GC Points in a Threaded Environment

Ole Agesen

GC Points in a Threaded Environment

Ole Agesen

SMLI TR-98-70 December 1998

Abstract:

Many garbage-collected systems, including most that involve a stop-the-world phase, restrict GC to so-called GC points. In single-threaded environments, GC points carry no overhead: when a GC must be done, the single thread is already at a GC point. In multi-threaded environments, however, only the thread that triggers the GC by failing an allocation will be at a GC point. Other threads must be *rolled forward* to their next GC point before the GC can take place. We compare, in the context of a high-performance Java™ virtual machine, two approaches to advancing threads to a GC point, polling and code patching, while keeping all other factors constant. Code patching outperforms polling by an average of 4.7% and sometimes by as much as 11.2%, while costing only slightly more compiled code space. Put differently, since most programs spend less than 1/5 of the time in GC, a 4.7% bottom-line speedup amounts to more than a 20% reduction in the *GC-related costs*. Patching is, however, more difficult to implement.

M/S MTV29-01
901 San Antonio Road
Palo Alto, CA 94303-4900

email address:
ole.agesen@east.sun.com

For information regarding the SML Technical Report Series, contact Jeanie Treichel, Editor-in-Chief <jeanie.treichel@eng.sun.com>.

GC Points in a Threaded Environment

Ole Agesen
Sun Microsystems Laboratories
One Network Drive
Burlington, MA 01803-0903

1 Introduction

Many garbage collection (GC) algorithms copy (or move) objects to reduce fragmentation and/or to increase performance. For example, to collect an area of memory where one expects the majority of objects to be dead (garbage), the most efficient approach could be one that evacuates the remaining live objects from the area by copying them to some other place and then returns the entire area to the free pool of memory. The full story, of course, is more complicated: when objects are copied, all references to them, whether in (live) objects, local variables, global variables or registers, must be adjusted to point to the new locations. Thus, the garbage collector must be able to locate all references in the system *exactly*, since if just one reference to a copied object is left unadjusted or one integer that happens to look like a reference is updated, the program execution could go wrong.

There are two basic approaches to supporting exact GC: tagging and reference maps. *Tagging* reserves one or more bits in each word-sized data item to indicate whether the word represents a reference or a non-reference. *Reference maps*, or just *maps* for brevity, associate a data structure with an area of memory that contains a mixture of references and non-references. To determine if a given word is a reference, the garbage collector consults the map. In their simplest form, maps may consist of a bit vector and one may think of them as external tag bits.

Each of the two approaches has its advantages and disadvantages. Tagging is simple but reduces the range of integers and the precision of floating point numbers and incurs overhead on mutators to clear the tag bits from operands and reinsert them into results. (A well-designed system of tag bits will attempt to minimize these disadvantages by carefully choosing the number, location, and sense of tag bits.) Reference maps, in contrast, retain full range and precision, but incur storage overhead and complicate the system, since the compiler must emit extra information and the runtime system must maintain bindings from areas of memory to maps.

Even systems that rely mostly on tagging often use reference maps for some structures. For example, the Self system uses tag bits in the heap and for parts of stack frames but reference maps to describe the contents of CPU registers. The EVM[1], a high-performance Java™ virtual machine for the Solaris™ operating system, relies exclusively on reference maps. This choice is natural since the Java programming language demands 32 and 64 bit integers and floating point numbers, leaving no bits for tagging on most contemporary hardware. In more detail, two forms of reference maps are used:

1. EVM, known previously as ExactVM, is embedded in Sun's Java 2 SDK, available at http://www.sun.com/solaris/java/.

- *object maps:* in the Java programming language, every object field has a static type that determines, for the lifetime of the object, whether the field contains a reference. This makes it possible to store a constant object map in each class, giving the garbage collector easy access to the layout of any object since objects contain a pointer to their class.

- *stack frame maps* cover a stack frame and the CPU registers. Stack frames differ from objects in that they change layout as the computation proceeds. For example, on entry to a method, most of the slots in its frame will contain uninitialized values (non-references); as the computation progresses, some slots will change to contain references, and when the method returns, most of the slots will again have reverted back to being non-reference slots (assuming that only *live* values are reported as references in the stack frame maps [2]). To reduce the storage overhead, stack frame maps commonly are only kept for certain so-called *GC points* in each method. This means that GC can only be performed when all mutator threads have reached GC points.

In single-threaded environments, GC points work extremely well. One simply has to make allocation points be GC points to ensure the availability of a map in the active frame and make call sites be GC points to ensure the availability of maps for frames deeper in the stack. The GC points incur no mutator overhead: allocations have to check that enough space is free anyway, and when this check fails and a GC is initiated, the single mutator thread is automatically at a GC point.

This ideal situation extends to multi-threaded systems in which only a single thread executes at a time and context switching is restricted to occur only at GC points, but, unfortunately, does not apply to unrestricted multi-threaded environments. In the latter class of systems, when one mutator thread attempts an allocation that exceeds the available free space, all we know is that *this* mutator is at a GC point. Before the GC can be performed, however, we must ensure that *all* mutator threads are at GC points. The latter does not come for free. The most common solution involves advancing the other threads to GC points by:

- increasing the number of GC points so that they are densely spaced in time, preventing mutators from executing for an unbounded period of time without reaching GC points. Typically, this involves adding a GC point to any loop that would not otherwise contain one, i.e., any loop that contains no allocation or calls.

- providing a mechanism for suspending mutators when they reach a GC point.

The primary contributions of this paper are a detailed description of how mutator threads and GC coordinate, and a quantitative comparison of two different techniques, *polling* and *code patching*, for suspending mutators at GC points in the context of a high-performance, multi-threaded Java virtual machine. The opportunity to compare the two techniques came about as EVM was recently changed from using polling to suspend at GC points to using code patching. Thus, we have an occasion to compare the two approaches while keeping all other factors constant. Patching is similar to debuggers' use of break points to suspend execution at a given location. Our poll-less GC design and implementation grew out of the collaborative work of our team at Sun, notably Mario Wolczko, Ross Knippel, Bill Bush, and the author, and it benefited from discussions with our Sun colleague Lars Bak related to his similar work.

Our results show that on the SPARC™ version of EVM, code patching outperforms polling by 0.6% - 11.2% in terms of execution speed (instruction counts), but increases code space by 5.1% -

8.5%, and is somewhat harder to implement correctly (mainly because it interacts with fundamental properties of the architecture such as registers and delay slots).

The focus in this paper is stop-the-world GC algorithms that require mutator threads to be suspended while garbage collection takes place. Other GC algorithms in which mutator threads run concurrently with the garbage collector most of the time, still require short periods of no mutator activity. An example is the "mostly parallel garbage collector" by Boehm, Demers and Shenker [3]. Thus, the results presented in this paper apply more broadly than stop-the-world GC.

The rest of this paper is organized as follows. Section 2 presents in detail the thread suspension process and GC points in EVM. The key aspect of the suspension process is the roll-forward, which allows the garbage collector to nudge a thread forward to its next GC point where it will be stopped either by polling code or patched code. In Section 3, we measure total execution time and code space on a set of benchmark programs, showing that code patching always runs faster than polling, but uses more compiled code space. Section 4 briefly discusses an alternative to code patching, an optimized poll sequence, and puts the achieved speedup into perspective. Subsequently, Section 5 discusses related work, and Section 6 summarizes and concludes.

2 The thread suspension process in EVM

In the default configuration, EVM uses a two-generation memory system. The young generation employs a semispace copying collector, typically configured with 2 Mb in each semispace, and the old generation employs a mark-compact collector within a single contiguous space. All but the largest objects are allocated in the youngest generation. The garbage collector tenures objects from the young generation into the old generation once their age, measured in terms of young generation collections, reaches the current tenuring threshold. The threshold is adjusted after each young generation collection by monitoring the survival rate, as suggested by Ungar [18]. Further information about the memory system in EVM can be found in [1, 2, 19].

The principal goal of EVM is high performance for multi-threaded programs executing on hardware with multiple CPUs. To achieve sequential efficiency, EVM uses a combination of interpretation and compilation to execute the bytecode format defined by the Java platform ("Java bytecode"). The interpreter executes methods that are not performance critical, such as methods without loops and methods that are invoked infrequently, while an optimizing just-in-time compiler translates the remaining methods into machine code. To achieve parallel efficiency, EVM maps threads defined by the Java platform ("Java threads") to Solaris threads [9] while the synchronization operations are mapped directly to hardware primitives (compare-and-swap and swap instructions). EVM runs on both x86 and SPARC processors, although in this paper we will describe the SPARC version only.

2.1 Co-existence of low-level C code and garbage collection

In EVM, as in most implementations of garbage-collected programming languages, two kinds of code must co-exist: the low-level implementation code (C), also known as native code, and the target code (Java bytecode). For Java virtual machines, the amount of native code tends to be large, since the core class libraries have a fairly broad interface to the virtual machine (e.g., facilities such as reflection, class loading, and threads are all to a large degree implemented in the core

virtual machine). The large amount of native code, combined with the general complexity of operating in a multi-threaded environment, can easily lead to errors, unless a strict discipline is followed. In EVM, this discipline consists of two interface layers, implemented using C preprocessor macros and functions, that allow C code to operate safely on garbage collected objects ("Java objects").

The *direct pointer layer* lets C code use direct pointers to operate on Java objects as if they are C structs. This layer provides all the basic operations on objects, including allocation, field access, modification, and hashing, and serves to encapsulate object layout such as the location and size of fields, class pointers, array lengths, etc. The indirect layer, named *LLNI* for *low-level native interface*, is implemented on top of the direct layer. It supports the same operations as the direct layer, while adding two features: it provides a means for the memory system to keep track exactly of all locations containing direct pointers to Java objects (thus, fully enabling object relocation), and it synchronizes object manipulation operations with the garbage collector to ensure that both mutator and garbage collector threads always see a consistent view of memory.

The direct pointer layer provides maximally efficient access to objects but it is unrealistic to use it throughout the VM. It is essential that it can be used in the most performance critical parts such as the bytecode interpreter loop, but for the majority of the code in the virtual machine, a slightly less efficient but more convenient object interface is preferable. LLNI was designed to add convenience while minimizing performance loss. First, let us review the significant drawbacks that clients of the direct pointer layer would face:

- The collector must know the location of all direct pointers to Java objects so that they can be updated when the objects move, or the collector must be prevented from moving (or reclaiming) objects that are referenced by pointers of which it has no knowledge. Hence, either the client must meticulously register every location containing a pointer to a Java object (and it is not obvious how to do this when passing pointers into and out of functions) or ruthlessly pin objects while holding on to such pointers (the ramifications of pinning even a few objects temporarily can be significant for many GC algorithms).

- Even if the collector knows about a pointer and updates it, a problem can occur if the collector relocates an object "while" the object is being modified. Conceivably, a modification such as

  ```
  ptr->fieldName = newContents;
  ```

  involves computing `ptr + offset` and then storing into that address. In the worst case, a garbage collection will happen and the object gets relocated between computing this address and performing the store.

The LLNI layer overcomes the former problem by introducing an extra layer of indirection and the latter problem by (efficiently) synchronizing object access with the garbage collector. Figure 1 shows the use of registered indirection cells to allow the GC to track all direct object pointers. Once the indirection cell has been registered with the garbage collector, the indirect pointer, which we refer to as an LLNI handle, can safely be passed from function to function, stored, or returned. (It should be emphasized that even though the LLNI layer uses indirect pointers, Java objects still reference each other directly, and local variables in Java bytecode stack frames operate with direct references. We also note that LLNI handles share only the indirection property with the "handles" described in the JVM specification [10]; LLNI handles are not necessarily in

one-to-one correspondence with objects and they can reference different objects at different times.)

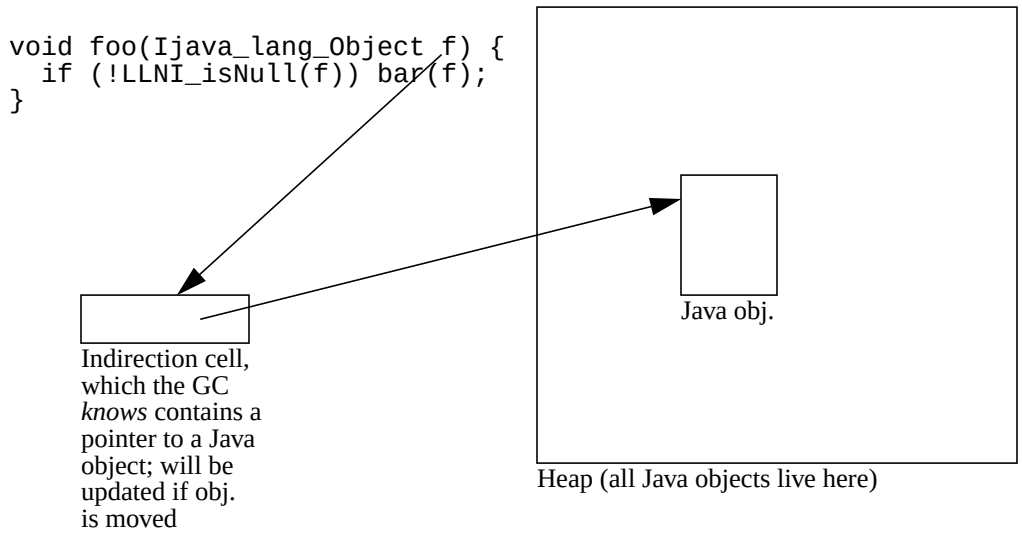

Figure 1. LLNI uses indirect pointers, registering the indirection cell with the garbage collector.

LLNI overcomes the second problem by synchronizing with the garbage collector to ensure that objects are not moved while the operation is performed. Several implementations of this synchronization are possible, with various efficiency/complexity trade-offs. The most important technique uses a thread-local counter, `inconsistentCount`. When this counter is greater than zero, the mutator thread is *inconsistent*. When `inconsistentCount` is zero, the mutator is *consistent.* An inconsistent mutator signals to the garbage collector that it—temporarily—cannot tolerate GC. For a given mutator thread, the counter reveals the number of on-going LLNI operations (some operations can nest, although most cannot). Now, an operation to set a field in an object,

```
LLNI_writeRefField(object, fieldName, newValue)
```

may be implemented as follows:

```
1. threadLocal->inconsistentCount++; /* Operation starts: GC cannot be tolerated. */
2. (*object)->fieldName = newValue;
3. threadLocal->inconsistentCount--; /* Operation ends; GC OK as far as this op. goes. */
4. if (threadLocal->inconsistentCount == 0 && gc_stop_threads)
5.   wait_for_gc_end();
```

The statement in line 1 causes the mutator to become inconsistent before executing the GC-sensitive operation in line 2. Here we have written out the pointer dereference and field access for specificity, but in reality the body of an LLNI operation is usually an invocation of the corresponding direct pointer operation. Lines 3 to 5 return the thread to the consistent state (the role of the `if` statement will be explained in Section 2.2). The code delineated by the increment and decrement of `inconsistentCount` is called an *inconsistent region*. In the example above, line 2 is an inconsistent region.

In general, inconsistent regions must execute in bounded time because while one thread is inconsistent, garbage collection is impossible, potentially blocking progress of all threads in the system until the inconsistent thread becomes consistent again. However, indefinite inconsistent regions can be permitted, as long as the thread periodically (and frequently) *offers* to suspend for GC. We use this relaxation to allow threads to execute inconsistently in compiled code and in the bytecode interpreter loop, *offering* to become consistent at each GC point. Being able to assume no interference from GC and no object movement, except at the well-defined GC points, allows more efficient execution with direct pointers in the performance-critical interpreter loop and compiled code.

For completeness, though it is not directly relevant to the present paper, let us mention that the LLNI layer in EVM is used to implement the Java platform's public interface for native code (JNI). For the duration of JNI calls, except for short routines that are implemented using LLNI, the mutator thread remains consistent and is therefore able to tolerate GC at most times. In this manner, the support for exact GC provided by LLNI carries over to arbitrary native code written by users.

In summary, LLNI affords convenience and exactness to C code operating on garbage-collected objects by using registered indirection cells to refer to objects and synchronizing with the garbage collector by means of short-duration lightweight inconsistent regions.

2.2 Common GC suspension steps

We can now describe the sequence of steps that take place when a garbage collection is performed. These steps, except for the differences explained in Section 2.3 and 2.4, are the same whether polling or code patching is used.

1. A new thread, the *GC thread*, is created to perform the GC (use of a separate thread ensures that a predictable amount of stack space is available, regardless of the depth of the stack of the mutator thread that triggers GC).

2. The GC thread suspends all the mutator threads and sets the global boolean `gc_stop_threads` to true.

3. The GC thread iterates over all mutator threads. For each mutator *T*, if *T* is consistent, it is already able to tolerate a GC, so it is simply left suspended. If *T* is inconsistent, the GC thread restarts *T*. There are now two possibilities: *T* is executing in an inconsistent region, in which case it will suspend itself at the end of the region, or *T* is executing in compiled code or in the bytecode interpreter loop, in which case *T* will suspend itself when it reaches the next GC point (where it offers to become consistent).

4. The GC thread waits for all the restarted inconsistent mutators to reach GC points and suspend themselves (monitoring progress with a counter that the mutators increment just before they suspend themselves).

5. Now all mutators in the system have been stopped either in a consistent state or at a GC point, so the garbage collection can be done.

6. Finally, the GC thread resumes all the mutators and, having completed its job, terminates.

The significance of the `if` statement at the end of inconsistent regions should be clear now. If a mutator *T* is suspended in an inconsistent region, the garbage collector will resume it (step 3), expecting it to resuspend itself soon (step 4). The `if` statement in line 4 of the code fragment given in Section 2.1 takes care of this suspension.

The key to achieving good overall performance is to limit the overhead imposed on mutator threads in their normal execution. GC happens infrequently enough, even in a generational system with a relatively small young generation, that the GC thread's actions have little impact on overall system performance. There are two direct sources of overhead imposed on mutators in the scheme described above: overhead resulting from transitioning in and out of inconsistent regions and overhead resulting from offering consistent points in the compiled code and the interpreter loop. We will study each of these costs in Section 3.2, but first we describe in detail the two techniques for offering consistent points. We will not look at costs, such as the serialization resulting from the stop-the-world operation itself.

2.3 Polling to suspend for GC

Our first implementation of EVM used polling in both the interpreter loop and the compiled code to offer suspension at GC points. In the interpreter loop, poll code of the following form would be invoked before every back-branch and method call:

```
if (gc_stop_threads) wait_for_gc_end().
```

The just-in-time compiler generated equivalent polling code:

```
    sethi %hi(&gc_stop_threads), %g1
    ld [%g1 + %lo(&gc_stop_threads)], %g1  -- load the boolean gc_stop_threads into %g1
    brz,pt %g1 skip_wait                   -- common case is to take the branch (no GC)
    nop                                    -- branch delay slot
    call wait_for_gc_end                   -- GC is happening, suspend till it is done
    nop                                    -- call delay slot
skip_wait:
```

In the common case, when no GC is pending, four of the six instructions execute. The compiler would emit polling code prior to back branches and in method prologues (rather than at call sites). As an optimization, polling would be elided from the prologue of any method that contained at most one call site. To see why this optimization does not jeopardize prompt GC suspension, we note that in the absence of tail-calls, if a thread executes entirely within a set of loop-free methods *M* each with at most one call site, the thread will amass a stack of height proportional to the execution time. Thus, the thread will either produce a stack overflow or step outside of *M* very rapidly. By offering a consistent point in the code that handles stack overflow, prompt GC suspension has been maintained.

The polling in the interpreter loop has virtually no impact on overall system performance. The general execution speed of interpreted bytecode is slow enough that it overwhelms the cost of polling, which executes at the speed of optimized, compiled C code. More importantly, performance critical bytecode should be compiled, so if a program spends a significant fraction of its time in the interpreter loop, we are already losing. In contrast to the interpreter loop, polling is a significant source of overhead for compiled code. The major problem is its cost in the frequent case when the call to `wait_for_gc_end()` is *not* taken. Worse yet, the overhead is imposed in

two situations that are particularly important for overall system performance: loops and method calls.

2.4 Code patching to suspend for GC

To eliminate the polling overhead in compiled code, we designed a suspension scheme based on code patching. The idea is to "patch in" calls to `wait_for_gc_end()` when needed, i.e., when preparing for GC. This saves work for the mutator threads, since they will not be executing any GC-related code unless GC is imminent, but adds work to the GC thread since it must now patch and unpatch code. Since GC is relatively rare, shifting work from mutators to the garbage collector is a net gain.

More concretely, consider three arbitrary machine instructions in a compiled method:

```
 instr1
*instr2
 instr3.
```

We have marked $\texttt{instr}_2$ with an asterisk to denote that it is a GC point. Code patching temporarily overwrites the instruction at the GC point ($\texttt{instr}_2$) with some other instruction that causes suspension. Once the GC completes, the original instruction is restored and the mutators are restarted. While the basic idea behind code patching is straightforward, the details turn out to be nontrivial.

2.4.1 The modified GC suspension steps

With code patching, the GC thread performs the same as before, except for additions to step 3 to patch the compiled code and to step 6 to restore the original code. The modified steps 3 and 6 become:

3′. The GC thread iterates over all mutator threads. For each mutator *T*, if *T* is consistent, it is already able to tolerate a GC, so it is simply left suspended. Otherwise, if *T* is executing in a compiled method *C* (determined by looking up *T*'s program counter[2] in a global data structure that maps code addresses to compiled methods), the garbage collector replaces the instructions at each GC point in *C* with a *GC trap*. Any attempt to execute a GC trap will cause the thread to suspend itself. Then *T* is resumed and, as before, it will execute until it either becomes consistent (at which point the polling at the end of the consistent region will cause suspension) or it hits a GC trap (at which point the trap will force suspension).

6′. Finally, the GC thread removes the GC traps that were inserted in step 3′, resumes all the mutators and, having completed its job, terminates.

2. The SPARC architecture actually has two program counters, PC and nPC, having to do with idiosyncrasies of delay slots. The consequence is that in the rare cases when PC and nPC point into different compiled methods, we must patch both methods. This happens when a thread is suspended after executing a call, but before executing the delay slot instruction.

2.4.2 Choosing code for GC traps

Execution of a GC trap must force a control-flow change to a routine that (eventually) suspends the mutator thread. On the way, it must capture the full state of the mutator at the time of the trap. The mutator needs the state when it resumes after the GC and the garbage collector needs the state to scan the registers and current frame of the mutator.

These requirements would be nicely satisfied if we could use a `call` instruction to implement GC traps. The call would branch to a routine that saves all registers to thread-local storage and then suspends. Unfortunately, on SPARC processors, calls have delay slots, making the instruction following a call execute "while the call is being taken." Consider again the three instructions shown in Section 2.4 where the GC point is at the second instruction, $\texttt{instr}_2$. Should a non-idempotent instruction $\texttt{instr}_3$ follow the GC point, an error would occur if we used plain calls to implement GC traps, since $\texttt{instr}_3$ would be executed when the trap is taken (in the delay slot of the call) and again when the mutator is resumed after GC. Could we use a call followed by a `nop`, which replaces the delay slot instruction, to implement a GC trap? The answer is no. We could have multiple mutators executing in the same method and one of them may be just about to execute the delay slot instruction. If this instruction were to be replaced with a `nop`, incorrect behavior would follow. We considered rendering the delay slots harmless by requiring any instruction following a GC point to be idempotent or invertible. In practice, many instructions are idempotent, including most tests, register moves, stores, loads that don't use the target register to compute the effective address, and others. Several more instructions, such as add, sub, and xor, are invertible, making it possible to undo their effect in the GC trap routine and therefore making it safe to re-execute them when the mutator resumes after GC. In our case, however, retrofitting such a constraint into the just-in-time compiler's code generator was nontrivial, so we decided to look for alternative solutions.

As a first try, we implemented GC traps using hardware traps rather than calls (traps have no delay slots). The beauty of this scheme is that it leverages the operating system's routines for capturing thread state. Specifically, we used a `clrb [%g0]` instruction to generate a trap (clear the byte at address zero (`%g0`)), which Solaris then transforms into a segmentation violation signal. The signal handler will be presented with a so-called `ucontext` data structure, which contains the full state of the thread at the point of the trap, and which can be used to resume the thread by issuing a `setcontext()` call. In terms of programming effort and portability, this solution is preferable over the `call` instruction that we first considered. Unfortunately, upon implementing the solution, we found that performance was not as good as we would have liked. On a 167 MHz UltraSPARC™ workstation, we timed the delivery of a segmentation violation signal and subsequent cleanup of the signal mask to 0.16 ms. An extreme but not totally improbable Java application could contain 200 active threads and allocate at a rate that requires 10 young generation collections per second. This amounts to a total signal delivery time of 200 * 10 * 0.16 ms = 320 ms in each second of execution. Spending 32% of the total elapsed time on suspending threads for GC is not satisfactory.

We also tested the `ta` (trap always) instruction. Its signal delivery was a bit faster, at 0.15 ms, but could still become a bottleneck. While much faster signal delivery is now possible through the so-called user-traps in Solaris 7, the need to support earlier versions of Solaris, made us settle on a more direct solution than kernel signals.

We patch in annulled, unconditional branches at the GC points (annulment suppresses the instruction following the GC point). To recover the PC of the GC point, which the branch instruction does not save and to get greater reach than the 1Mb that branches span on SPARC processors, we branch to a two-instruction trampoline routine and from there do a forwarding call to the suspension routine. Each GC point is given its own trampoline, found at the end of the compiled method, allowing recovery of the GC point address from the trampoline address. Essentially, we simulate an annulled call by using an annulled unconditional branch to a call. Figure 2 shows a method with two GC points before and after insertion of GC traps. The two trampolines correspond to each of the two GC points. The instruction in the call delay slot in the trampoline is a "fat nop:" it has no effect but encodes the GC point number by issuing a `sethi` with that number into the constant zero register `%g0`.

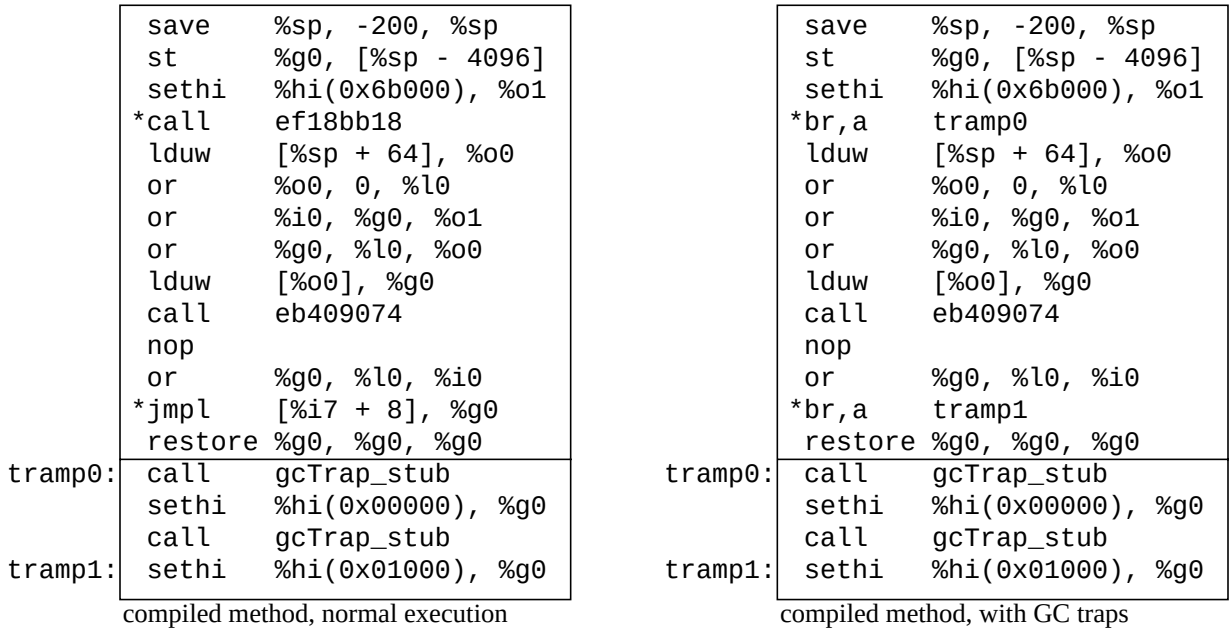

Figure 2. Compiled code for `StringBuffer.toString()`, slightly simplified.
Asterisks denote GC points, the only places that change when GC traps are inserted.

One subtle problem remains to be solved for GC traps to always work correctly. It involves two threads executing in the same method. Consider, this example:

```
       call F
L:  *ld [%l1 + 8], %o1
```

Suppose that `L` is a GC point and that a thread T_1 has just executed the call instruction, but not yet the delay slot instruction at address `L`. Now a garbage collection starts. T_1 is suspended and, because some other thread T_2 is executing in the above method, the instruction at `L` is replaced with a gc trap `ba,a`. When T_1 is resumed, it will execute the delay slot instruction, which is now an unconditional branch. The branch will yank T_1 out of the `F` function and cause T_1 to suspend. After the GC, the GC traps will be removed and T_1 will be resumed at address `L`, causing T_1 to

skip the call to F. We avoid this problem by requiring that GC traps stay clear of delay slots of other instructions. In practice, this restriction turned out to be trivially fulfilled by EVM's just-in-time compiler. Should this not be the case, problematic GC points can simply be preceded by a nop to shift them out of the delay slot.

Finally, a complication, which did not occur on SPARC processors, may have to be addressed on other architectures. On SPARC processors, all instructions are 32 bits wide. On other architectures, instruction widths may vary. To avoid having GC traps partially overwrite the instruction(s) that follow a GC point or have threads execute a suffix of a GC trap instruction, it may be necessary to have multiple forms of GC trap instructions so that a trap instruction can be selected which has the same size as the instruction it replaces.

2.4.3 Limiting the amount of code patching

The optimal placement of polling code differs from the optimal placement of GC traps. Polling code should be placed in method prologues rather than at call sites, since there are fewer methods than call sites in most programs. GC traps, in contrast, should be placed at call sites. The reason is that for virtual calls, it is not possible *a priori* to determine the actual method that will be invoked. If we had to place GC traps in all methods that could potentially be invoked from a call site in a compiled method, in the worst case, we would do work proportional to the size of the program. The alternative, placing the traps at call sites, bounds the work by the size of the largest method times the number of active mutator threads (each mutator causes GC traps to be inserted into zero, one, or two compiled methods). The only additional precaution we have to take is to place GC traps at return instructions to prevent mutators from escaping out of methods with GC traps.

Extremely large methods represent another concern. While the Java platform's bytecode verifier limits individual methods to 64 kb of bytecode, an optimizing compiler could create vastly bigger compiled methods by inlining. Patching all GC points in a monster method could be costly. We have not found this to be significant problem in EVM, where the inliner is only moderately aggressive, but more selective placement of GC traps could go a long way to alleviating the problem. Specifically, upon observing a mutator thread executing in a compiled method *M* that has a large number of GC points, rather than placing traps at all GC points, it suffices to place traps at enough places to guarantee that the mutator will hit one of them (the compiler must still emit trampolines for all the GC points, of course). For example, if the mutator is executing within a loop and we know the loop structure of the compiled method, it suffices to place a trap in the back branch of each loop that surrounds the mutator's location. If other mutators were found at different PCs in the same method, more traps could be inserted as necessary.

2.4.4 Interaction with other code patching

GC traps are just one form of code modification in EVM. Code modification is also used for on-demand class loading at the point of first active use [4], to back out optimizations when optimistic assumptions are violated by later class loading [5], and for fast virtual calls implemented with inline caches [6, 8]. In a multi-threaded environment, these different uses of code modification could collide and lead to races. For example, a trap at a virtual call site could be eradicated by an inline cache update, letting a mutator escape out of a method with GC traps. This mutator could then enter a long-running loop in a method without GC traps and indefinitely avoid GC suspension, thus blocking further progress for all other mutators in the system.

Fortunately, the mechanism to avoid such calamity was already in place in EVM. By imposing the restriction that only *consistent* threads are allowed to read and write compiled code, we ensure that mutators can never observe or overwrite GC traps, since when traps are present, the boolean `gc_stop_threads` is true, resulting in immediate suspension of any mutator that attempts to become consistent.

3 Measurements

Although our primary concern is execution speed, we measured both space and time aspects of the GC suspension code. Section 3.1 reports space data and Section 3.2 reports time data, in both cases using the set of benchmark programs shown in Table 1. The first seven benchmarks, _201_compress through _228_jack, constitute the SPECjvm98 suite [15]. The next two, _224_richards and _233_tmix, were among several additional SPECjvm98 candidate programs that were rejected in the final round of voting. We have included these two because they are multi-threaded. Finally, we selected the remaining benchmarks because they were nontrivial, stress the memory system, and use many threads.

Table 1. Characterization of benchmark programs.

Benchmark	**Description**	**#lines**[a]	**#threads**[b]	**#GCs**
_201_compress	LZW compression and decompression	927	1	19
_202_jess	Version of NASA's CLIPS expert system shell	10,579	1	142
_209_db	Search and modify a database	1,028	1	42
_213_javac	Source to bytecode compiler	25,211	1	114
_222_mpegaudio	Decompress audio file	n/a	1	2
_227_mtrt	Multi-threaded image rendering	3,799	3	73
_228_jack	Parser generator generating itself	8,194	1	125
_224_richards	Five threads running multiple versions of O/S simulator	3,637	1 (5[c])	2
_233_tmix	Thread mix: sort, crc, producer-consumer, primes, etc.	8,194	9	23
simplifier	Boolean expression simplifier written; by Robert Cartwright	472	1	591
java-in-java [16]	Bytecode interpreter written in Java language running the DeltaBlue prg.	10,069	1	87
java verifier	Bytecode verifier verifying itself; by Christine Flood	4,032	1	33
volano server[d]	"Chat server," reads and distributes messages	n/a	407	3
volano client	Generates work-load to stress server	n/a	401	16

a. Approximate number of lines of source code in the benchmark itself, excluding standard class library code.

b. Max number of mutator threads encountered during a GC, *excluding* three system threads (finalizer, reference handler, and signal dispatcher).

c. The _224_richards benchmark runs five user threads, but when the GCs occur, only a single thread is active.

d. VolanoMark version 2.0.0 build 137 [13].

Although many of the benchmarks are single-threaded, it is, we argue, still fair to use them in a study of GC suspension code in multi-threaded environments. Indeed, unless the Java virtual machine special-cases single-threaded programs (EVM does not), the execution overhead of GC polls remain the same whether one or multiple threads execute the code containing the polls. It is also worth observing that essentially all Java applications are de facto multi-threaded, even those

that only start a single thread, since the standard class libraries in the Java platform and the virtual machine itself start a number of “system threads” to handle object finalization and other background tasks.

3.1 Space

The space costs of exact garbage collection have two components: the maps required to decode stack frames at GC points and the code and data structures required to suspend threads at GC points. Counted per GC point, the space overhead for maps is independent of the GC suspension mechanism and has previously been documented [2, 7], so here we will concentrate on the latter.

In EVM, the GC poll sequence contains 6 SPARC instructions (see Section 2.3), which amounts to 24 bytes. A just-in-time compiler doing more flow analysis than our present one should be able to share the `sethi` instructions among several poll points, reducing the average cost to just over 20 bytes, but in the following we will use the slightly more conservative number of 24 bytes.

The poll-less GC suspension mechanism requires a 2 instruction trampoline (8 bytes) for each GC point, placed at the end of methods. We also need a way to map a given trampoline back to the corresponding GC point in the compiled code. However, in EVM, each method has an array of stack frame maps, allowing a positional matching of, say, the *i*’th trampoline back to the *i*’th stackmap, from which the address of the GC point in the compiled code can be found easily (stack maps contain a relative PC). Since these stack maps are necessary anyway, and since they are the same whether polling or patching is used, we don’t include their costs in the space budget for poll-less GC suspension.

Table 2 gives compiled code space costs for the two GC suspension approaches. For polling, the table gives three numbers for each benchmark: the number of GC points in compiled code, the number of poll points, and the total code space. In this case, there are many more GC points than GC polls, since many GC points need no poll (e.g., call sites need a stack map, but require no poll code since there is a poll in the prologue of the callee; exception handler entry points constitute another category of GC points that require no poll code). Overall, polling code amounts to at most 4% of the total code space. For patching, there are slightly more GC points than for polling. The increase results from removing a GC point from many but not all method prologues (recall that the poll was optimized away for trivial methods, see Section 2.3) but adding a GC point in *every* method return. Adding further to the space cost, every GC point now has an associated trampoline, which, although smaller than the poll sequence, occurs more often. The net result is that patching increases the total compiled code space over polling by 5.1% - 8.5%. We note, however, that the increased code space does not represent a significant icache dilution, since the added instructions (the GC trampolines) are kept out of line. In fact, icache utilization should improve, since the patching system avoids inline poll sequences.

Architecture and operating systems properties made us design a code patching scheme that uses an annulled branch to jump to a call. If the SPARC architecture had no delay slots or had an annulled call, or if the operating system had fast traps, trampolines would be unnecessary and code patching would produce a small space savings, since the poll code could be removed without the addition of trampolines.

Table 2. Compiled code space costs of polling versus code patching.

	polling			code patching		expansion,
Benchmark	**#GC points**	**#GC polls**	**total code**	**#GC points**	**total code**	**total code**
_201_compress	2,577	287	223 kb	2,706	237 kb	6.2%
_202_jess	4,537	525	358 kb	4,697	381 kb	6.3%
_209_db	2,739	319	230 kb	2,863	244 kb	6.0%
_213_javac	9,538	905	710 kb	10,014	765 kb	7.6%
_222_mpegaudio	2,947	415	287 kb	3,114	302 kb	5.1%
_227_mtrt	3,562	359	318 kb	3,718	339 kb	6.5%
_228_jack	5,432	463	408 kb	5,665	442 kb	8.2%
_224_richards	3,664	398	320 kb	3,927	341 kb	6.5%
_233_tmix	3,139	378	259 kb	3,299	276 kb	6.5%
simplifier	1,826	237	186 kb	1,953	196 kb	5.3%
java-in-java	5,722	346	432 kb	5,908	469 kb	8.5%
java verifier	3,126	366	265 kb	3,260	282 kb	6.3%
volano server	4,955	428	396 kb	5,191	426 kb	7.5%
volano client	2,744	314	252 kb	2,906	268 kb	6.2%
Average						6.6%

3.2 Time

Now consider time. We primarily report cycle and instruction counts gathered using the hardware performance counters in the UltraSPARC chip. Such counters are more accurate than elapsed time, with instruction counts being repeatable to at least five digits of precision, regardless of other activity on the machine, and cycle counts repeatable to two or more digits of precision. Since the counters only measure user level activity we also report the elapsed time as a bottom-line validation.

3.2.1 Cost of GC points in compiled code

The most important time component is the cost of offering consistent points during compiled code execution. This cost is the only time measure that depends on whether we use polling or traps. Table 3 summarizes the results obtained on our benchmarks. Code patching reduces total instruction counts by an average of 4.7%, ranging from 0.6% to 11.2% improvement, with a similar reduction in cycles and elapsed time.

3.2.2 Cost of inconsistent regions

Our primary aim is to compare the cost of polling with the cost of patching. However, whether the system polls or patches, another source of time overhead remains: the cost of transitioning in and out of inconsistent regions. This overhead is specific to the approach taken in EVM. In this section, we estimate the cost of these transitions. Our motivation for reporting these numbers is that we have found the inconsistent/consistent thread distinction an effective technique to facilitate coexistence of GC and large amounts of core virtual machine code written in a low-level language

Table 3. Time comparison of polling and patching.

	polling			patching			relative improvement		
Benchmark	**instructions**	**cycles**	**elapsed time**	**instructions**	**cycles**	**elapsed time**	**instructions**	**cycles**	**elapsed time**
_201_compress	11992.336M	15510.227M	103.9s	11129.601M	14731.292M	89.5s	7.2%	5.0%	13.8%
_202_jess	4299.264M	6145.159M	40.0s	4108.723M	6129.151M	39.8s	4.4%	0.3%	0.6%
_209_db	9386.964M	17171.363M	144.1s	9033.079M	16703.559M	140.2s	3.8%	2.7%	2.7%
_213_javac	8781.495M	11409.766M	82.7s	8422.158M	11223.930M	79.8s	4.1%	1.6%	3.5%
_222_mpegaudio	13781.832M	16163.663M	98.7s	13253.414M	16017.162M	97.1s	3.8%	0.9%	1.6%
_227_mtrt	3351.789M	5269.001M	33.0s	3251.974M	5110.383M	31.8s	3.0%	3.0%	3.7%
_228_jack	5906.806M	8353.468M	61.0s	5432.232M	7963.031M	58.1 s	8.0%	4.7%	4.7%
_224_richards	8616.369M	11662.088M	70.3s	7805.225M	10737.659M	64.7s	9.4%	7.9%	7.9%
_233_tmix	23042.037M	31766.119M	193.4s	20450.609M	29499.330M	179.5s	11.2%	7.1%	7.3%
simplifier	3037.558M	5578.371M	45.9s	3018.266M	5443.144M	45.8s	0.6%	2.4%	0.2%
java-in-java	4196.891M	5684.919M	42.6s	4007.502M	5285.704M	37.2s	4.5%	7.0%	12.7%
java verifier	537.325M	714.751M	5.9s	527.468M	677.197M	5.8s	1.8%	5.3%	1.2%
volano server	1112.771M	2191.149M	52.7s	1084.244M	2125.578M	51.5s	2.6%	3.0%	2.2%
volano client	1137.254M	2222.953M		1117.006M	2215.791M		1.8%	0.3%	
Average							4.7%	3.7%	4.8%

like C. By reporting the overhead, we hope other projects can make more informed decisions when designing and implementing run-time systems for languages with garbage collection.

To estimate the cost of transitioning in and out of inconsistent regions, we first determined the cost of an empty inconsistent region (consistent-inconsistent-consistent). Since threads keep their `inconsistentCount` in thread-local storage, and since the garbage collector only inspects counts on suspended mutators, the count can be manipulated without any locking. This enables very fast inconsistent regions. On an UltraSPARC CPU, an empty inconsistent region can be executed in 7 cycles or 10 instructions. While this measurement reflects the best case (no register pressure because the inconsistent region is very small), and does not account for the costs incurred by inconsistent regions preventing code motion and scheduling, we have no reason to expect the average inconsistent region to be vastly slower. Next, we instrumented the virtual machine to count the number of times a thread decrements its `inconsistentCount`, shown in the second column of Table 4. (This method of counting slightly overestimates the costs since for nested inconsistent regions only the outermost one incurs the full cost of transitioning.) Multiplying the per-region cost with the number of times threads leave inconsistent regions gives us an estimate of the total cost of managing inconsistent regions (third column). This cost can then be compared with the total number of instructions executed in each benchmark (fourth column). As the ratios in the last column of the table show, the cost of inconsistent regions never exceeds 4% of the total instructions and is usually significantly lower.

In an independent experiment, Mario Wolczko modified the virtual machine to use no inconsistent regions. This is *almost* safe for single-threaded programs. The tests that successfully ran on this modified virtual machine produced results in agreement with those in Table 4. At the time of this writing, unfortunately, these measurements can no longer be repeated since parts of the system have come to rely on inconsistent regions in a way that is nontrivial to take out.

Table 4. Estimated cost of transitioning in and out of inconsistent regions.

Benchmark	#inconsistent regions	#instructions to establish regions	total #instructions w. patching	estim. relative cost of incons. regions
_201_compress	87,768	0.877M	11129.601M	0.00%
_202_jess	5,371,894	53.718M	4108.723M	1.30%
_209_db	144,094	1.440M	9033.079M	0.01%
_213_javac	7,811,040	78.110M	8422.158M	0.92%
_222_mpegaudio	2,545,358	25.453M	13253.414M	0.19%
_227_mtrt	108,285	1.082M	3251.974M	0.03%
_228_jack	5,591,570	55.915M	5432.232M	1.02%
_224_richards	91,252	0.912M	7805.225M	0.01%
_233_tmix	3,620,231	36.202M	20450.609M	0.17%
simplifier	1,870,479	18.704M	3018.266M	0.61%
java-in-java	6,894,003	68.940M	4007.502M	1.72%
java verifier	2,090,997	20.909M	527.468M	3.96%
volano server	895,548	8.955M	1084.244M	0.82%
volano client	1,335,357	13.353M	1117.006M	1.19%
Average				0.85%

4 Discussion

Our colleague Steve Dever proposed an optimization of the polling scheme for compiled code that reduces the number of instructions for a non-taken poll from four to two. The optimized scheme uses Solaris' ability to write-protect a page of virtual memory. The virtual machine allocates a "poll page" of virtual memory. In normal operation, this page is write-enabled, but when threads must suspend for GC the garbage collector write-protects the page. A poll, then, consists of two instructions:

```
sethi %hi(pollPageAddr), %g1
clrb [%g1].
```

Normally, the write succeeds, but when the poll page is write-protected, the write results in a signal. The signal handler analyzes the signal to identify it as a poll, and then suspends the thread. Mosberger, Druschel, and Peterson refer to this technique as "controlled faults" [11]. The main advantage of this alternative poll sequence over the poll code shown in Section 2.3 is the reduction in the number of instructions from four executed (and six emitted) to two executed (and two emitted). Thus, in terms of time efficiency, the optimized poll code falls approximately halfway between the original unoptimized poll code and the code patching solution. The main disadvantages of the optimized poll code are the relatively high cost of UNIX signals as discussed in Section 2.4.2 and the fact that it does not completely eliminate the polling overhead.

If an interpreter for the hardware instruction set is available, it can be used instead of code patching to roll threads forward to the next GC point. The interpreter stops execution when the current thread becomes consistent or reaches a GC point. Now, an inconsistent mutator can be rolled forward simply by resuming it on the interpreter instead of the raw hardware. Moss and Kohler used this interpretation ("emulation") technique in an implementation of Trellis/Owl [12]: to prevent

untimely preemption from leaving threads in a non-debuggable state or a state in which garbage collection cannot be allowed, the system would use emulation to roll preempted threads forward into a safe region of code. Interpretation is attractive on systems where icache coherency or variable instruction sizes make code modification difficult. The main disadvantages to interpretation are, first, that it is a considerable task to implement an interpreter for a modern architecture, and, second, that in the roll-forward phase threads will execute at a fraction of their normal speed, potentially requiring a higher density of GC points to keep the GC start latency reasonable.

The reduction in instruction counts that patching achieves over polling may seem relatively small in the context of Java virtual machines where, these days, advances in technology almost routinely deliver much more dramatic speedups. However, several points should be kept in mind. First, as unrelated optimizations continue to improve the speed of the rest of the system, the relative advantage of patching over polling will increase. Second, when Java virtual machine technology matures to the same level as current implementations of older languages like C and Fortran, a speed difference of a few percent may be all that separates the most competitive Java virtual machines. Third, unlike compiler optimization, which in a dynamic system like a Java virtual machine doesn't come for free (compile time takes away from execution time), GC poll elimination benefits even short-running programs since it has virtually no fixed overhead.

Finally, we would like to offer an different angle on our results. The instruction count reduction resulting from patching should be seen not just in relation to the total number of instructions, but also in relation to the number of instructions spent in the garbage collector, since, after all, GC points are a GC-related cost. Because most programs on EVM spend less than a fifth of their total time in GC, one may view code patching as reducing the cost of garbage collection by approximately 5 * 4.7% = 23%. This is a significant speedup to get in a highly tuned memory system.

5 Related work

Mosberger, Druschel, and Peterson discuss software implementations of atomic ("uninterruptible") sequences for uniprocessor operating systems [11]. They advocate the use of roll-forward: if a thread is about to be interrupted in the middle of an atomic sequence, it is rolled forward to the end of the sequence by the interrupt handler. In many regards, atomic sequences resemble the regions of code between GC points in our system. Mosberger, Druschel, and Peterson suggest several ways to stop threads at the end of an atomic sequence, including code patching, faulting instructions (such as a dummy write to page that can be write-protected), and code cloning (threads are restarted in a copy of the code that has been modified to cause suspension at the desired point).

Tamches and Miller have developed a code instrumentation scheme that allows instrumentation of a running operating system kernel [17]. Their technique, implemented on the SPARC processor, is similar to the present work and also must work correctly in a threaded environment. To instrument a point in a piece of active code, they replace a single instruction with an annulled branch. The branch diverts control flow to a so-called springboard routine, similar to our GC trampolines, from where a longer-reaching branch or call forwards to the actual instrumentation code. Tamches and Miller have no control over the layout of the code that they instrument, so they could not assume the luxury of springboards conveniently placed at the end of every method. Instead, they

find pieces of unused memory, such as that resulting from initialization code that has already run, to place their instrumentation code.

Peyton Jones and Ramsey discuss machine-independent support for, among other things, garbage collection in the context of C--, a “portable assembler” language [14]. They define safe points as program points where it is safe to suspend execution and subsequently inspect and modify variables and perform GC. Thus, safe points generalize our GC points. They also propose the operation `ExecuteToNextSafePont()`, which can be invoked after a thread has received an asynchronous event, such as an exception or a GC suspension request originating in another thread. They do not give a concrete implementation of `ExecuteToNextSafePont()`, but for performance reasons it would seem attractive to use code patching similar to the scheme described in this paper.

6 Conclusions

We have described the steps involved in suspending all mutators at GC points in EVM. Its cornerstone is a distinction between inconsistent and consistent mutator threads, where the former temporarily cannot tolerate a GC. When GC is imminent, all mutators are suspended, whereupon the inconsistent ones are restarted to allow them to execute up to the next consistent point. Our study shows that a polling system, the most straightforward approach to stopping threads at GC points, on average executes 4.7% more instructions than patching. The main disadvantages to patching are a 6.6% higher space cost, in our implementation, and increased implementation complexity.

By today’s standard, a single-digit percentage speedup of Java applications may seem relatively insignificant, but as Java virtual machine technology continues to mature, the relative importance of GC poll elimination will likely increase. Moreover, a justifiable way to view our results is that patching yields a 23% reduction of GC-related costs in EVM. We hope that by documenting the present work and results, future implementors of garbage-collected run-time systems can make a more informed choice about the thread suspension mechanism.

Acknowledgments. The concrete design and implementation of the poll-less GC suspension code described in this paper evolved through discussions with several people at Sun Microsystems. Ross Knippel and Mario Wolczko, in particular, were directly involved in the design and implementation. LLNI was developed in early 1997 at Sun Labs by the Java Topics group. Lars Bak, David Detlefs, Alex Garthwaite, Urs Hölzle, Doug Lea, Steve Heller, Derek White, and Mario Wolczko read drafts and offered advice and suggestions that significantly improved this paper. Thanks to Sreeram Duvvuru for explaining intricacies of SPARC architecture delay slots.

References

1. Ole Agesen and David L. Detlefs. Finding References in Java Stacks. *OOPSLA'97 Garbage Collection and Memory Management Workshop*. Atlanta, GA, October 1997. http://www.dcs.gla.ac.uk/~huw/oopsla97/gc/papers.html.
2. Ole Agesen, David Detlefs, and J. Eliot B. Moss. Garbage Collection and Local Variable Type-Precision and Liveness in Java Virtual Machines. In *Proceedings of the ACM SIGPLAN '98 Conference Programming Language Design and Implementation (PLDI)*, p. 269-279, Montreal, Canada, June 1998.
3. Hans-Juergen Boehm, Alan J. Demers, and Scott Shenker. Mostly Parallel Garbage Collection. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation (PLDI)*, p. 157-164, Toronto, ON, Canada, June 1991.
4. Timothy Cramer, Richard Friedman, Terrence Miller, David Seberger, Robert Wilson, and Mario Wolczko. Compiling Java Just in Time: Using runtime compilation to improve Java program performance. *IEEE Micro*, 17(3), p. 36-43, May/June 1997.
5. David Detlefs and Ole Agesen. *Inlining of Virtual Methods*. To be presented at ECOOP'99.
6. L. Peter Deutsch and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th Symposium on the Principles of Programming Languages*, p. 297-302, Salt Lake City, UT, 1984.
7. Amer Diwan, Eliot Moss, and Richard Hudson. Compiler Support for Garbage Collection in a Statically Typed Language. In *Proceedings of the ACM SIGPLAN '92 Conference Programming Language Design and Implementation (PLDI)*, p. 273-282, San Francisco, CA, June 1992.
8. Urs Hölzle. *Adaptive Optimization in Self: Reconciling High Performance with Exploratory Programming*. Ph.D. thesis, Department of Computer Science, Stanford University, Stanford, CA, August 1994. Also published as Sun Microsystems Laboratories Technical Report, SMLI TR-95-35, March 1995.
9. Bil Lewis and Daniel L. Berg. *A Guide to Multithreaded Programming: Threads Primer*. SunSoft Press and Prentice Hall, 1996.
10. Timothy Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series, Addison-Wesley, 1996.
11. David Mosberger, Peter Druschel, and Larry L. Peterson. Implementing Atomic Sequences on Uniprocessors Using Rollforward. *Software—Practice and Experience*, 26(1), p. 1-23, January 1996.
12. J. Eliot B. Moss and Walter H. Kohler. Concurrency Features for the Trellis/Owl Language. In Proc. *European Conference on Object-Oriented Programming (ECOOP'87)*, p. 171-180, LNCS 276, 1987.
13. John Neffinger. Which Java VM scales best? *JavaWorld*, August 1998. http://www.javaworld.com/javaworld/jw-08-1998/jw-08-volanomark.html. See also www.volano.com.
14. Simon L. Peyton Jones, and Norman Ramsey. *Machine-Independent Support for Garbage Collection, Debugging, Exception Handling, and Concurrency*, Technical Report, CS-98-19, Department of Computer Science, University of Virginia, August 1998.
15. SPEC jvm98 Benchmarks. August 19, 1998 release. http://www.spec.org/osg/jvm98.
16. Antero Taivalsaari. *Implementing a Java™ Virtual Machine in the Java Programming Language*. Technical Report, SMLI TR-98-64, Sun Microsystems Laboratories, Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303 USA, March 1998.
17. Ariel Tamches and Barton P. Miller. *Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels*. Third *Symposium on Operating Systems Design and Implementation (OSDI'99)*, New Orleans, February 1999.
18. David Ungar and Frank Jackson. An Adaptive Tenuring Policy for Generation Scavengers. In *ACM Transactions on Programming Languages and Systems*, 14(1), p. 1-28, January 1992.
19. Derek White and Alex Garthwaite. *The GC Interface in the EVM*. Technical Report, SMLI TR-98-67, Sun Microsystems Laboratories, Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303 USA, December 1998.

About the Author

Ole Agesen is a Senior Staff Engineer in the Java Topics Group in Sun Labs. Previously, he worked in the Kanban group and the Self group, also in Sun Labs. He has an M.S. degree from Aarhus University in Denmark and M.S. and Ph.D. degrees from Stanford University in California.