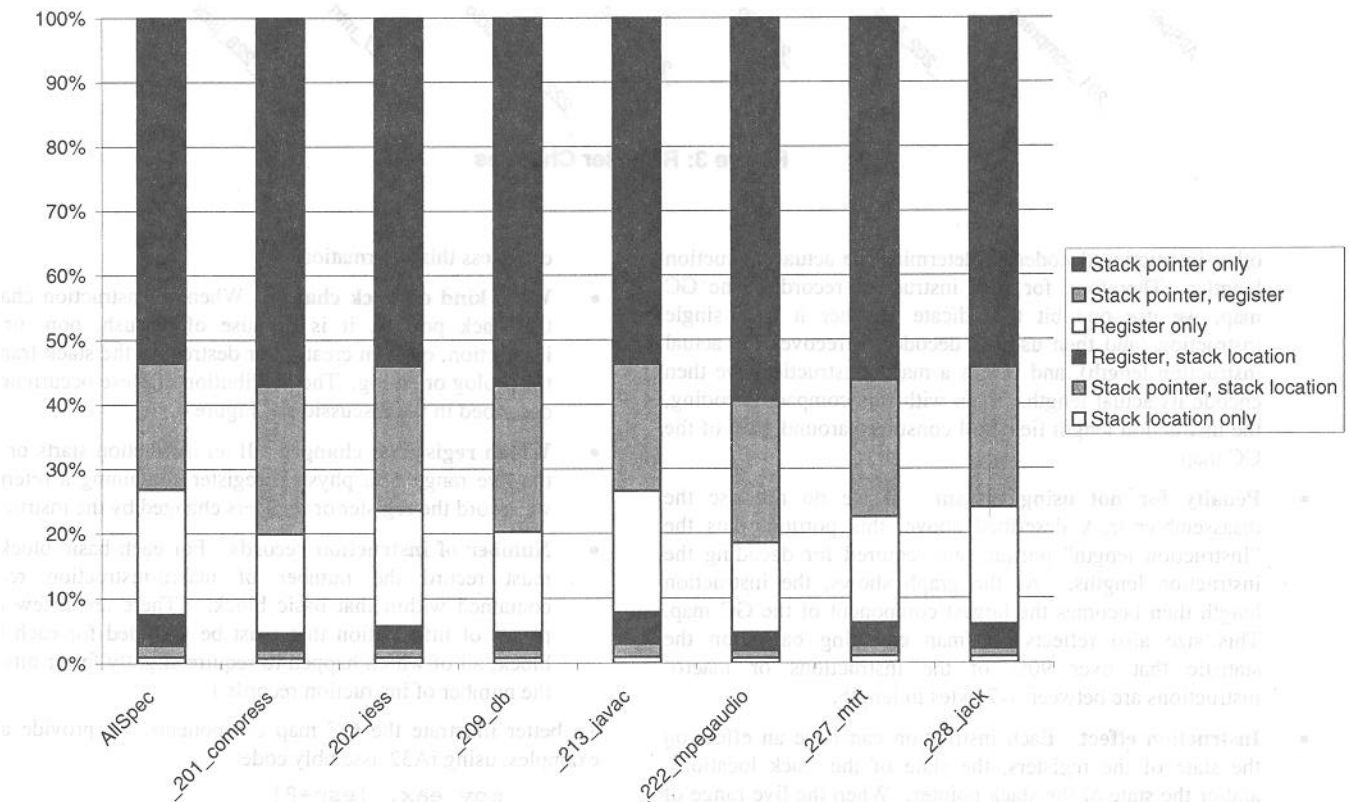
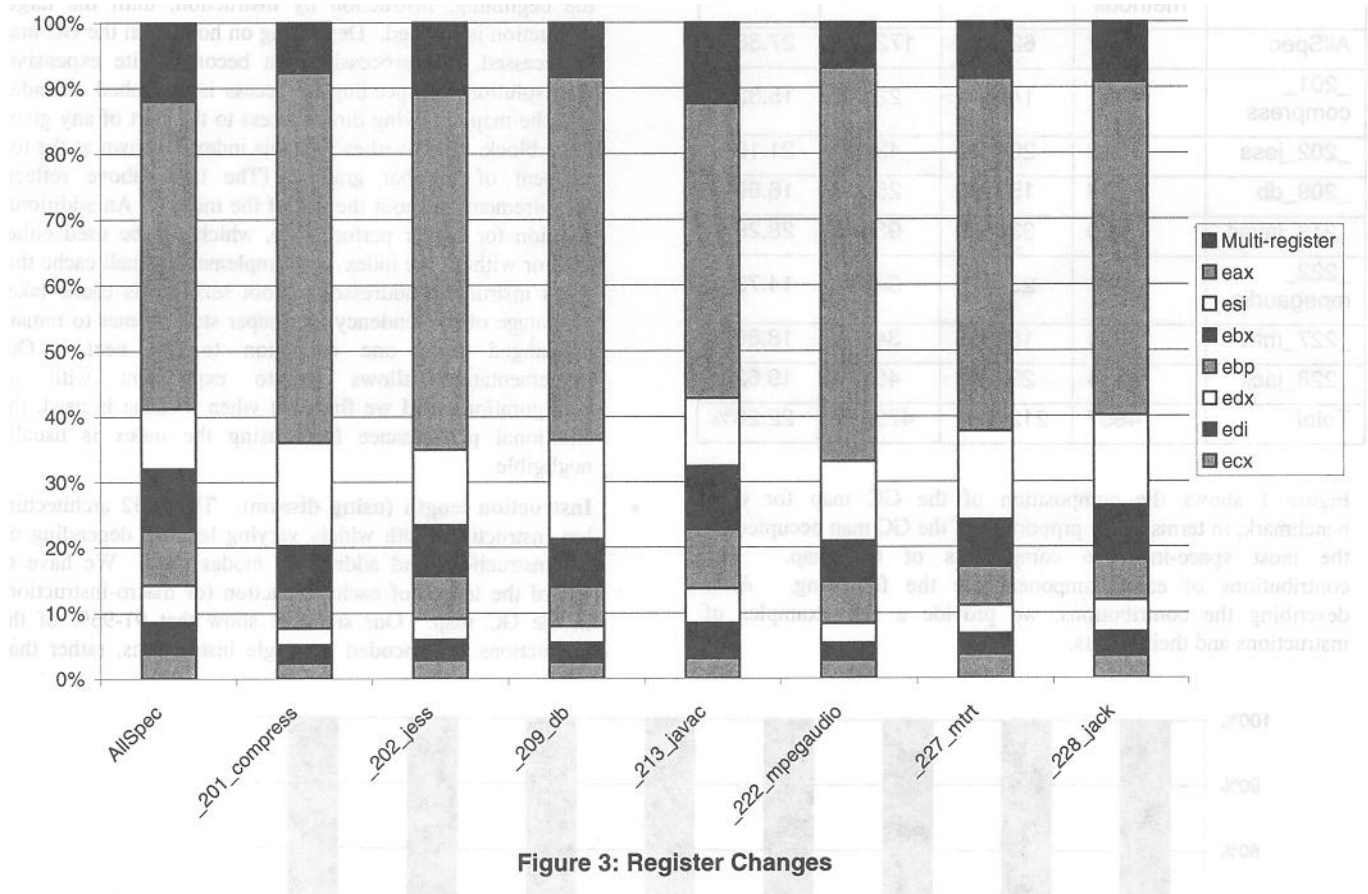


Figure 2: Instruction Effect

macro-instructions. Thus, since the native code is readily available at run time, we can use a disassembler, or some

changes. As described below, we use Huffman encoding to



other instruction decoder, to determine the actual instruction lengths. Therefore, for each instruction record in the GC map, we use one bit to indicate whether it is a single instruction (and then use the decoder to recover the actual instruction length), and if it is a macro-instruction, we then encode its actual length. Even with this compact encoding, the instruction length field still consumes around 10% of the GC map.

Penalty for not using disasm. If we do not use the disassembler trick described above, this portion, plus the “Instruction length” portion, are required for decoding the instruction lengths. As the graph shows, the instruction length then becomes the largest component of the GC map. This size also reflects Huffman encoding based on the statistic that over 90% of the instructions or macro-instructions are between 1-7 bytes in length.

Instruction effect. Each instruction can have an effect on the state of the registers, the state of the stack locations, and/or the state of the stack pointer. When the live range of a reference pointed to by a register begins or ends, the register state changes. Similarly, when the live range of a stack location containing a reference begins or ends, the stack location state changes. When an argument is pushed or popped, or a method is called, or the stack frame is constructed/destroyed in the prolog/epilog, the stack pointer

compress this information.

- **What kind of stack change.** When an instruction changes the stack pointer, it is because of a push, pop, or call instruction, or from creating or destroying the stack frame in the prolog or epilog. The distribution of these occurrences is described in the discussion of Figure 4.
- **Which register(s) changed.** If an instruction starts or ends the live range of a physical register containing a reference, we record the register or registers changed by the instruction.
- **Number of instruction records.** For each basic block, we must record the number of macro-instruction records contained within that basic block. (There are a few other pieces of information that must be recorded for each basic block, all of which happen to require slightly fewer bits than the number of instruction records.)

To better illustrate the GC map components, we provide a few examples, using iA32 assembly code:

```
mov eax, [esp+8]
```

Assume [which is a stack location, holds a reference, and that this instruction is the last use of the stack location. This instruction affects the register state, since **eax** now contains a reference (this instruction starts **eax**’s live range). In addition,

the stack location's live range ends. There is no stack pointer change in this instruction.

`push ecx`

Assume `ecx` contains a non-reference. Its live range may end in this instruction, but we do not record it in the GC map because it is not a reference. In this case, only the stack pointer changes in this instruction, and we record it as a push of a non-reference.

`push [esp+8]`

Assume `[esp+8]` holds a reference, and that this is its last use. This instruction does not affect the register state. It does affect the stack pointer (push of a reference), and it affects a stack location because the live range of `[esp+8]` ends.

`mov eax, [ebx+8]`

This is the kind of code typically generated for a `get` bytecode. Assume it is a `getf` of a reference, and that it is the last use of `ebx`, which holds the base pointer for the `getf`. In this case, there is no effect on the stack pointer or the stack locations, but there is a change to the register state. In particular, the live range of `eax` begins and the live range of `ebx` ends. This is one of the rare cases where more than one register's state changes in a single instruction.

The top two bars of the graph in Figure 1 show the overhead required to provide fast access to the GC map. This fast access comes at a price—increasing the GC map size by **25%**—and may be unnecessary when we use the simple caching mechanism described above. Note that the table listing the compressed GC map sizes reflects the GC maps without this additional overhead, so the numbers in the table are the best numbers we can report.

The largest single component of the GC map is the instruction effect. In any given iA32 instruction, the register liveness state

can change, the state of a stack location can change, and the stack pointer can change. At least one of the three must occur, but all three cannot occur in a single instruction. Figure 2 shows the frequency of each combination. Based on these frequencies, we construct a Huffman encoding, rather than always using three bits per instruction.

Figure 3 shows the distribution of register liveness changes. Over 95% of the time, an instruction affects the live range of only a single register containing a reference. Interestingly, the majority of the single-register changes involve the `eax` register, primarily because the calling conventions specify that when a method returns a reference, it is returned in the `eax` register.

Figure 4 shows the distribution of stack pointer changes. About 95% of the instructions that affect the stack pointer are evenly distributed between pushing a reference, pushing a non-reference, and a method call. (A method call simply indicates that the instruction has the effect of resetting the stack pointer to its default location.) The other possibilities are a pop instruction (generally found only in the epilog of a method), and an arbitrary stack pointer adjustment, such as setting up or destroying a stack frame in the prolog or epilog.

We need to record the stack pointer changes for two reasons. First, when stack locations contain live references, we identify the locations with respect to an offset from a “canonical” stack pointer value, so we must be sure the stack pointer value is correct before reporting live stack locations. Second, when we unwind to the calling stack frame, we must keep the stack pointer consistent with the JVM's stack unwinding conventions.

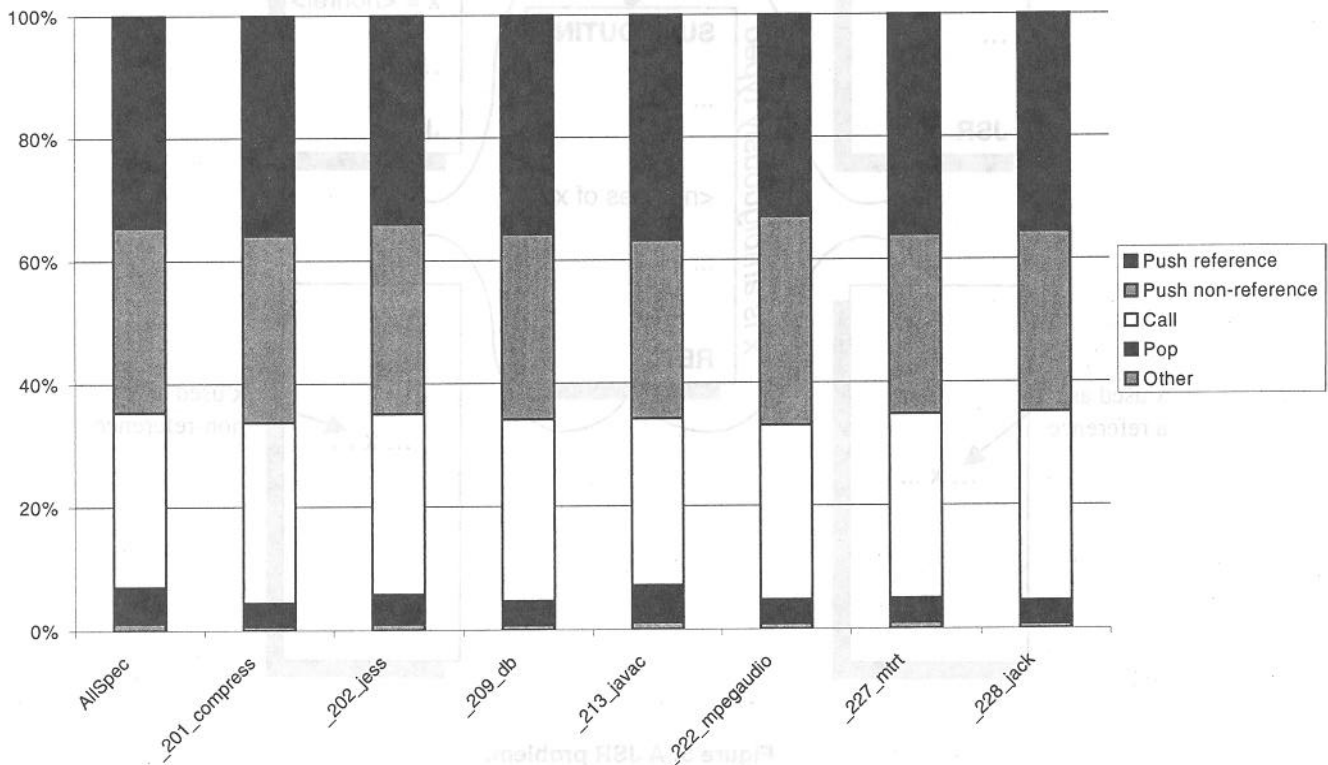


Figure 4: Stack Pointer Changes

2.5 Discussion

From observing the graphs, we see that there is a remarkable similarity across all of the benchmarks in all of the components of the GC map. This uniformity suggests that we can achieve near-optimal compression by embedding the compression and decompression algorithms completely within the compiler, without using dynamic approaches such as embedding dictionaries in the GC map.

However, the particular statistics we gathered are only valid within our compiler. Different code generation strategies will result in different distributions of instruction effects, register changes, etc. For example:

- Modifications to the register allocation algorithm will almost certainly change the distribution of register changes and of the instruction effect.
- Our measurements reflect a particular policy (and implementation) for inlining of methods. Different policies for inlining are likely to affect all of the statistics.
- The statistics also reflect specific calling conventions in the JVM. In particular, all arguments are passed through the stack, and return values are usually returned in specific registers. Changing the calling conventions (e.g., passing some arguments in registers) will change the statistics.

These statistics were gathered specifically for the iA32 architecture. Nonetheless, we expect the behavior to remain roughly the same across other architectures, in the sense that GC maps can be compressed to around 20% of the generated code size. The most striking difference between iA32 and other architectures is that iA32 has only 7 registers available (not including the floating-point stack, which cannot contain references). If the architecture has more registers, it will require more bits to encode the register changes in the GC map. However, it will require the same number of bits to record the register in the actual instruction, so the ratio of the GC map to the code size should remain balanced.

Another architectural difference is that iA32 has a wide variance in the length in bytes of each instruction. As such, encoding the instruction lengths in the GC map requires a large fraction of the total space (unless we use the disassembler trick described above). If the target architecture has more uniform instruction lengths, we would expect the compression to improve.

3. SOLVING THE JSR PROBLEM

Java is a type-safe language. In particular, the compiler can analyze a method and statically determine whether each variable is a reference or a non-reference, and thus which variables contain live references at GC time. However, when a Java program is compiled into Java bytecodes, some of this type safety is lost. The

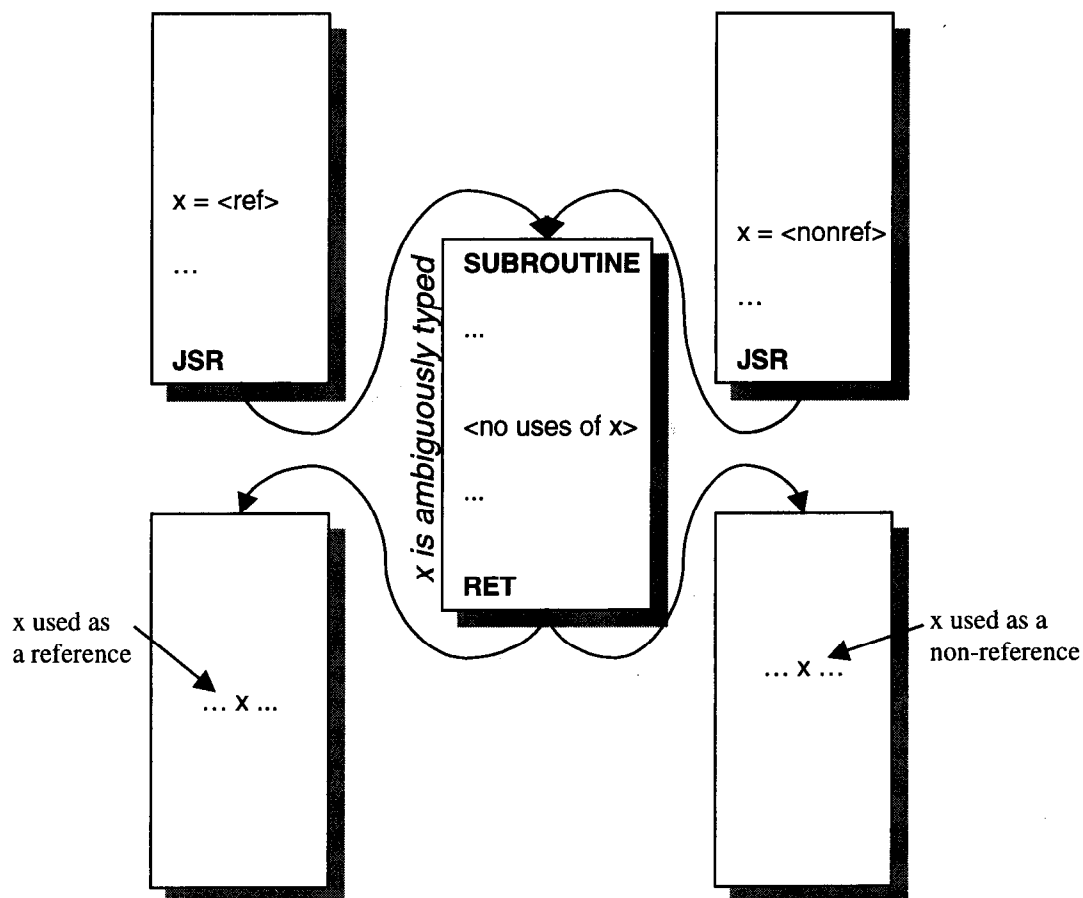


Figure 5: A JSR problem.

Java bytecodes contain a pair of instructions, JSR and RET, which Javac uses when compiling a `finally` clause. JSR/RET define an intra-method subroutine call, without creating a new Java stack frame. The JVM allows a low-level variable to contain a reference on one path into the subroutine, and a non-reference on another path into the subroutine, provided that the subroutine does not attempt to access the variable, as pictured in Figure 5 (see [10], Section 4.9.6). This rule appears to preserve the type safety property, since the subroutine contains no explicit references to ambiguously-typed variables. However, if GC happens within such a subroutine, there is an implicit use of the variable, and the compiler cannot statically determine whether that variable contains a reference, and thus whether to report it to the garbage collector. We refer to this situation as the “JSR Problem”.

Agese, Detlefs, and Moss[2] describe a method for rewriting the Java bytecodes to remove this type ambiguity. Their approach has two parts. In the first part, they prevent ambiguity between reference and non-reference values by “splitting” an ambiguously-typed variable into a reference variable and a non-reference variable. The semantics of the Java bytecodes guarantee that this transformation is always correct. However, during GC, it is still possible that the variable contains a live reference on one path into the subroutine, but is uninitialized on another path. The variable can only be reported to the garbage collector if it has been initialized. Hence the second part of their solution: explicitly initialize the variable to `null` on every path into the subroutine for which the variable is uninitialized.

Another approach[1] involves maintaining explicit runtime type information for all ambiguously-typed variables. This involves an initialization of a bitmask at method entry, and an update of the bitmask every time a value is stored into the variable. The approach is extremely simple to implement, but requires even more runtime overhead.

Although methods exhibiting the JSR problem are rare, we still object to the overhead of additional initializations. In this section, we describe our method for determining at GC time whether an ambiguously-typed variable contains a reference or a non-reference, without affecting the generated code. Our approach also does not require variable splitting.

Our approach contains two parts, described in detail below. The first part is compile-time analysis, and the second part is runtime recovery of the type information. At compile time, we determine which variables contain ambiguously-typed variables in each basic block, and in every basic block, we record the location of the JSR return address for the current subroutine. We then use runtime analysis to recover the type information for ambiguously-typed variables at GC time. If our compile-time analysis identified an ambiguously-typed variable, we simply revert to the variable’s type information after the return from the subroutine. We continue this analysis recursively for nested subroutines, until we find a point where the variable’s type is statically known. This compile-time analysis and runtime recovery allows us to determine precise type information for all variables, without requiring variable splitting or additional overhead in the compiled code.

3.1 Compile-time analysis

We start by constructing the control-flow graph (CFG). We then perform a forward dataflow analysis[3] over the CFG to identify subroutines. For every basic block, we record the subroutine that contains the block. (If there are nested subroutines, we record the innermost subroutine that contains the basic block.) We also ensure that every time a variable is initialized or referenced, we record type information at that instruction (regardless of whether the variable is used as a reference or a non-reference). This type information can be computed incrementally as the IR is built, or through a forward dataflow analysis algorithm.

Next, we compute variable liveness information through a backward dataflow analysis. By extending this liveness analysis as described below, we discover every occurrence of ambiguously-typed variables. In the standard algorithm, we compute the liveness at the end of a basic block, then iterate backwards through the basic block and update the liveness for every instruction. The set of live variables at the end of the basic block is the union of the sets of live variables at the beginning of every successor basic block.

We extend the liveness algorithm in two ways. First, we subdivide the set of live variables into four sets: live references, live non-references, live “unknowns” (i.e., the ambiguously-typed variables), and dead variables. Second, we use three different sets of equations for merging the liveness sets, depending on the context of the basic block within the CFG.

In defining the equations, we define R_n , N_n , U_n , and D_n to be the sets of live reference, non-reference, unknown, and dead variables, respectively, in basic block n . We also define $\text{succ}(n)$ to be the set of nodes in the CFG that are successors of n . For every node n , the sets R_n , N_n , U_n , and D_n are disjoint, and the union of the four sets is the set of all variables ever used in the function.

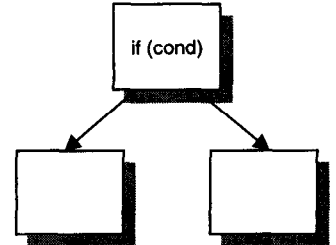
3.1.1 Merging at a conditional branch

$$D_n = \bigcap_{s \in \text{succ}(n)} D_s$$

$$R_n = (\bigcap_{s \in \text{succ}(n)} R_s + D_s) - D_n$$

$$N_n = (\bigcap_{s \in \text{succ}(n)} N_s + D_s) - D_n$$

$$U_n = \text{all} - (R_n \cup N_n \cup D_n)$$



In other words, the dead variables are the ones that are dead in all successors. The live references are the variables that are either live references or dead in all successors (but live in at least one successor). The live non-references are the variables that are either live non-references or dead in all successors (but live in at least one successor). The live unknowns are all other variables.

For this kind of merge, the JVM specification makes it illegal for a variable to be a live reference at the beginning of one successor and a live non-reference at the beginning of another successor. Similarly, a variable cannot be a live unknown at one successor and a live reference or non-reference at another successor.

3.1.2 Merging at an indirect branch (RET)

$$D_n = \bigcap_{s \in \text{succ}(n)} D_s$$

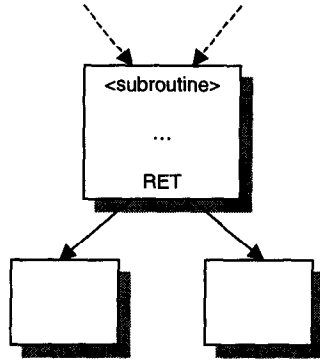
$$R_n = \bigcap_{s \in \text{succ}(n)} R_s$$

$$N_n = \bigcap_{s \in \text{succ}(n)} N_s$$

$$U_n = \text{all} - (R_n \cup N_n \cup D_n)$$

In other words, for a variable to be dead, a live reference, or a live non-reference, it must be consistently dead, a live reference, or a live non-reference, respectively, in every successor. Otherwise, it is a live unknown.

Note that because of the JVM specification, all possible targets of this indirect branch are statically known at compile time.



3.1.3 Merging at a JSR

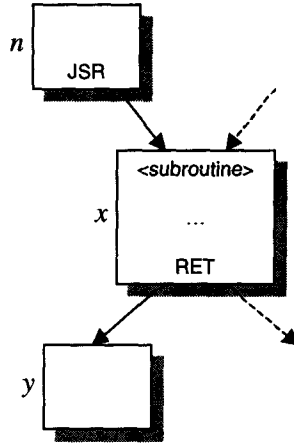
When a basic block ends with a JSR instruction, it has two successors in the CFG: the beginning of the subroutine (call it node x), and the code immediately following the JSR instruction (call it node y). Node x is an explicit successor in the CFG, whereas node y is only an implicit successor.

$$D_n = D_x \cup (U_x \cap D_y)$$

$$R_n = R_x \cup (U_x \cap R_y)$$

$$N_n = N_x \cup (U_x \cap N_y)$$

$$U_n = U_x \cap U_y$$



If a variable is a live unknown at the beginning of the subroutine, then it could not have been accessed within the subroutine. Therefore, the variable's type at the JSR instruction must be the same as its type in the instruction immediately following the JSR. On the other hand, if the variable has a known type at the beginning of the subroutine, then that type is propagated unchanged to the JSR instruction.

3.2 Runtime recovery

At compile time, in the GC map, the compiler records the liveness information at the beginning of every basic block. It records which variables contain live references, and which variables contain live unknowns. In addition, when the basic block is contained within a subroutine, the compiler records where the subroutine's return address can be found.

When GC happens at a particular instruction at run time, the compiler decodes the GC map and determines which variables contain live references at the instruction, and which variables contain live unknowns. For every live unknown, the compiler must determine whether or not it contains a live reference. If

there are any live unknowns, the compiler first locates the basic block corresponding to the subroutine's return address. Then it looks at the set of live references and live unknowns at the beginning of that basic block. If the variable is unknown within the subroutine and is a live reference outside the subroutine, then it is reported as a live reference. If the variable is unknown within the subroutine and neither unknown nor a live reference outside the subroutine, then the variable is not reported as a live reference. However, if the variable is unknown both within the subroutine and outside the subroutine, then the subroutine must be nested, and the compiler continues this analysis recursively until it is no longer looking at a basic block contained within any subroutine.

4. CONCLUSIONS

In this paper, we have shown that it is feasible to provide support in a Java compiler to make every instruction GC-safe. This kind of support can decrease the GC latency in a multithreaded application (however, we currently do not have multithreaded benchmarks and timings to prove or disprove our conjecture), and can simplify the design of a JVM. We have shown that the resulting GC maps can be compressed to a quite acceptable 20% of the generated code size, on average. We do not claim that this ratio is the smallest possible, only that ratios of this size or smaller are achievable through simple compression techniques. This result debunks the widely held myth that making every instruction GC-safe would be too expensive.

Our results were gathered on a collection of 7 benchmarks, written in a fairly wide variety of coding styles. Nonetheless, our results show an interesting trend: across all the benchmarks, the generated code tends to have approximately the same statistical properties, in the attributes required for the GC map. This uniformity suggests that simple offline analysis can yield a near-optimal compression algorithm. This analysis can be repeated and revised for different compilers and different target architectures.

In addition, we have shown how to work around the single type-unsafe property of the Java bytecodes, the "JSR problem". Our solution adds information to the GC map that allows the compiler to recover the type of an ambiguously-typed variable at GC time, without any restrictions or additional overhead in the generated code.

5. REFERENCES

- [1] A. Adl-Tabatabai, M. Ciermiak, G.-Y. Luch, V.M. Parikh, and J.M. Stichnoth. Fast, Effective Code Generation in a Just-In-Time Java Compiler. In proceedings of PLDI'98, Montreal, May 1998, pp. 280-290.
- [2] O. Agesen, D. Detlefs, and J.E.B. Moss. Garbage Collection and Local Variable Type-Precision and Liveness in Java Virtual Machines. In proceedings of PLDI'98, Montreal, May 1998, pp. 269-279.
- [3] A. V. Aho, R. Sethi, and J. Ullman. Compilers: Principles, Techniques, and Tools. Addison-Wesley, Reading, MA, second edition, 1986.
- [4] H.-J. Boehm and M. Weiser. Garbage Collection in an Uncooperative Environment. Software, Practice and Experience, September 1988, pp. 807-820.
- [5] A. Diwan, J.E.B. Moss, and R. Hudson. Compiler Support for Garbage Collection in a Statically Typed Language. In

proceedings of PLDI'92, San Francisco, CA, June 1992, pp. 273-282.

- [6] J. Gosling, B. Joy and G. Steele. The Java Language Specification. Addison-Wesley, 1996.
- [7] R. Hudson and J.E.B. Moss. Incremental Collection of Mature Objects. In proceedings of International Workshop on Memory Management, volume 637 of Lecture Notes in Computer Science, St. Malo, France, September 1992. Springer-Verlag.
- [8] Intel Corp. Intel Architecture Software Developer's Manual, order number 243192. 1997.
- [9] R. Jones and R. Lins. Garbage Collection. John Wiley & Sons, 1996.
- [10] T. Lindholm and F. Yellin. The Java Virtual Machine Specification. Addison-Wesley, 1996.
- [11] Standard Performance Evaluation Corporation.
<http://www.spec.org/osg/jvm98>

APPENDIX

Performance of the individual SpecJVM98 benchmarks, measured on a PC with two 450MHz Intel® Pentium® II Xeon™ processors.

Benchmark	Time [s]
_201_compress	21.7
_202_jess	15.3
_209_db	57.4
_213_javac	27.0
_222_mpegaudio	19.6
_227_mtrt	8.9
_228_jack	22.9