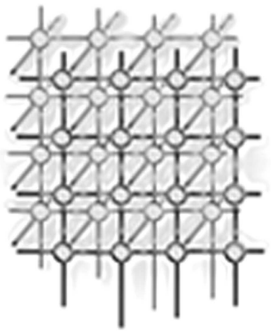
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2006; **00**:1–7 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*

Accurate Garbage Collection in Uncooperative Environments Revisited



J. Baker, A. Cunei,* T. Kalibera, F. Pizlo, J. Vitek

Purdue University, West Lafayette, IN 47907, U.S.A.

SUMMARY

Implementing a concurrent programming language such as Java by the means of a translator to an existing language is attractive as it provides portability over all platforms supported by the host language and reduces development time – as many low-level tasks can be delegated to the host compiler. The C and C++ programming languages are popular choices for many language implementations due to the availability of efficient compilers on a wide range of platforms. For garbage-collected languages, however, they are not a perfect match as no support is provided for accurately discovering pointers to heap-allocated data on thread stacks. We evaluate several previously published techniques, and propose a new mechanism, lazy pointer stacks, for performing accurate garbage collection in such uncooperative environments. We implemented the new technique in the Ovm Java virtual machine with our own Java-to-C/C++ compiler using GCC as a back-end compiler. Our extensive experimental results confirm that lazy pointer stacks outperform existing approaches: we provide a speed-up of 4.5% over Henderson's accurate collector with a 17% increase in code size. Accurate collection is essential in the context of real-time systems, we thus validate our approach with the implementation of a real-time concurrent garbage collection algorithm.

1. Introduction

Implementing a high-level programming language involves a large development effort. The need for performance of the resulting environment has to be balanced against concerns such

*Correspondence to: Antonio Cunei, Department of Computer Science, Purdue University, 305 N. University Street, West Lafayette, IN 47907, USA

†A preliminary version of this work was presented at the 16th International Conference on Compiler Construction (CC 2007), part of the Joint European Conferences on Theory and Practice of Software (ETAPS 2007), March 26–30, 2007, Braga, Portugal.

Contract/grant sponsor: National Science Foundation; contract/grant number: HDCCSR-0341304 and CAREER-0093282

Received
Revised 5 March 2008

as portability and extendibility. One popular implementation technique is to use a language translator to translate the high-level code into an equivalent program in an existing, often lower-level, language, thus leveraging existing technology for part of the implementation. A time tested road is to use C or C++ as a form of portable assembly language, thus benefiting from C's portability and offloading many optimizations to the native compiler. However, these advantages come at a price. Some control over representation and code generation must be relinquished. One often encountered problem is that a C or C++ compiler such as GCC [20] will not provide support for automatic memory reclamation. It is up to the language implementor to bridge the semantic gap between the features of the high-level language and what is available in the low-level language. In the case of garbage collection, implementors end up programming *around* the back-end compiler to enable automatic memory reclamation.

The most straightforward solution to this particular problem is to use a *conservative* garbage collection algorithm. A garbage collector reclaims objects that have become unreachable. While it is desirable to reclaim garbage promptly, it is always sound for a collector to over-estimate the set of reachable objects. Conservative algorithms avoid the need for cooperation from their environment by treating some numeric values as pointers. In particular, they traverse the call stack conservatively, treating every value that could possibly be a pointer as a pointer to dynamically allocated data. This may result in suboptimal memory usage as some memory that does not contain live data may be retained, but there is a guarantee that memory that is freed is actually unreachable. There exist excellent off-the-shelf conservative collectors such as [13, 12] that can be used with very little implementation effort.

This paper looks at how to support *accurate* garbage collection, in which all pointers can be correctly identified in an uncooperative environment. Our ultimate goal is to provide the necessary support for the implementation of a variety of *concurrent garbage collection algorithms* for a high-performance implementation of the Java programming language. Before embarking on any non-trivial implementation exercise, one should first get a clear picture of the tradeoffs involved. In the present case, why is conservative collection not good enough? If our main worry is portability and ease of development, a conservative collector is ideal as it imposes no burden on the high-level language implementor and usually performs well. But the choice between accurate and conservative collection has far-reaching consequences. Conservative collectors may encounter pathological cases in which they may fail to free any memory because of a non-pointer word that happens to look like a pointer to a dead object. Features such as memory defragmentation require moving objects from one memory address to another. To do this, the collector must be able to update all pointers to an object. In a conservative setting, the collector may not be sure that a location holds a pointer or just a sequence of bits that looks like one. Opportunities for moving objects are thus reduced to parts of the memory that can be treated accurately. One application domain where accurate garbage collection is essential is that of real-time systems. Real-time systems that make use of a garbage collector must ensure that all application deadlines are met even in the presence of pauses introduced by a concurrent garbage collection thread. For this, the garbage collector has to be predictable – a trait not found in conservative collectors. There will always be a worst case scenario in which a conservative collector is either not able to reclaim as much memory as it should, or experiences a slowdown because it has to traverse additional memory ranges due to an unlucky value in some local variable.

This paper looks at how to support *accurate* garbage collection, in which all pointers can be correctly identified in an uncooperative environment. Although our work environment is Java, a high-level, multi-threaded, object-oriented language, the discussion generalizes to other high-level language translators. We evaluate several approaches to generating idiomatic C/C++ code that maintains enough information to allow a garbage collector to accurately find and replace pointers. Our goal is to minimize the overheads, bringing the performance of our accurate configuration as close as possible to that of our conservative configuration. The work is being done in the context of the Ovm virtual machine framework. We offer the following contributions:

- **Lazy pointer stacks:** We present a class of new techniques for maintaining accurate information on the call stack of any thread in the system. It promises lower overheads than previous work because the information is only materialized when needed, leaving the native compiler free to perform more optimizations.
- **Thunked stack walking:** We propose an efficient technique for saving and restoring the pointers on a call stack, which improves performance of lazy pointer stacks using call stack manipulation. The technique can use the C++ exception handling mechanism, but can also be efficiently implemented without it. Thus, our technique is applicable to translators that target C as well as C++.
- **Implementation:** We implemented our technique in the Ovm framework and observed that there was some space for a performance improvement, which we achieved by making a number of changes to our bytecode to C/C++ compiler. We report on our implementation and describe and motivate the compiler optimizations that were added.
- **Validation:** We compare our technique against an efficient conservative collector and two previously published techniques for accurate collection. The results suggest that our approach incurs less overhead than other techniques. To further validate our work, we report on the implementation of a concurrent real-time garbage collector within Ovm using lazy pointer stacks to obtain accurate stack roots.

The remainder of this paper is structured as follows. Section 2 gives an overview of the Ovm virtual machine. In Section 3 we describe existing techniques for accurate stack scanning in uncooperative environments. In Section 4 we introduce our lazy pointer stack approach. Section 5 discusses compiler optimizations. The experimental evaluation is found in Section 7. Our real-time garbage collector, which uses accurate stack scanning, is described in Section 8. Section 9 discusses related work. Finally, Section 10 summarizes our conclusions.

2. The Ovm Virtual Machine

Ovm is a framework for building virtual machines with different features. An Ovm *configuration* determines a set of features to be integrated into an executable image. While Ovm supports many configurations, one of the project's topmost goals was to deliver an implementation of the Real-time Specification for Java running at an acceptable level of performance [8]. This section discusses the two most important aspects of the real-time configuration of Ovm with

respect to our implementation of the collection algorithms described in this paper. The reader is referred to [25, 8, 18] for further descriptions of the framework.

2.1. The J2c Translator

The Real-time Ovm configuration relies on ahead-of-time compilation to generate an executable image that can be loaded in an embedded device (such as the UAV application discussed in [8]). The Ovm ahead-of-time compiler called `j2c` performs whole-program analysis over the user code as well as the Ovm source (the virtual machine framework consists of approximately 250 000 lines of Java code). `J2c` translates the entire application and virtual machine code into C or C++ which is then processed by the GCC compiler. It should be noted that while we focus on `j2c` results, the techniques presented in this paper do not require whole-program optimizations.

2.2. Threading subsystem

The version of Ovm used in this work targets uniprocessors and implements threading at the user level[†]. Multiple Java threads are mapped onto one operating system thread. Threads are implemented by *contexts* which are scheduled and preempted under VM control. Asynchronous event processing, such as timer interrupts and I/O completion is implemented by the means of compiler-inserted *poll checks*. A poll check is simply a function call guarded by a branch on the value of a global variable:

```
if (pollWord == 0) {
   handleEvents();
}
```

For typical programs poll checks need only be inserted on back-branches – since we do not support tail call optimization and call stack heights are bounded, it is not mandatory to insert poll checks upon method entry. Two concerns arise with the use of poll checks. First, the execution of poll checks, as well as compiler optimizations impeded by their presence, will tend to slow the program down. Second, an incoming event cannot be handled until a poll check is executed. Put another way, we do not wish for the overhead of poll checks to be too great while minimizing the effect on event handling latency caused by placing poll checks too sparsely. It turns out that our current poll check insertion strategy leads to less than 2.5% overhead. Further, studies we have done with a real-time application show that the latency between the arrival of an event and the execution of a poll check tends to be under $6\mu s$. Additionally, Ovm provides a compile-time option for increasing the frequency of poll checks – for example, the insertion of poll checks at method prologues can be easily enabled and results in only a modest amount of additional overhead.

[†]We also have a native-threaded multiprocessor version of Ovm, however we have not performed the experiments found herein on that version.

Our approach to poll checks is typical of virtual machines that perform user-level scheduling. Notably, Jikes RVM uses poll checks that compare a processor-local value against zero in prologues, epilogues, and back branches [1]. Our approach differs from virtual machines that only require garbage collection safe points – a garbage collection "safe point" triggers when a collection is needed, whereas a "poll check" also triggers whenever a scheduling decision is requested, or in the case of Jikes RVM, whenever profiling or on-stack-replacement are needed. Since systems that *only* require garbage collection safe points see these points trigger much more rarely, they typically use alternate implementation strategies – often based on page poisoning – that are faster in the untriggered case but much slower in the triggered case. A detailed description of Ovm's poll checks and a study of their performance appeared in [8].

We leverage poll checks in our implementation of memory management. The garbage collector can only run if all threads are blocked either at a poll check, calling the memory allocator or invoking a scheduler operation. This makes for a simple definition of safe points: in Ovm the only safe points are poll checks and certain Java method calls.

3. Previous Work: Explicit Pointer Stacks & Henderson's Linked Frames

It is often possible to assume that heap-allocated data structures have accurate type-descriptors, and that information can be used by the garbage collector. Determining the location of pointers in the stack, however, is less easy. While the native compiler knows which locations in the call stacks contain pointers and which do not, this knowledge is normally lost once the executable has been produced. We found two previously used techniques for accurately scanning call stacks. The simpler of the two uses an explicit stack of live pointers. The other technique, presented by Henderson [21], involves building a linked list of frames that contain pointers. This section describes both techniques in detail.

3.1. Explicit Pointer Stacks

While a compiler is free to lay out local variables however it wants, it has less freedom when dealing with objects in the heap. When generating C code, a translator can choose to emit code that will store all pointers in an array that is at a known location and has a fixed layout. We call this array an *explicit pointer stack*. Consider Fig. 1(a), where a function allocates an object, stores a pointer to it in a local variable, and then calls a second function passing the pointer as an argument. Fig. 1(b) illustrates the same function using an explicit pointer stack. The code uses a global pointer to the topmost element of the stack, `PtrStackTop`. The prologue of the function increments the stack top by the number of pointer variables used in the function (one in this case), and the epilogue decrements it by an equal quantity. References are then stored in the reserved stack slots.

3.2. Henderson's Linked Frames

Henderson proposed a different approach that allows pointers to be stored on the stack, taking advantage of the fact that the address of a C local variable may either be passed to another

```
void Foo(void) {
   void *ptr = AllocObject();
   Bar(ptr);
   ...
}
```

(a) Generated C code

```
static void **PtrStackTop;
void Foo(void) {
   // allocate stack slot for the pointer
   PtrStackTop++;
   PtrStackTop[-1] = AllocObject();
   Bar(PtrStackTop[-1]);
   ...
   // relinquish stack slot
   PtrStackTop--;
}
```

(b) Explicit pointer stack

```
struct PtrFrame {
   PtrFrame *next;
   unsigned len;
}
static PtrFrame *PtrTop;
void Foo(void) {
   // describe this frame
   struct Frame: PtrFrame {
      void *ptr;
   }
   Frame f;
   f.len = 1;
   f.next = PtrTop;
   PtrTop = &f;
   f.ptr = AllocObject();
   Bar(f.ptr);
   ...
   // pop the stack
   PtrTop = f.next;
}
```

(c) Henderson's linked lists

Figure 1. Example of previous techniques for accurate stack traversal. In (a), we see the original code. In (b) and (c) we see the same code converted to use explicit pointer stacks and Henderson's linked frames.

function or stored in the heap. A native compiler handles these variables specially, ensuring that changes made through these external references are visible locally. Fig. 1(c) illustrates Henderson's technique. The `PtrFrame` data structure is used to build a linked list of frames that hold live pointers. The translator emits a function prologue that declares a frame with sufficient space to hold all the pointers (just one in our example). The frame is placed into a linked list that can be subsequently traversed by the garbage collector.

Both techniques pin local variables into specific memory location that cannot easily be optimized by the compiler. In the absence of good alias analysis, any write to a pointer variable will invalidate previous reads of all other pointer variables. Hence, the effectiveness of optimizations such as register allocation is limited, as pointers can not be moved around or stored in registers. These approaches thus have both direct and indirect performance impact. We will next look at alternative implementation techniques that reduce that costs of keeping accurate information about the location of pointers on the stack.

```
void Foo(void) {
   void *ptr = AllocObject();
   Bar(ptr);
   if (save()) {
      lazyPointerStack->pushFrame(1);
      lazyPointerStack->pushPtr(ptr);
      return;
   }
   ...
}
```

Figure 2. Lazy pointer stack construction: generated code for function `Foo` from Fig. 1(a).

4. Accuracy with Lazy Pointer Stacks

The key to accurately obtaining references in the call stack is to force the compiler to place references in specific locations, which the approaches above do by segregating references to an explicit pointer stack, or in Henderson's case, to a linked frame structure. Both approaches are *eager* in the sense that the data structures describing live pointers are always up-to-date. In our work we investigate techniques that construct the equivalent of a pointer stack on demand. We refer to this approach as *lazy pointer stacks*. The expected advantages of laziness are that accesses to references are direct and that the native compiler will have more opportunities for optimization; in particular, to allocate references in registers.

The goal of a lazy pointer stack algorithm is to produce at any GC safe point a list of all references on the call stack of a given thread. For simplicity, we assume that safe points are associated to call sites[‡]. Other granularities are possible, but increasing the frequency of safe points is bound to diminish optimization opportunities and increase overheads of the lazy approach. The eager approaches mentioned in the previous section take the extreme view that every instruction is a safe point.

For every safe point, the translator has a set of reference variables that may be live. Each safe point is followed by a guarded sequence that saves all the live references and simply returns, as in Fig. 2. When a stack needs to be scanned, we arrange for the guard to evaluate to true and return from the topmost frame. The call stack then unwinds, saving all the references in the process. Once all pointers are saved to the lazy stack, the GC can use this data to work accurately.

After unwinding the stack, we restore the thread to its initial state; specifically we restore the C call stack and the register file. If we are unwinding a thread we just context-switched to,

[‡]This is *de facto* the case in Ovm. GC can only be triggered by a call to the memory allocator. By definition, all other threads are blocked in calls to the event processing method that follows a successful poll check.

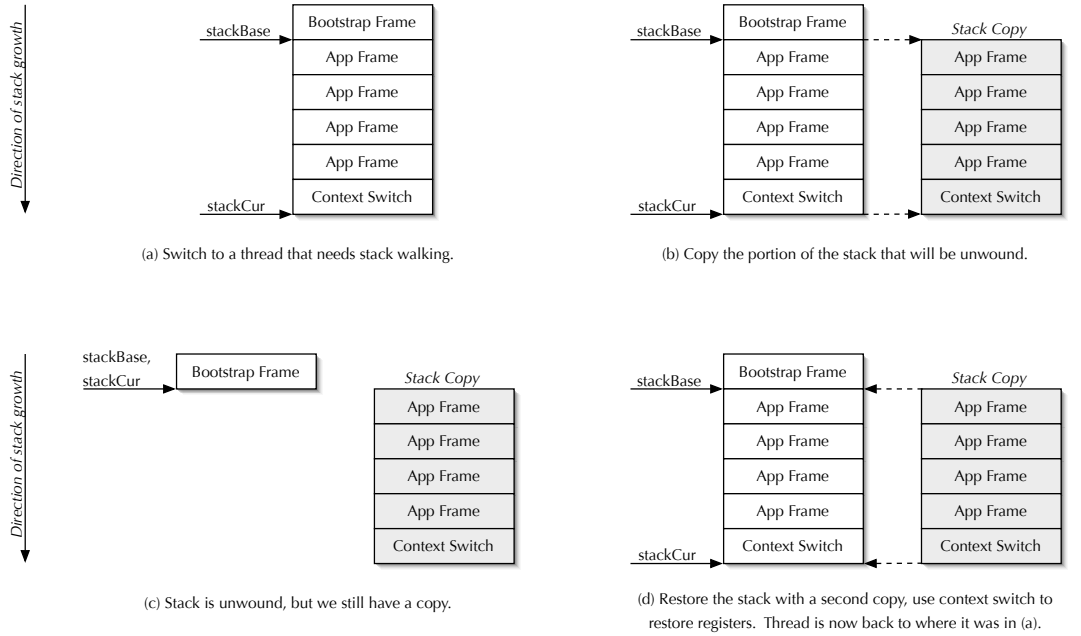


Figure 3. Example of stack unwinding and restoration.

```
void *stackBase; // we save the base of the stack when bootstrapping a thread
void *stackCur = GetSP(); // GetSP is a platform-specific function for getting the stack pointer
void *backupStack = malloc(stackBase - stackCur); // code for saving the stack
memcpy(backupStack, stackCur, stackBase - stackCur);
memcpy(stackCur, backupStack, stackBase - stackCur); // code for restoring the stack
free(backupStack);
```

Figure 4. Code to save and restore the call stack stack, assuming downward stack growth. Note that if we save the stack and then execute the stack unwinding procedure, we can use a combination of a context switch and the restore code above to return to the thread its original state
.

```
class LazyPointerStack {
public:
   LazyPointerStack(unsigned maxHeight);
   void pushFrame(unsigned N);          // start a new frame with N pointers
   void pushPtr(void *ptr);             // push pointer into frame
   void *popPtr();                      // pop pointer from top-most frame
   void popFrame();                     // pop a frame; can be used to skip entire frame
}
extern LazyPointerStack *lazyPointerStack; // pointer to lazy pointer stack for current thread
```

Figure 5. Lazy pointer stack mutator interface.

we already have a copy of the register file, otherwise, we save it using `setjmp`[§]. This procedure is justified by the fact that the C call stack is always a contiguous block of memory; provided we know the direction of growth, the base of the stack, and the current stack pointer, we can make a bit-for-bit copy of the stack before the unwind operation, and subsequently restore it with a further copy operation. The procedure is illustrated in Fig. 3. Notice that, in Fig. 3(d), the context saved on the top of the stack, which typically includes the frame pointer and local variables, is also restored. This allows the thread to continue executing normally after garbage collection. The restore operation must be executed on an alternate stack. This can be accomplished in a number of ways. For example, POSIX provides an alternate stack for signal handlers. Switching to this stack can be accomplished with a `raise` operation. Alternatively, if user-level threads are already implemented, it is possible to simply restore one thread's stack from another thread's context. Fig. 4 shows code for saving and restoring the C call stack in systems where the stack grows down. We provide a lazy pointer stack interface for use by the mutator in Fig. 5. The collector has a separate interface for scanning the pointer stack and replacing pointers in it.

Putting all of this together, the sequence of steps to discover all pointers on a thread's call stack is the following: (**a**) copy the thread's stack, and if this is the thread that triggered GC, the thread's context; (**b**) set the guard to true; (**c**) switch to the thread; (**d**) as the stack unwinds, pointers are pushed on the GC's stack; (**e**) at the bottom of the stack, proceed with stack restoration.

The procedure we just described allows the collector to build up a pointer stack for each thread. It is less restrictive than eager approaches: the compiler is free to choose the location of pointer-valued locals because they are explicitly copied to the heap when and if needed. This simple strategy is all that is needed if the garbage collector does not move objects. Supporting a moving collector, however, requires the ability to update the pointers contained in local variables.

Updating the pointers held in local variables, which might have been modified by the garbage collector, can also be done lazily. After collection, when each thread resumes execution, we cause each frame to perform pointer restoration as control returns to it. As the garbage collector runs, the pointers stored in the lazy pointer stack structure are used and modified. When the collector yields back to the application threads, the pointers are automatically restored from the pointer stack, frame by frame. The restoration logic has two key aspects. First, restoration must only be started the first time we return to a frame after a collection, which may happen immediately after the collection, or later. Second, a thread may return to a frame after multiple collections have occurred. This complicates the stack unwinding procedure. If at the time of a stack scanning request it is found that a frame has not been active since before a previous garbage collection, then the pointers in that frame are no longer valid, and the collector should not use that frame's pointers as roots but rather reuse the pointers in its lazy pointer stack. A frame is said to be *stale* if it contains references to objects that were not updated after a

[§] As the content of `setjmp` buffer is platform dependent, this code is also platform dependent. On some platforms, `setjmp` does not save all needed registers, and thus we complement it by a few lines of inline assembly. A similar effect could be accomplished by using POSIX get/setcontext functions.

collection (these references are stale if the objects were moved). We see that for each thread's call stack, there is a frontier between stale and clean frames. For stale frames, the lazy pointer stack has correct pointers. For clean frames, the lazy pointer stack has no information.

We developed two original solutions allowing pointer restoration: *counted* and *thunked* lazy pointer stacks.

4.1. Counted Lazy Pointer Stacks

In counted lazy pointer stacks, we keep track of the frontier between stale frames and clean frames with two counters: `entered`, the number of frames the thread entered, and `stale`, the number of stale frames on the thread's call stack. Fig. 6 shows a slightly simplified view of the output of the translator. `entered` is updated and read exclusively by the thread whose stack is scanned. It is incremented before each call and decremented after it. After garbage collection, before a thread regains control, `entered` equals `stale`. At that time, the lazy pointer stack contains an up-to-date version of all pointers in the thread's call stack. Until the next garbage collection, the size of the lazy pointer stack is equal to `stale`. `stale` $\leq$ `entered` must hold, except for immediately after a function return, when `stale` $>$ `entered` signals a special situation.

```
void *ptr1, ..., *ptrn; // pointers in locals
entered++;
functionCall();
entered--;
if (entered < stale) {
   if (HIGHEST_BIT_SET(stale)) { // stack scanning
      if (entered < CLEAR_HIGHEST_ORDER_BIT(stale)) {
         stop stack scanning, unwind and return to gc;
      } else {
         lazyPointerStack->pushFrame(nptrs);
         lazyPointerStack->pushPtr(ptr1);
         ...
         lazyPointerStack->pushPtr(ptrn);
         return;
      }
   } else { // pointer restoration
      ptrn = lazyPointerStack->popPtr();
      ...
      ptr1 = lazyPointerStack->popPtr();
      lazyPointerStack->popFrame();
      stale--;
   }
}
```

Figure 6. Counted lazy pointer stack technique.

```
unsigned savedEntered = entered;
void *ptr1, ..., *ptrn; // pointers in locals
...
try {
   ...
} catch (const ApplicationException&) {
   // restore counts
   entered = savedEntered;
   while (entered < stale-1) { // ignore pointers in frame
      lazyPointerStack->popFrame();
      stale--;
   }
   if (entered < stale) {
      ptr1 = lazyPointerStack->popPtr();
      ...
      ptrn = lazyPointerStack->popPtr();
      lazyPointerStack->popFrame();
      stale--;
   } // handle application exception
   ...
}
```

Figure 7. Compiling try blocks to restore the pointer frame counts.

Upon a normal return, `entered` can become smaller than `stale` by one. This is the signal that the frame being returned into is stale and its pointers must be updated. Once done, `stale` can be decremented.

Upon an exceptional return, `stale` may be larger than `entered` by more than one. In this case, the stale frames belonging to frames bypassed by the exceptional return are discarded. Fig. 7 shows the compilation of exceptions. In addition, maintaining correct value of `entered` requires `entered` to be copied to a thread's frame local variable prior a function call that may return by an exception.

When garbage collection is needed, the collector requests stack scanning by setting the highest-order bit of `stale`, making it definitely larger than `entered`. `stale` is then constant throughout the whole process of stack scanning of a single thread, and except for its highest-order bit it still contains the number of stale frames on the thread's call stack. After updating `stale`, the garbage collector initiates scanning by passing control to the thread. The technique of signaling stack scanning using the highest-order bit of `stale` reduces the number of tests that have to be made after each function return in case the current frame is clean, and thus there is nothing special to be done – neither stack scanning, nor pointer restoration. After a thread regains control and decrements `entered`, if `stale` $\leq$ `entered`, there is nothing special to be done.

The stack scanning is distinguished from pointer restoration by checking the highest-order bit of `stale`. In stack scanning, the logic for finding out if the current thread's frame is stale, is

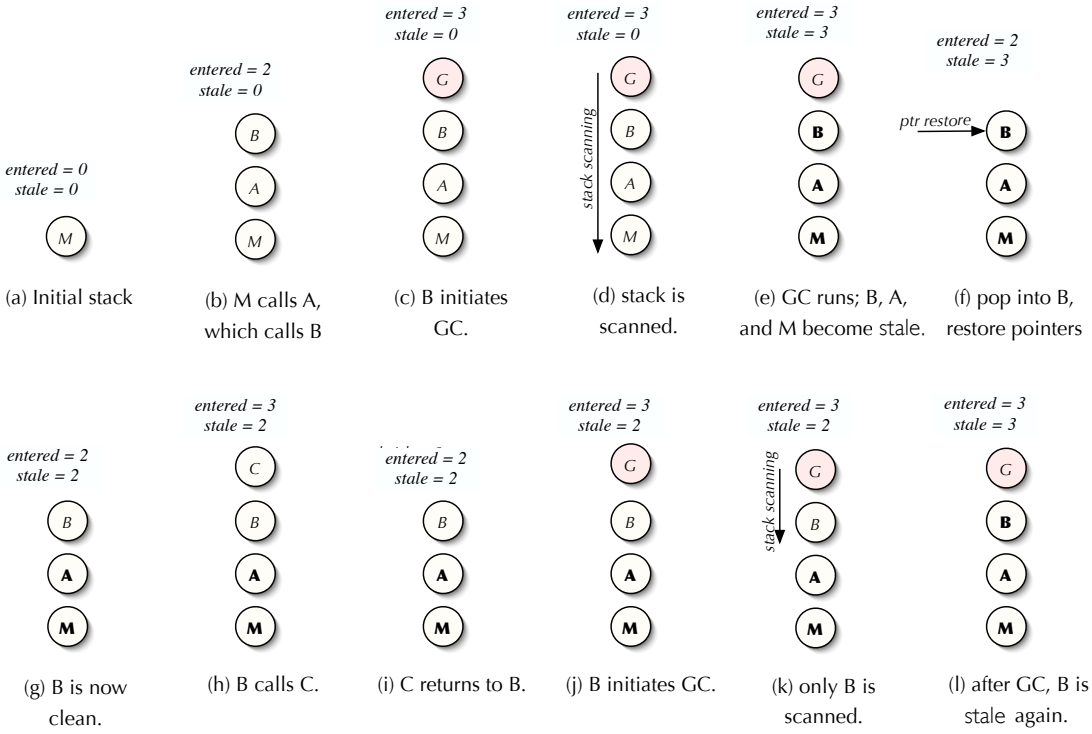


Figure 8. Example of lazy pointer stacks with pointer restoration.

the same as in pointer restoration – when `entered` < `CLEAR_HIGHEST_ORDER_BIT(stale)`, the frame is stale. When the frame is stale, it means that the lazy pointer stack already contains up-to-date pointers, and thus stack scanning of the thread finishes, returning control to GC. If the frame is clean, pointers are saved on the lazy pointer stack and stack scanning continues by returning to a previous frame on the thread's call stack.

We illustrate the algorithm on an example shown in Fig. 8. The program is composed of four methods M, A, B, C, and G, which is the invocation of the memory allocator which triggers garbage collection. We denote a stale frame using bold face.

(a) $[M]$ The main function.
(b) $[M \rightarrow A \rightarrow B]$ M calls A, which then calls B.
(c) $[M \rightarrow A \rightarrow B \rightarrow G]$ B requests memory and triggers a collection.
(d) $[M \rightarrow A \rightarrow B \rightarrow G]$ The stack is scanned and restored.
(e) $[\mathbf{M} \rightarrow \mathbf{A} \rightarrow \mathbf{B} \rightarrow G]$ The garbage collector runs, potentially moving objects referenced from the stack. All frames below that of the garbage collector are now stale as they contain pointers to the old locations of objects.
(f) $[\mathbf{M} \rightarrow \mathbf{A} \rightarrow \mathbf{B}]$ We return to a stale frame, $\mathbf{B}$, and must restore pointers.
(g) $[\mathbf{M} \rightarrow \mathbf{A} \rightarrow B]$ Execution proceeds in a clean frame.

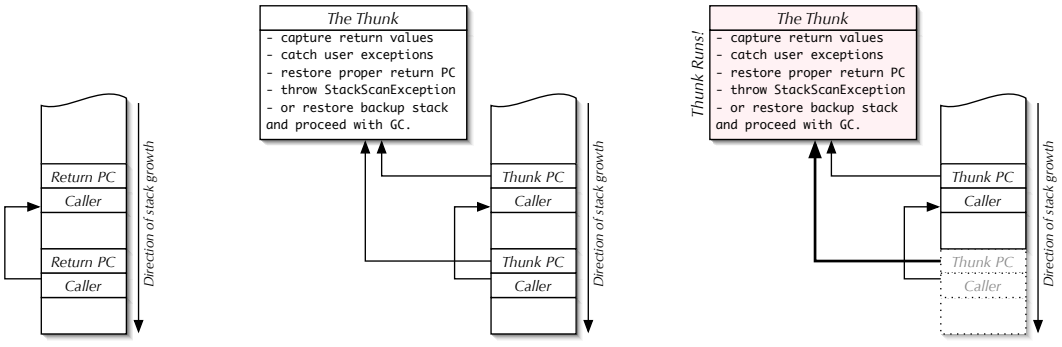
(h) $[\mathbf{M} \rightarrow \mathbf{A} \rightarrow B \rightarrow C]$ Call into C.
(i) $[\mathbf{M} \rightarrow \mathbf{A} \rightarrow B]$ Return to B. Because it is clean, we do not restore pointers.
(j) $[\mathbf{M} \rightarrow \mathbf{A} \rightarrow B \rightarrow G]$ B triggers another collection.
(k) $[\mathbf{M} \rightarrow \mathbf{A} \rightarrow B \rightarrow G]$ The stack is scanned only as far as B, since frames below it contain contain old, now invalid, pointers.

This gives us a complete system, with all the features necessary to accurately scan the stack and find pointers. However, this solution has overheads. It is necessary to execute code which maintains a stack height counter before and after each function call, even when no garbage collection is requested nor any exceptions are being thrown. As we will see, this approach does not outperform previous approaches.

4.2. Thunked Lazy Pointer Stacks

The expensive counting operations performed on function calls can be avoided by keeping information on stale/clean frames on the call stack itself. We propose to mark each stale frame by modifying the return address of the preceding frame to execute an additional code fragment (called a *thunk*) prior to returning to the stale frame. It is the thunk's job to restore pointers from the lazy stack. The original return addresses to the stale frames can be stored on the lazy stack, as it already contains an entry for each stale frame.

Fig. 9 illustrates the approach. We see a call where all return PCs have been set to refer to a thunk. The thunk is run when a function completes by either returning or throwing an application exception. It will save the return value or exception, restore the original return PC stored on the lazy stack, and throw a special exception (of type `StackScanException`). It is this exception that will trigger the pointer restoration code in the caller frame. A thunk can also be run during unwinding, when the garbage collector is trying to walk the stack. In this case, the thunk will simply stop the unwinding because the pointers held in older frames have potentially been outdated by earlier garbage collection (the up-to-date values of these pointers



(a) Ordinary callstack for C or C++ code.

(b) "Thunkified" callstack.

(c) If a function completes (either by return or throw), the thunk runs.

Figure 9. Thunked call stacks.

```
if (save()) {
   stop saving pointers, restore the stack and make gc run
} else {
   if (thunk entered because of application exception)
      save application exception
   else // thunk entered because of function return
      save return value
   restore proper return PC
   throw StackScanException;
}
```

Figure 10. Pseudocode for thunking logic called at return to a stale frame.

are still on the lazy pointer stack). Pseudocode for the thunking logic is given in figure Fig. 10. Modification of the return PC on the stack requires platform dependent code.

Fig. 11 shows the code generated by the translator for every call site. Pointers are stored in local variables and the call is wrapped in a C++ `try-catch` statement. To initiate stack scanning, the GC simply throws the distinguished `StackScanException`. This exception is immediately caught and the code guarded by the `save()` predicate is executed, saving all

```
void *ptr1, ..., *ptrn; // pointers in locals
try {
   functionCall();
} catch (const StackScanException&) {
   if (save()) {
      lazyPointerStack->pushFrame(nptrs);
      lazyPointerStack->pushPtr(ptr1);
      ...
      lazyPointerStack->pushPtr(ptrn);
      throw;
   } else {
      ptrn = lazyPointerStack->popPtr();
      ...
      ptr1 = lazyPointerStack->popPtr();
      lazyPointerStack->popFrame();
      if (had application exception) {
         throw application exception
      } else
         retrieve return value
   }
}
```

Figure 11. Pseudocode added to each function call to support thunked lazy pointer stacks.

pointer variables. The exception is propagated further until all frames have been scanned. During the traversal, every frame's return PC is set to point to a thunk.

After GC, whenever control would return to a function with a stale frame, the thunk runs instead, throwing again a `StackScanException`. That causes the pointers in the corresponding frame to be restored before normal execution resumes. This approach is illustrated in Fig. 11. To summarize, the implementation is as follows:

1. To scan the stack, we throw an exception. During stack scanning, thunks block the propagation of the `StackScanException` into stale frames.
2. When returning into a stale frame during normal execution, the thunk will throw the `StackScanException`, which will then be caught by code that restores pointers.
3. When an exception is thrown into a stale frame, the thunk will run, adjust the lazy pointer stack accordingly, and throw a `StackScanException` to restore pointers.

When an application exception is thrown, the C++ runtime unwinds the stack using the return PCs to determine whether a frame is able to handle exceptions of a given type. The thunks thus do not interfere with the C++ exception handling. If an application exception is thrown, the C++ runtime discovers that it is "handled" by a thunk, and thus returns control to the thunk. For C++, the pseudocode shown in Fig. 10 would include idiomatic C++ code ensuring the exception handler for application exception is included into C++ runtime exception table.

Although thunks do incur some execution overhead, they are only installed for the stack frames seen at the time of a garbage collection, and run once per frame. Hence, the thunk overhead is bounded by the stack height at the time of the collection.

4.3. Practical Considerations

Henderson [21] argues that his approach is fully portable as it uses only standard C. The same holds for explicit pointer stacks. However, both approaches to lazy pointer stacks require some platform-specific knowledge. They require access to the stack pointer and platform dependent use of *setjmp* to perform context switches, stack copying, and stack restoration. The thunked stacks require detailed knowledge of the calling convention. Return PC values must be saved and restored from the call stack. Thunk code must also be defined that saves return values from registers. The thunk implementation is invoked on function returns, and in the case of translators that use C++ exceptions, exception propagation as well. Hooking thunks to function returns requires knowledge of the calling convention, and catching exceptions in the thunk requires knowledge of the exception handling binary interface. In the version of our system that uses C++, we rely on the C++ compiler's stack walking code to use return PCs when looking for catch blocks. Indeed, if we use plain C code to implement Java exceptions and `StackScanException`, the platform dependencies related to C++ exceptions handling are eliminated. While the implementation of thunks requires some platform specific adaptation, we argue that the platform dependencies are small. The Ovm implementation of thunking has about 30 lines of platform specific code, supporting both IA32 and PPC architectures. The entire Ovm has just 1000 lines of platform specific code. Both numbers are quite small when

compared with an optimizing compiler. For example, in Jikes RVM [2] the IA32 and PPC specific code bases are 19 000 and 22 000 lines, respectively.

5. Compiler Extensions

We have presented four methods for accurate stack scanning, explicit pointer stacks and Henderson's frame lists, as well as two new techniques, counted and thunked lazy pointer stacks. All four techniques have been implemented in the Ovm virtual machine. We have found that in order to obtain a high-performance implementation, changes to Ovm's ahead-of-time compiler were needed.

5.1. Compiler Optimizations

Before we began seriously considering accurate garbage collection, Ovm's ahead-of-time compiler, `j2c`, took a relatively simple approach to performance. A lightweight whole program analysis, such as class hierarchy analysis [16] or rapid type analysis [5] was used to decide which methods to compile. This analysis was also used to devirtualize calls and optimize subtype tests. The compiler output was a single C++ file that contained all Java methods as static C++ functions. Java classes were mapped to POD C++ classes[¶], Java exceptions were mapped to C++ exceptions, though virtual calls and subtype tests were implemented directly (with our own vtable-like mechanism expressed using regular C++ fields and function pointers). This C++ file was handed over to the GCC compiler, which was then free to perform aggressive inlining and other optimizations. The combination of whole-program analysis in Java code and inlining in GCC is a great improvement over no inlining at all. However, this approach misses a large number of inlining opportunities. Inlining provides a level of context that can be fed back into a whole-program analysis to improve its results, but because GCC cannot communicate with the whole-program analysis, this loop is broken.

We found that our two-stage inlining strategy, combined with a non-generational garbage collector and relatively large heap, gave us reasonable performance. This level of performance allowed us to implement and optimize new virtual machine features, rather than focus solely on compiler work. However, this simple approach to optimization did not lend itself to accurate garbage collection with low overheads. When we added accurate stack scanning techniques, we saw slow-downs in excess of 20% in some SPECjvm98 benchmarks compared to our conservative configuration and a geometric mean slow-down of 9%. The causes of these overheads are as follows.

- Increase in prologue and epilogue of functions. When using the counting, Henderson, and explicit pointer stack approaches, the function call overhead is increased because of counting code, pointer stack operations, and conditional statements.

[¶]A POD class, or plain-old-data class, in C++ does not contain a C++ vtable or any C++ runtime type information.

- For the explicit pointer stack and Henderson approaches, local pointers are effectively spilled. Hence, all operations on pointers are slower.
- In the lazy pointer stack approaches, many optimizations that GCC would otherwise perform are inhibited. This is because of additional uses of pointer variables at safe points. Consider that every function call and poll check will have a use and a definition of every live pointer.

When accurate stack walking is enabled, the garbage collector should not see dead variables. If the collector attempts to update an uninitialized variable, memory corruption is possible, and if it updates a variable that is no longer live, it may fail to collect unreachable objects. In a traditional approach to accurate garbage collection, stack maps are generated after register allocation has been performed. The liveness information computed for register allocation can be reused to generate a stack map. In a compiler that targets C, live variables must be computed in an extra pass before C code is generated. The precision of this extra liveness analysis affects the quality of subsequent assembly code generation. Suppose that lazy pointer stacks are used, and we incorrectly mark a variable as live after its last use. If this variable is saved and restored at safe points, GCC is obligated to keep it live too.

The presence of code to explicitly save and restore pointers (either after GC or on every use) also inhibits alias analysis. GCC can see that the garbage collector may update pointers at a safe point, but it has no way to prove that these updates will preserve aliasing relationships. And, because GCC knows nothing about garbage collection, it has no reason to even suspect that aliases will be preserved. Consider what happens when we allow GCC to inline one Java method with safe points into another Java method with safe points. GCC cannot rename a formal parameter to the inlined function to an actual parameter, or otherwise observe that these variables alias each other. In addition, it has no way to combine the stack walking code of the two methods. In most cases, that means that after GCC performs inlining there may be multiple logical stack frames corresponding to a single stack frame in machine code. In the case of thunked lazy pointer stacks, the situation is far worse. Thunks are used to intercept the return from a function, and because an inlined function does not have an activation record of its own, there is no way to intercept the return from an inlined function. For this reason, GCC cannot be allowed to inline a method that contains safe points into any other method.

To improve the performance of the system, we added two new features to the Ovm ahead-of-time compiler. First, we wrote our own inliner. This turns out to improve the performance of all configurations, including the conservative configuration. It is particularly beneficial to the configurations that use accurate garbage collection because it eliminates many explicit pointer stack operations and lazy pointer stack guards. Second, we implemented copy propagation to eliminate any variable that will always alias another.

5.1.1. Bytecode inlining

Inlining at the bytecode level produces smaller code than inlining at the C++ level without sacrificing performance. This is because our inliner correctly assumes that inlining opportunities are plentiful, whereas GCC's inlining heuristics have been tuned for opportunities that are few and far between. Initially, we where surprised that inlining at both bytecode and

C++ levels produces better results still. In other words, simply adding bytecode-level inlining improves performance, but we still can use GCC's help due to two limitations in our initial inliner. First, our current bytecode inliner does not compute the type of a method's receiver. This means that when rapid type analysis is used, the bytecode inliner misses opportunities that will be exposed to GCC through the devirtualization step. Devirtualization is more effective after inlining, because call sites at inlined methods can be analyzed with additional context. Second, our bytecode inliner cannot inline at the boundary between application code and the Java code that implements the VM. It appears that bytecode-level inlining does simplify the program enough that GCC can inline more aggressively at this boundary. In fact, the fast path for object allocation can only be fully inlined when both techniques are used in conjunction.

5.1.2. Copy propagation

Unlike inlining, copy propagation does not improve runtime performance. However, it does provide a benefit in our particular case. In addition to removing inlined method parameters, copy propagation removes many temporary variables that our compiler introduces while translating from Java's stack-oriented bytecode to a more conventional format. This cleanup should facilitate subsequent optimizations such as bounds check elimination.

5.2. Fine Tuning

When lazy pointer stacks are used, we treat safe points where no pointers are live as a special case. With thunked lazy pointer stacks, a function call with no live pointers does not require a *try/catch* block at all. In the counted lazy pointer stack approach, empty safe points just require a simple guard, as shown below. In SPECjvm, 26% of all safe points are empty.

```
functionCall();
if (save()) {
   return;
}
```

While live variable analysis is required for lazy pointer stacks, it is not strictly needed for the eager approaches. Liveness can, however, be used to optimize this code. If a variable is not live at any safe point, it can be defined as an ordinary C local variable. And, if no variables are live across any safe point in a method, that method can avoid using the pointer stack entirely. Because we only emit poll checks on backward branches, many methods fall into this category (roughly 34% of all methods after inlining). Certain function calls do not need a guard even if pointers are live at the call. This includes function calls that: (a) are known not to return normally, (b) where the set of exceptions thrown is known, and (c) where exceptions are not caught locally.

We currently use this optimization at array bounds checks, explicit type checks, and implicit type checks when writing to an array of object references. Bounds checks in Ovm are emitted roughly as shown below:

```
Array *a;
```

```
jint idx;
if (idx ≥unsigned a->len) {
   throwOutOfBounds();
}
```

Notice that every bounds check has a function call that throws the out-of-bounds exception. This function call is known not to return, and is known to only throw one type of exception. Hence, if this exception is not caught locally, we can treat this call as an empty safe point. Only 5% of our runtime checks include non-empty safe points.

We have also optimized synthetic exception handlers that Ovm introduces at the boundary between application and virtual-machine code. The synthetic handlers are used to translate VM-internal exceptions to application-level exceptions such as *OutOfMemoryError* and *VMError*. We initially wrapped every call into the VM with an exception handler to perform this translation, but we found that this approach greatly increased the code bloat associated with lazy pointer stacks. In C++, we thus coalesce exception translation code within each method. As a result, 90% of these handlers wrap entire method bodies, so that safe points in the handler are empty. Section 7.4 describes the overhead we still suffer from exception translation, and the way that we plan to address that overhead in the near future.

We went even further in optimizing exception translation in our C implementation. In C, we translate an exception lazily, only when its user-domain version is needed, which is at first try-catch block in user-domain that is reached during exception propagation. Each try-catch block of every user domain method is thus accompanied with translation code, coalesced for all calls within the respective try-catch block.

Moreover, in addition to the exception itself, we also store its originating domain in a global variable. Therefore, the originating domain of an exception can be checked quickly, which is helpful as most of the exceptions in fact originate in the user domain, and thus do not need translation.

When compared to lazy pointer stacks, conservative garbage collection requires less up-front compiler work to establish good baseline performance. However, conservative garbage collection benefits from most of the additional compiler work that lazy pointer stacks requires.

6. Supporting C

The original `j2c` compiler used in [9] output C++ code primarily because of the language's support for exceptions. In C++, exceptions are optimized for fast entry into and exit out of a try block, which at the time of `j2c`'s design seemed compatible with the intended use cases of Java exceptions. Other features – such as inheritance and templates – were used as a convenience with the understanding that they could be trivially replaced by some combination of standard C features. However, we have since decided to remove the reliance on C++ for a combination of reasons:

- *Overhead.* Java programs can throw lots of exceptions – 213_javac and 228_jack are good examples – when this happens, C++ exceptions perform poorly. We also found that

in GCC, C++ try blocks incur overheads even in the non-exceptional case because the compiler often treats variables used in the catch block more carefully, thus inhibiting optimizations. Further, the exception handling code and meta-data emitted by GCC is quite large.

- *Portability.* Generating C code gives us access to a wider range of platforms.
- *Validation of lazy pointer stacks.* We want to show that lazy pointer stacks work even in a translator that outputs plain C.

At this time we have two versions of `j2c` – one that outputs C++ and one that outputs plain C, with the former being phased out. The main challenge to switching to plain C was implementing exceptions. We chose an implementation strategy based on simplicity and portability: intraprocedural exceptions become `goto`s, while exceptions that escape become `return`s. Immediately following each callsite, we emit code to check if the return was normal or exceptional. A code snippet with mapping of Java exceptions to C is shown below:

```
void m() {
   try {
      f();
   } catch (Exception e) {
      g();
   }
   return;
}
```

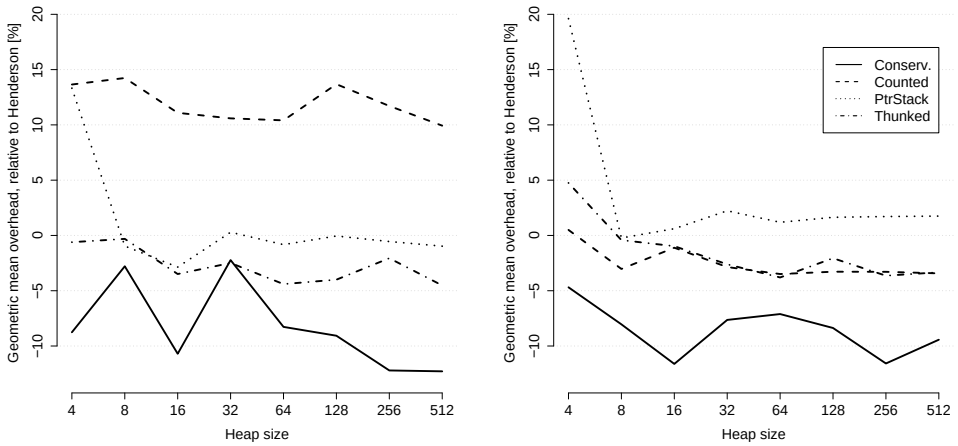
```
void m() {
   _exception = NULL;
   f();
   if (_exception) goto _tryblock1;
_label1: return;
_tryblock1:
   if (is_subtype_of(_exception, Exception)) {
      _exception = NULL;
      f();
      goto _label1;
   }
   return;
}
```

Our implementation manages preemption explicitly – thus the `_exception` is a global variable; for systems in which preemption is handled by the operating system it could just as well be a thread-local variable, using either GCC's thread-local storage support or a third-party library [23]. It should be noted that whether or not a virtual machine manages preemption explicitly, it will already be obligated to have fast thread-local storage for good allocator performance – as such we do not see the need for fast thread-locals to be a hindrance.

The only overhead of this approach is the post-call site check. However, the performance penalty of this check seems to be far outweighed by the performance overhead of C++ `try/catch` – as is shown in Section 7, our plain C exceptions outperform C++ exceptions in almost all configurations and benchmarks.

7. Experimental Evaluation

Our experimental evaluation was performed on a Pentium 4 machine at 3.8 GHz, 4GB of RAM, running Linux 2.6. All results are based on the SPECjvm98 benchmark suite. The results reported here are the arithmetic mean of ten individual runs of a VM with each test,



(a) Geom. mean of SPECjvm overheads (in C) (b) Geom. mean of SPECjvm overheads (in C++)

Figure 12. Geometric mean of execution time overhead of accurate collection, calculated over results from SPECjvm98 benchmarks. The overhead is relative to Henderson.

using a range of heap sizes from 2MB to 512MB. In order to avoid measuring the start-up fluctuations, we repeated each test 20 times within each run, reporting values only from the last repetition. In Figure 13 we show confidence intervals for mean execution times which verify that the measured numbers are stable.

We show the results from the smallest heap size in which each test ran successfully. The heap sizes do not include static data, which is instead pre-allocated by the `j2c` ahead-of-time compiler at compile time. We use Ovm's most reliable production garbage collector, called *mostlyCopying*, which has two operational modes. When accurate information is available it behaves as a traditional semi-space collector. Otherwise it runs in 'conservative' mode and pins pages referenced from the stack.

7.1. Overhead of Accurate Techniques

Figure 12 shows the percent overhead of using the four accurate stack scanning techniques (individual benchmarks are shown in Figures 14 to 20). Thunked lazy pointer stacks are significantly better than other techniques. With 512M heap, it has a geometric mean speed-up of 4.5% compared to a Henderson collector when Java exceptions are implemented in plain C, and 3.3% speed-up when Java exceptions are implemented in C++. In large heap configurations, many of the SPECjvm98 benchmarks only collect when the benchmark asks for it directly using *System.gc()*. Hence, results using the large heap configurations place more emphasis on the mutator overheads of the stack scanning techniques. Smaller heap configurations place more emphasis on the cost of stack scanning and collection time. Detailed overhead numbers for 32MB and 512MB heaps are shown in Figure 13.

Benchmarks in which accurate collection is faster than conservative collection can be explained by observing that in our mostlyCopying collector, an ambiguous pointer requires pinning and scanning an entire page. This strategy is usually a win because it means that the collector spends less time deciding how to handle ambiguous pointers. However, in some cases, we may have a pointer or pointer-like-value on the stack that points at a page in which most objects are no longer reachable. In these cases, the total cost of scanning the page, which can be high, would not have been incurred by an accurate collector. This explains speed ups we observe in our accurate configuration even compared to the conservative collector.

If we compare absolute performance of Ovm, measured as geometric mean of SPECjvm98 benchmarks with 512M heap, with Java exceptions implemented using C++ exceptions and using plain C, the plain C implementation is faster in all configurations but counting. With thunking, the speed up is 11%.

Sometimes thunking can lead to a speed up, as our garbage collector can work more efficiently if accurate pointer information is available. We profiled the time spent in the garbage collector,

	compress	db	jack	javac	jess	mpegaudio	mtrt
Conserv.	4.42±0.04	7.94±0.02	6.53±0.02	4.61±0.04	1.77±0.01	4.71±0.01	1.01±0.01
Counted	4.66±0.04	8.39±0.02	7.28±0.02	4.67±0	2.03±0	4.92±0.01	1.07±0
Henderson	4.67±0.02	8.63±0.02	7.24±0.01	5.08±0.01	2.31±0	4.99±0.01	1.09±0
PtrStack	5.37±0.03	8.58±0.01	7.14±0.09	4.94±0	2.17±0.01	5.29±0.05	1.12±0.01
Thunked	4.36±0.04	8.49±0.01	7.73±0.02	4.98±0.01	2.02±0	4.79±0.01	1.05±0

(a) 512MB heap, C++

	compress	db	jack	javac	jess	mpegaudio	mtrt
Conserv.	4.37±0.03	8.29±0.02	3.35±0.01	3.12±0.02	1.94±0.01	3.58±0.01	1.27±0
Counted	5.15±0.03	9 ±0.02	4.94±0.02	3.89±0.01	2.9 ±0	4.24±0.02	1.53±0
Henderson	4.47±0.01	8.92±0.02	4.04±0.01	3.77±0	2.55±0	3.95±0.01	1.4 ±0
PtrStack	4.97±0.02	8.96±0.01	3.89±0.01	3.51±0.02	2.31±0.02	4.15±0.01	1.39±0
Thunked	4.36±0.04	8.75±0.01	4.1 ±0.01	3.37±0	2.35±0	3.79±0.01	1.34±0

(b) 512MB heap, C

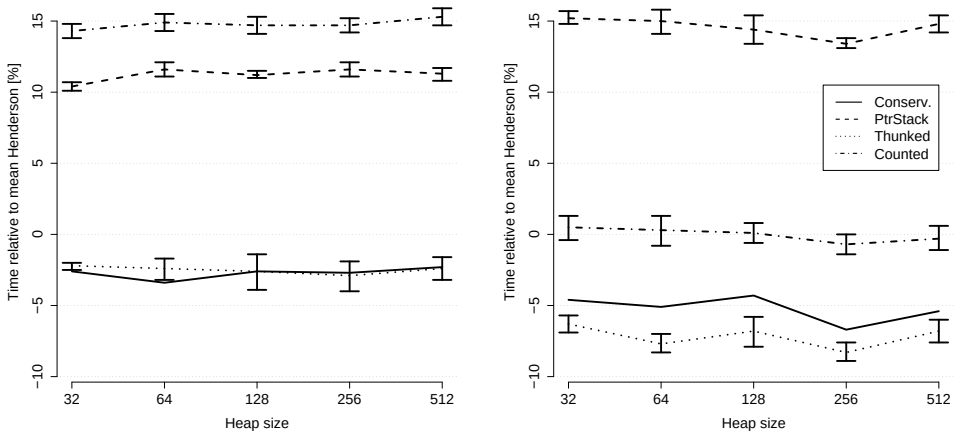
	compress	db	jack	javac	jess	mpegaudio	mtrt
Conserv.	4.49±0.05	8.57±0.01	5.84±0.02		2.01±0.03	4.7 ±0	
Counted	4.72±0.04	9.19±0.02	7.3 ±0.01	7.02±0.01	2.21±0	4.88±0.01	1.52±0
Henderson	4.7 ±0.02	9.42±0.02	7.35±0.01	7.46±0.01	2.45±0	5.01±0.01	1.53±0
PtrStack	5.42±0.02	9.37±0.04	7.22±0.04	7.41±0.01	2.4 ±0.01	5.28±0.01	1.55±0.01
Thunked	4.4 ±0.03	9.3 ±0.01	7.89±0.02	7.31±0.01	2.19±0	4.83±0.02	1.5 ±0

(c) 32 MB heap, C++

	compress	db	jack	javac	jess	mpegaudio	mtrt
Conserv.	4.4 ±0.04	9.04±0.01	3.51±0		2.2 ±0.05	3.57±0	2.38±0.12
Counted	5.17±0.02	9.93±0.03	4.95±0.01	6.69±0.01	3.1 ±0.01	4.22±0.01	2.06±0
Henderson	4.52±0.04	9.8 ±0.03	4.16±0.01	6.12±0.01	2.67±0.01	3.87±0.01	1.9 ±0
PtrStack	4.99±0.01	9.76±0.03	4 ±0.01	6.3 ±0	2.52±0.01	4.13±0.01	1.82±0
Thunked	4.42±0.01	9.62±0.01	4.3 ±0.01	5.89±0	2.55±0	3.76±0	1.82±0

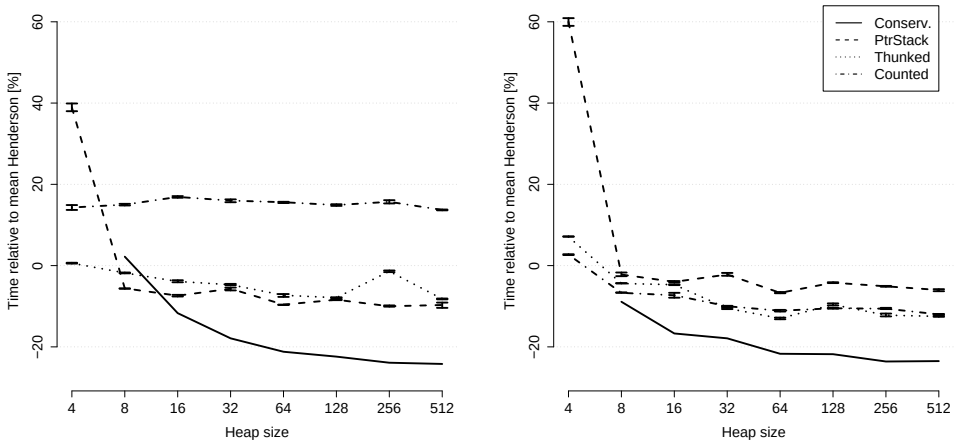
(d) 32 MB heap, C

Figure 13. Mean execution times of SPECjvm98 benchmarks with 95% confidence intervals. Both the means and the intervals' half widths are in seconds. As can be seen from the small width of the confidence intervals, the results are very stable.



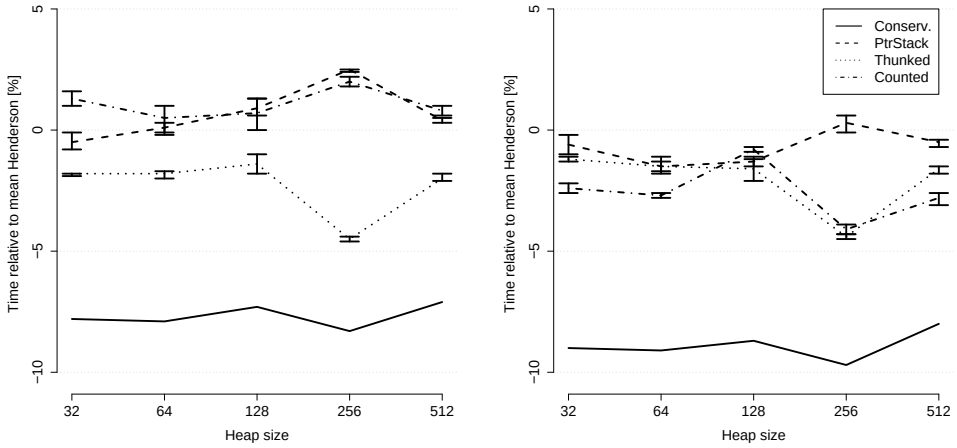
(a) 201_compress in C (b) 201_compress in C++

Figure 14. Execution time overhead of accurate collection in SPECjvm98's 201_compress relative to Henderson. Bars denote 95% confidence intervals for the mean.



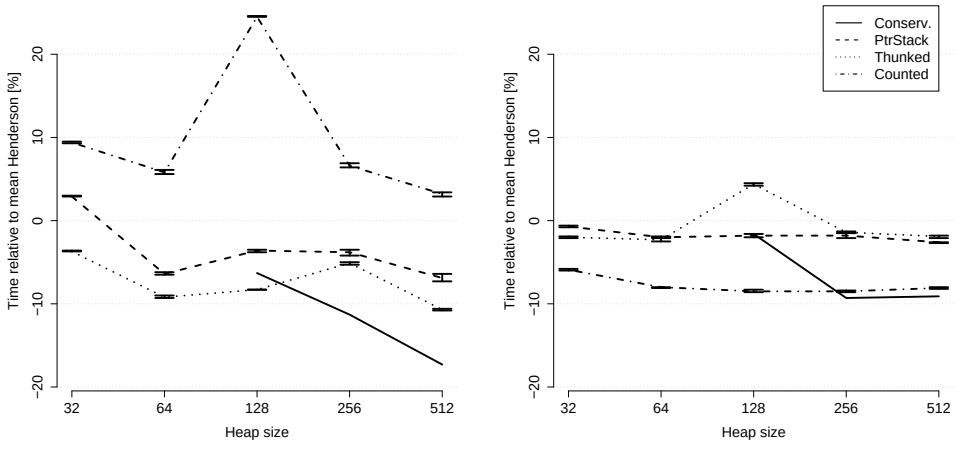
(a) 202_jess in C (b) 202_jess in C++

Figure 15. Execution time overhead of accurate collection in SPECjvm98's 202_jess relative to Henderson. Bars denote 95% confidence intervals for the mean.



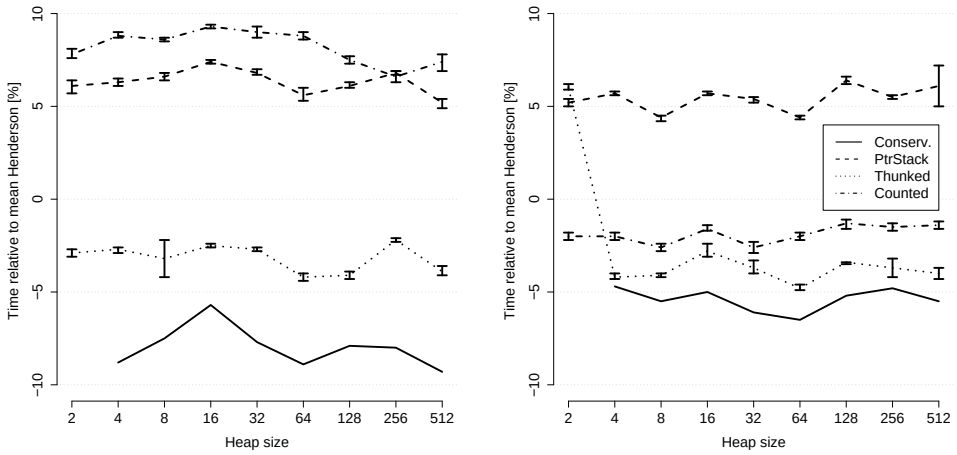
(a) 209_db in C (b) 209_db in C++

Figure 16. Execution time overhead of accurate collection in SPECjvm98's 209_db relative to Henderson. Bars denote 95% confidence intervals for the mean.



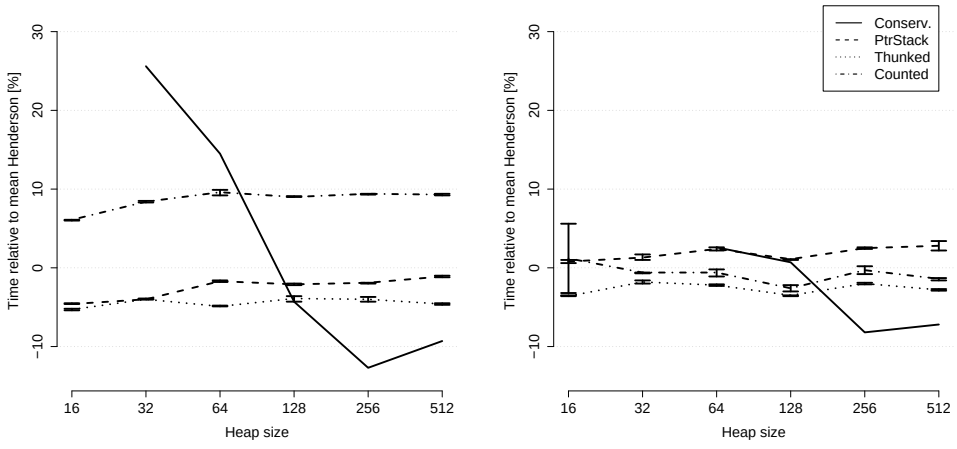
(a) 213_javac in C (b) 213_javac in C++

Figure 17. Execution time overhead of accurate collection in SPECjvm98's 213_javac relative to Henderson. Bars denote 95% confidence intervals for the mean.



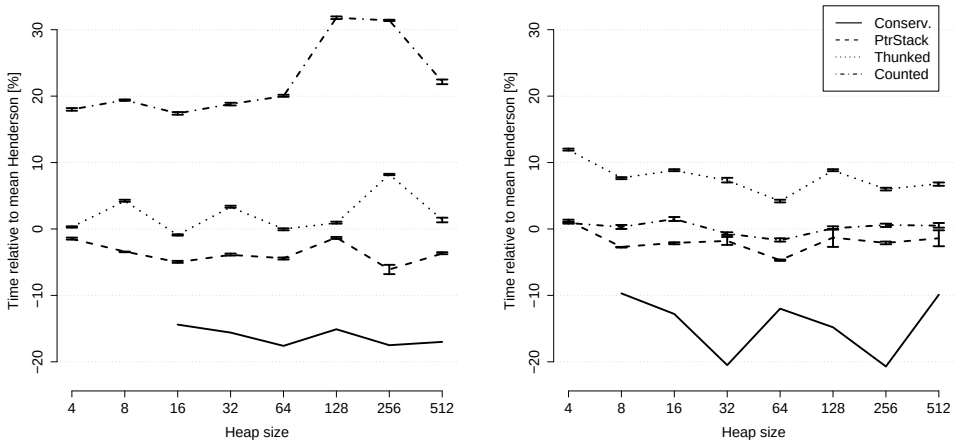
(a) 222_mpegaudio in C (b) 222_mpegaudio in C++

Figure 18. Execution time overhead of accurate collection in SPECjvm98's 222_mpegaudio relative to Henderson. Bars denote 95% confidence intervals for the mean.



(a) 227_mtrt in C (b) 227_mtrt in C++

Figure 19. Execution time overhead of accurate collection in SPECjvm98's 227_mtrt relative to Henderson. Bars denote 95% confidence intervals for the mean.



(a) 228_jack in C (b) 228_jack in C++

Figure 20. Execution time overhead of accurate collection in SPECjvm98's 228_jack relative to Henderson. Bars denote 95% confidence intervals for the mean.

and verified that the time used for GC in `mtrt` is shorter in the accurate configuration, consistently with the speed-ups shown in Fig. 12.

7.2. Code Size

All of the accurate techniques we have implemented increase code size. In the case of Ovm with `j2c` we can measure the size of the Ovm executable image. Fig. 21 shows the executables sizes in MBytes for the SPEC benchmark executable, both including and excluding the boot image (data, approximately 9MB). In C, the code size overhead of thunking vs. conservative is 33% and vs. Henderson 17%, when the boot image is excluded. Including the boot image, which makes more sense for embedded deployment, gives better relative overheads.

7.3. Comparing Virtual Machines

Our goal in implementing Ovm was to deliver a competitive Java implementation. We compare Ovm and stack walking configurations (Henderson and thunking) against HotSpot Client and Server versions 1.5 and 1.6, and against GCJ version 4.0.2, with a 512 MB heap. For each test, we have run each VM 20 times, repeating the test 50 times within each run and reporting only data from the last repetition. This excessive number of repetitions should allow HotSpot's just-in-time compiler to compile the code. Fig. 22 shows the results.

	C			C++		
	MBytes	vs. Conserv.	vs. Henderson	MBytes	vs. Conserv.	vs. Henderson
Conserv.	7.3MB		-12%	5.8MB		-13.4%
Counted	11MB	50.7%	32.5%	11.8MB	103.4%	76.1%
Henderson	8.3MB	13.7%		6.7MB	15.5%	
PtrStack	8.1MB	11%	-2.4%	6.5MB	12.1%	-3%
Thunked	9.7MB	32.9%	16.9%	11.7MB	101.7%	74.6%

(a) excluding boot image

	C			C++		
	MBytes	vs. Conserv.	vs. Henderson	MBytes	vs. Conserv.	vs. Henderson
Conserv.	16.9MB		-5.6%	15.4MB		-5.5%
Counted	20.6MB	21.9%	15.1%	21.4MB	39%	31.3%
Henderson	17.9MB	5.9%		16.3MB	5.8%	
PtrStack	17.7MB	4.7%	-1.1%	16.1MB	4.5%	-1.2%
Thunked	19.3MB	14.2%	7.8%	21.3MB	38.3%	30.7%

(b) including boot image

Figure 21. Code size of the executable in megabytes, showing overhead relative to conservative and Henderson.

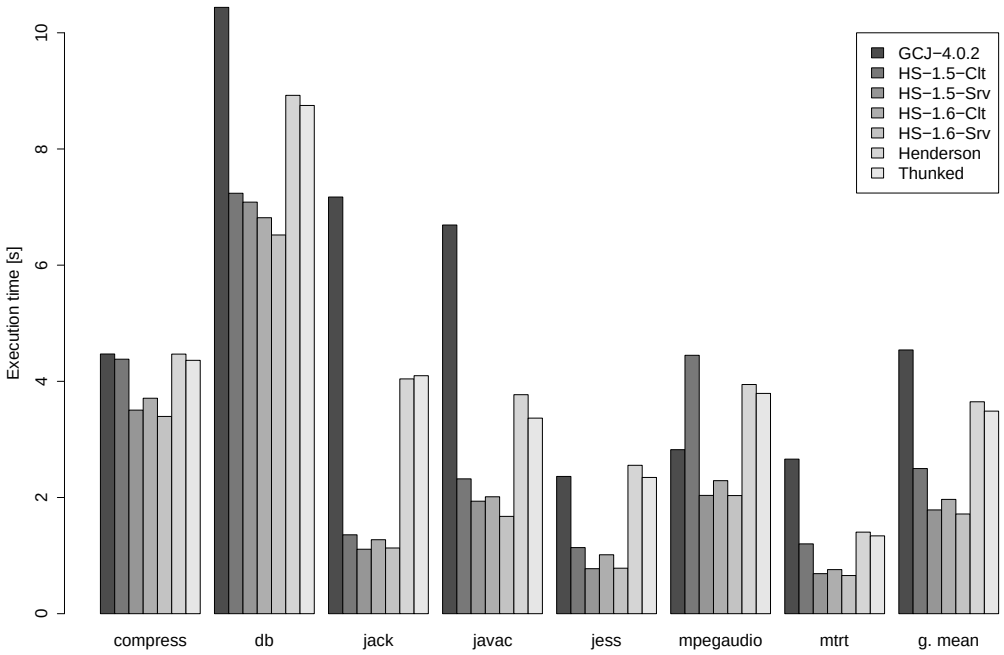


Figure 22. **Comparing Virtual Machines.** 512MB heap, arithmetic mean of 20 runs of 50 repetitions, taking only the last repetition from each run. Comparing two Ovm configurations (Henderson and thunking) with HotSpot Client 1.6 and 1.5, HotSpot Server 1.6 and 1.5, and GCJ 4.0.2. The measured Ovm configurations used plain C implementation of Java exceptions. Confidence intervals for the used values are shown in Fig. 23 and Fig. 13.

	compress	db	jack	javac	jess	mpegaudio	mtrt
GCJ-4.0.2	4.47±0.02	10.44±0.02	7.17±0.01	6.69±0.04	2.36±0	2.82±0	2.66±0
HS-1.5-Clt	4.38±0.01	7.24 ±0.01	1.36±0	2.32±0	1.14±0	4.45±0	1.2 ±0
HS-1.5-Srv	3.5 ±0.03	7.09 ±0.06	1.11±0.01	1.94±0.01	0.77±0.01	2.04±0.02	0.69±0.01
HS-1.6-Clt	3.71±0	6.82 ±0	1.27±0	2.01±0	1.01±0	2.29±0.01	0.76±0
HS-1.6-Srv	3.4 ±0.02	6.52 ±0.03	1.13±0.01	1.68±0.01	0.78±0	2.03±0	0.66±0.01

Figure 23. Mean execution times of SPECjvm98 benchmarks with 95% confidence intervals. Both the means and the intervals' half widths are in seconds. The data shown in this figure are used in Fig. 22.

7.4. Understanding the Overheads

This section describes sources of mutator overhead in our implementation of thunked lazy stack walking, and ways in which this overhead can be reduced. While on average mutator overhead is fairly low, it remains high in two SPECjvm98 benchmarks: `javac` and `jack`. We have attempted to track down this overhead using profiling tools and careful inspection of GCC version 4's output. We also explore ways in which this overhead can be minimized in the future. While none of the improvements discussed below are conceptually difficult, they do involve a small matter of programming.

We started by using `gprof` [19] to obtain profiling information for twenty iterations of the `javac` benchmark in both the conservative and C++ thunked configurations. We then examined the flat profile results, and found that in both cases roughly 20 methods accounted for half the CPU time, and that only two of these methods suffered a significant slowdown with thunked lazy stack walking. (A method called *ScannerInputStream.read* slowed down by 19%, while anther called *Scanner.xscan* slowed down by 40%.) We found three sources of overhead in these methods.

C++ Exception Dispatch Code. Up to two call-preserving registers may be used by C++ exception dispatch code generated by GCC. This appears to be the dominant cost in *ScannerInputStream.read*, where the presence of thunked stack walking code spills a loop induction variable from `%edi`. The generated code is significantly more complicated where two or more exception handlers are nested.

Extra Assignments of Return Values. We replace method calls with wrapper macros that add our lazy stack walking code. Those macros may lead to extra assignments of return values. When a method produces a value, the safe point code serves as the right-hand side of an assignment expression. The return value is saved in a macro-generated variable and returned to the macro's caller using GCC's statement-in-expression syntax. These extra assignments invariably remain after GCC's optimization, but are usually simple register-to-register moves. However, in *Scanner.xscan*, these extra variables and assignments do result in additional variables being spilled to the stack, leading to a marked slowdown (about 40%). It should be possible to eliminate this overhead by treating an assignment expression whose right-hand-side is a method call as a safe point, thus moving the real assignment inside the safe point try block.

Code Motion Across Exception Handlers. Code motion across exception handlers is sometimes less profitable than it would be in the absence of exception handlers. GCC occasionally performs extra work to ensure that variables that are not used by safe point code are available inside the safe point catch clause.

8. Validation: Real-time Garbage Collection

One of our goals in starting this project was to support real-time garbage collection (RTGC) in the real-time configuration of Ovm. While it is reasonable to think that lazy pointer stacks are able to deliver both the level performance and predictability needed in a real-time GC, it is difficult to have confidence in such a claim without an actual implementation. We therefore implemented a real-time garbage collector within Ovm using the lazy pointer stack technique [26]. The success in this endeavor increased our confidence in the general applicability of the techniques introduced here.

8.1. Basic Principles of RTGC

For garbage collection to be used in hard real-time applications, we need to ensure reasonable worst-case bounds on: (a) garbage collection pause times, (b) throughput, and (c) memory usage. Because events that arrive while the collector is operating cannot be handled until the collector yields, collector pause times must be bounded. Further, the mutator must not be interrupted too frequently. The effect on throughput must not be too severe, so that the application can handle events in a timely fashion. Finally, we must guarantee that memory allocation requests succeed. This requires finding a rate at which to run the collector to make it keep up with allocation. It also requires establishing a worst-case bound on memory usage. While it is possible to establish a worst-case bound on memory, we cannot guarantee that every allocation request will succeed when using conservative stack scanning. A conservative collector may in the worst case free no objects, causing subsequent allocation requests to fail. Worse, it is virtually impossible to predict when this worst case will occur.

Work on real-time collection can be traced back to Baker's incremental copying collector [6]. The central idea behind Baker's work is decreasing the intrusiveness of a collector by piggy-backing work onto mutator operations. To ensure consistency, a read barrier is inserted to perform copying, and allocation code is modified to perform a bounded amount of collection work. Because hard real-time tasks must be able to respond to events in a timely manner, what we seek here is to bound the worst-case execution. The worst-case execution in a program using Baker's collector involves a copy operation upon every read, and the largest unit of collection work on every allocation. Hence, even though individual pause times in Baker's collector can be made quite small, the impracticality of the worst case execution makes Baker's collector unsuitable for hard real-time applications. Put another way, the collector fails to bound the effect on throughput. The Baker collector can be said to be *work-based*, in the sense that work done by the mutator leads to work being done by the collector. Modern work-based collectors such as [28] have much tighter bounds on collector work and occupy an important niche in the real-time space – however, for these approaches to provide hard guarantees, all quanta

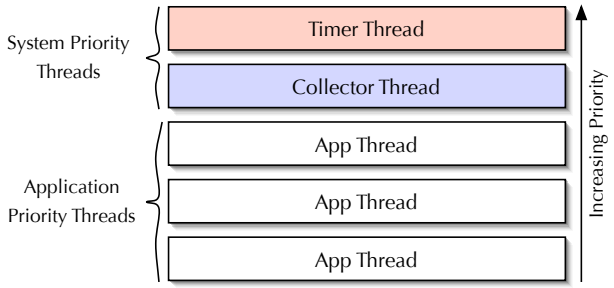


Figure 24. **RTGC in Ovm.** Two threads devoted to the collector run at system priorities. No changes to the scheduling infrastructure of Ovm were needed.

of collector work must be strictly bounded. Strict bounds on the quanta of collector work requires algorithms that are more concurrent and more incremental, which typically leads to higher overheads. For example, [28] requires that all objects are 32 bytes (larger objects are transformed in the compiler into tries or linked lists) and that all pointer moves on the stack are recorded by the collector. Bacon, Cheng, and Rajan [4] and Henriksson [22] investigate a different approach to real-time collection. In their work, the collector interleaves the mutator at regular intervals. Collector quanta are long enough for entire thread stacks to be scanned, and entire objects to be copied – leading to excellent collector performance. Only modest read or write barriers are used to maintain consistency, and an effort is made to implement constant-time allocation. As such, the worst-case bounds on execution time in the mutator become more realistic, allowing the collector to be used in hard real-time systems. This is the approach that we take in our own RTGC.

8.2. The Ovm RTGC

The Ovm real-time collector is a mark-sweep snapshot-at-the-beginning non-copying incremental garbage collector. The collector, just as the rest of the VM, is written in Java. We can thus utilize features of the Real-time Specification for Java in the implementation. The collector thread is a real-time Java thread with a priority high enough that unless it yields, it will not be interrupted by application threads. The timer that causes the context switches between the collector and application threads is simply a periodic task with higher priority than the collector. A second timer tick causes the collector to pause and allow the mutator to continue executing. Fig. 24 illustrates the use of Ovm's threading facilities.

When memory usage increases beyond a user-specified threshold, the collector thread is scheduled. Because of its priority, it immediately preempts any application threads. The collector then proceeds as follows:

1. Accurately scan the stack.
2. Scan the Ovm boot image which contains (immortal) objects internal to the VM.
3. Walk the heap starting with the roots found in the previous two steps.
4. Objects not marked in the previous three steps are reclaimed.

Following stack scanning, the collector periodically polls to see if it should yield to the mutator threads. The time between polls is small. During the marking phase, we poll after touching each pointer. Thus, the scanning of a large array can be interrupted at any time. In the sweep phase we poll in between pages. The amount of work to sweep a page is linear in the page size (in our test we used 2048 byte pages).

When the collector yields to the mutator, the mutator is free to change the structure of the heap. Changes are tracked using the Yuasa barrier [30], in which overwriting a pointer to an object results in that object being marked. Only object field and array stores are tracked in this way. Object field and array stores are sufficiently rare that instrumenting them does not typically lead to large overhead. Designing a real-time write barrier requires accounting for surprising subtleties. Traditional collectors make heavy use of barriers that are very fast most of the time, but occasionally quite slow. Much of the genius of such schemes lies in very simple and efficient tests that the barriers execute before doing anything else; the tests are designed to divert, with high probability, mutator code from executing the real barrier. But this approach has disastrous pathologies in which an unfortunate interleaving of collector and mutator execution may lead to the mutator executing a burst of slow paths. We avoid this by designing our barrier to exhibit worst-case behavior at all times – even when the collector is not running, the barrier performs a "mock" mark operation that is semantically a no-op, but exhibits similar timing behavior to a real mark operation. This simplifies the empirical evaluation of worst-case behavior, since there is no surprising "slow path" for writes that need to mark an object – indeed, our barrier takes the slow path every time. Nonetheless, our barrier is very light, leading to less than 7% slow-down in the average and not more than 11% for any individual benchmark [26].

It is the timer thread's task to inform the collector when to sleep, and to wake it. In Fig. 25 the beginning of a collection cycle for the `mtrt` benchmark is shown. Notice the first context switch to the collector. It is during this first quantum of collector activity that all thread stacks are scanned. Stack scanning took less than $250\mu s$ for this benchmark. Subsequent collector quanta are for stages two through four. The maximum collector pause time in this benchmark was $1.022ms$. Our target was to have the collector run for $1ms$ at a time. See Fig. 28 for a histogram of pause times. The Ovm RTGC is tuned to give at least 50% mutator utilization.

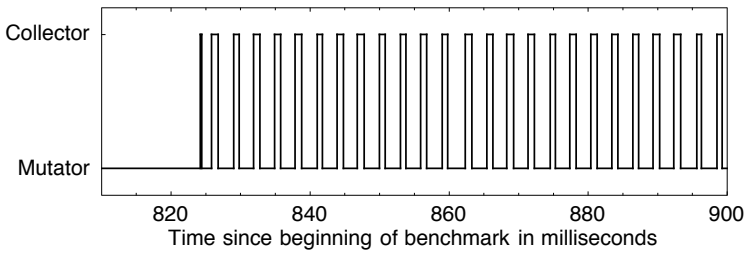


Figure 25. **Beginning of a collection cycle in the `mtrt` benchmark.** The X-axis shows time. The Y-axis is state: either the CPU is utilized by the collector or the mutator. Collector are regular and never exceed $1.022ms$ (see Fig. 28). The first collector pause is responsible for scanning all thread stacks. All of stack scanning – for all threads – took less than $250\mu s$.

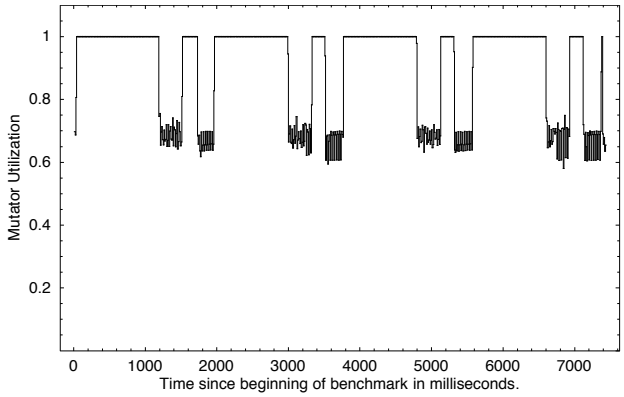


Figure 26. **Mutator utilization for the `javac` benchmark.**

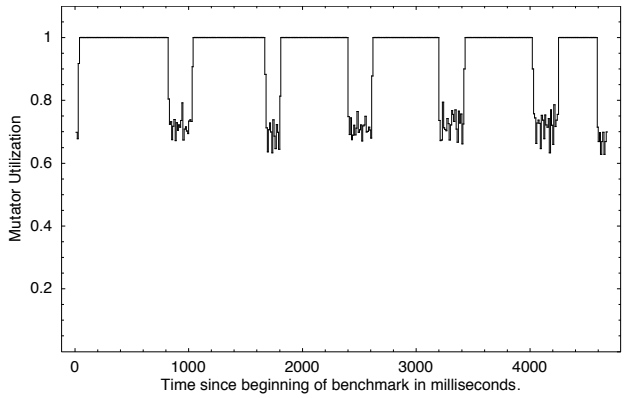


Figure 27. **Mutator utilization for the `mtrt` benchmark.**

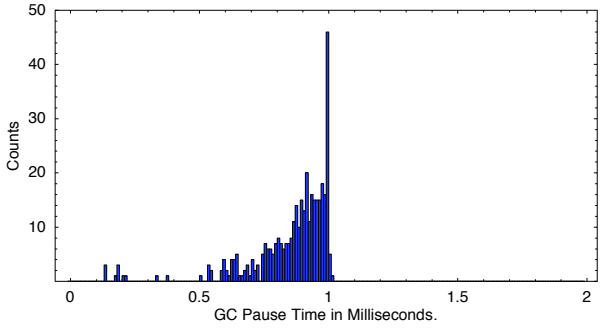


Figure 28. **Collector pause times for the `mtrt` benchmark.** The maximum pause time is $1.022ms$.

We measure mutator utilization during $10ms$ windows in the `javac` and `mtrt` benchmarks. These benchmarks were chosen because they exhibited the most collector activity. See Fig. 26 and Fig. 27 for the mutator utilization traces in the `javac` and `mtrt` benchmarks.

Accurate stack scanning takes less than $250\mu s$ for the `mtrt` benchmark, and the maximum collector pause time for this benchmark is $1.022ms$. It should be noted that a full-scale RTGC would use some form of incremental stack scanning – at the very least, scanning one thread's stack per increment instead of scanning all thread stacks in one, larger, atomic increment as we do here. The speed with which we scan `mtrt`'s thread stacks is particularly impressive given that `mtrt` runs multiple threads. Further details on our real-time collector are available in [26].

9. Related Work

Language implementations that use a C or C++ compiler as a back-end have a choice between conservative collection and the accurate techniques presented here. Techniques for accurate stack scanning in uncooperative environments have been previously described in detail in [27, 21]. Popular techniques for conservative garbage collection include the Boehm-Weiser collector [13] and various incarnations of mostly-copying collectors [10, 29, 11].

JamaicaVM uses explicit pointer stacks [27], but they differ from our implementation. First, objects referenced from the stack cannot move (in Ovm they can). Second, JamaicaVM uses write barriers on the pointer stack to enable incremental stack scanning. Ovm uses stop-the-world stack scanning. JamaicaVM may choose to place pointers on the pointer stack at safe points rather than upon each write. However, our lazy pointer stacks go further, only saving pointers when a stack scanning is actually requested, and additionally allowing for objects referenced by pointers on the stack to be moved.

The motivation behind generating C or C++ code is to create a competitive, portable language implementation with minimal effort. Jones, Ramsey, and Reig [24, 14] point out that what is really needed is a portable assembly language. They propose C--, which has a structured C-like syntax and comes complete with a runtime system that supports accurate garbage collection. C-- is attractive, but its stage of development cannot compete with GCC, especially for implementations of languages that map nicely onto C++, and where either conservative collection is acceptable, or the accurate stack walking techniques within this work are applicable. The Quick C-- compiler currently only supports IA32, while Ovm is available on IA32, PPC, and ARM. Using GCC allows us to generate fast code on each of these architectures.

For runtime systems that map functional languages to C, the problem of accurate garbage collection can be solved together with the problem of tail recursion using [7]. This technique is a heavier version of Henderson's, in that any local variable that may be interesting to the collector is part of a linked data structure whose nodes are stack-allocated.

It was possible to modify, with some effort, the GCC compiler to support accurate garbage collection. Diwan, Moss, and Hudson [17] describe changes to GCC version 2.0 to support accurate garbage collection in Modula-3. A further effort in this area is described in [15]. Our work has the advantage of not being strictly specific to GCC; the techniques described in this paper can be used with any compiler that has a reasonable binary interface for exceptions.

10. Conclusions

We started this work with a seemingly simple problem: enable Ovm to use accurate garbage collectors. Our original implementation used Henderson's technique, but it was quickly abandoned for a conservative collector due to performance overheads that were at the time unacceptable. The performance of Henderson's approach at the time was much worse than what we present here – but that is mainly because of the work we've done in Sec. 5, which significantly sped up all aspects of the system, but most notably, brought the performance of all accurate configurations closer to the performance of the conservative one. As well, we believe that even after our optimizations, the overhead of Henderson in the geometric mean is still too much; the goal of Ovm has always been to get as close as possible to the performance of commercial systems, and we have worked hard to get every percentage point wherever we can find it. This paper represents the best that we could do, after over a year of work, in designing an accurate stack scanning approach that fit our portability goals. Likely, the almost 10% overhead we see even with thunking – our best approach – in our C-based configuration is still too large for many, but we believe that it is good enough for our real-time configurations. It is noteworthy that we are still using the conservative collector for everything else.

But if we are to build a real-time collector, what accurate configuration should be used? Thunking gives the best mutator performance but increases pause times slightly; the other techniques degrade mutator performance but decrease pause times slightly. But – all of these techniques can be made incremental, thus eliminating the problem of pause time. For real-time systems what remains is the question of throughput when the collector is not execution an increment, and there, a 4% improvement can be a big deal. To make stack scanning incremental, we could easily employ sliding views [3] to scan each thread's stack separately; but we could go further and implement stacklets. We do not believe that it would be difficult to add stacklets to any of our approaches.

An even deeper question remains: should a VM be designed to use translation, as we have done, or should it have its own backend? We believe that our approach would be infeasible for a VM that performed dynamic loading – the loading would require an invocation of GCC, which is slow. But we are interested in ahead-of-time compilation only, and for this, we believe that the portability benefits of our approach are overwhelming. A single developer, who was not initially familiar with Ovm's J2c compiler internals, was able to port Ovm to two new hardware platforms in a handful of months. It is difficult to imagine that a VM such as HotSpot or Jikes RVM could be ported to a new platform like ARM or SPARC by one person, who starts with limited knowledge of the compilers, in just a few months. As well – systems that implement their own backend require heavy engineering work to ensure that the backend implements all of the optimizations that are essential for delivering the performance that programmers expect. We were able to skip this effort almost entirely; our optimizer just has inlining, copy propagation, and some smaller optimizations – in particular, we never implemented register allocation, loop unrolling, or common subexpression elimination – all of which are tricky to get right and essential for performance. Thus, we are quite sure that were we to repeat the exercise of building Ovm, a C compiler would probably still be our backend of choice.

We have extended the state of the art for accurate garbage collection in uncooperative environments. The lazy pointer stacks technique shows significantly improved performance

over previous techniques. Further, we show the need for optimizations such as inlining to be implemented in the high-level compiler for accurate garbage collection to pay off. To our knowledge, our experimental evaluation is the first to compare multiple approaches to accurate stack scanning within the same system. Of the previously known techniques, Henderson's approach fared the best in our tests; however, it showed more than twice the overhead of our new strategy. We claim therefore that our new approach improves the viability of accurate garbage collection in uncooperative environments and makes it easier for language implementors to use C++ and C as portable low-level representation.

REFERENCES

1. Jikes RVM. `http://jikesrvm.org`.
2. Bowen Alpern, Maria Butrico, Anthony Cocchi, Julian Dolby, Stephen J. Fink, David Grove, and Ton Ngo. Experiences porting the Jikes RVM to Linux/IA32. In Samuel P. Midkiff, editor, *Java Virtual Machine Research and Technology Symposium*, pages 51–64. USENIX, 2002.
3. Hezi Azatchi, Yossi Levanoni, Harel Paz, and Erez Petrank. An on-the-fly mark and sweep garbage collector based on sliding views. In Ron Crocker and Guy L. Steele Jr., editors, *OOPSLA*, pages 269–281. ACM, 2003.
4. David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 285–298, January 2003.
5. David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 324–341, October 1996.
6. H. G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.
7. Henry G. Baker. CONS should not CONS its arguments, part II: Cheney on the M.T.A. comp.lang.scheme.c newsgroup, 1994.
8. Jason Baker, Antonio Cunei, Chapman Flack, Filip Pizlo, Marek Prochazka, Jan Vitek, Austin Armbuster, Edward Pla, and David Holmes. A real-time Java virtual machine for avionics. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2006.
9. Jason Baker, Antonio Cunei, Filip Pizlo, and Jan Vitek. Accurate garbage collection in uncooperative environments with lazy pointer stacks. In *International Conference on Compiler Construction (CC)*, 2007.
10. Joel F. Bartlett. Compacting garbage collection with ambiguous roots. Research Report 88/2, Western Research Laboratory, Digital Equipment Corporation, February 1988.
11. Joel. F. Bartlett. Mostly-copying garbage collection picks up generations and C++. Technical Note TN-12, Western Research Laboratory, Digital Equipment Corporation, October 1989.
12. Hans-Juergen Boehm. Space efficient conservative garbage collection. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 197–206, June 1991.
13. Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software—Practice and Experience*, 18(9):807–820, September 1988.
14. C--. http://www.cminusminus.org.
15. Antonio Cunei. *Use of Preemptive Program Services with Optimised Native Code*. PhD thesis, University of Glasgow, December 2004.
16. Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 77–101. Springer-Verlag, 1995.
17. Amer Diwan, J. Eliot B. Moss, and Richard L. Hudson. Compiler support for garbage collection in a statically typed language. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 273–282, July 1992.
18. C. Flack, T. Hosking, and J. Vitek. Idioms in Ovm. Technical Report CSD-TR-03-017, Purdue University Department of Computer Sciences, 2003.
19. Free Software Foundation. Gnu binutils. `http://www.gnu.org/software/binutils/`.

20. Free Software Foundation. Gnu compiler collection. `http://gcc.gnu.org/`.
21. Fergus Henderson. Accurate garbage collection in an uncooperative environment. In *Proceedings of the ACM International Symposium on Memory Management*, pages 256–263. ACM, February 2002.
22. Roger Henriksson. *Scheduling Garbage Colection in Embedded Systems*. PhD thesis, Lund University, July 1998.
23. Kevin Hoffman. Speedy TLS. `http://www.kevinjhoffman.com/speedy-tls/`.
24. Simon Peyton Jones, Norman Ramsey, and Fermin Reig. C--: a portable assembly language that supports garbage collection. In *International Conference on Principles and Practice of Declarative Programming*, 1999.
25. Krzysztof Palacz, Jason Baker, Chapman Flack, Christian Grothoff, Hiroshi Yamauchi, and Jan Vitek. Engineering a common intermediate representation for the Ovm framework. *The Science of Computer Programming*, 57(3):357–378, September 2005.
26. Filip Pizlo and Jan Vitek. An empirical evaluation of memory management alternatives for Real-Time Java. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS 2006), 5-8 December 2006, Rio de Janeiro, Brazil*, 2006.
27. Fridtjof Siebert. Constant-time root scanning for deterministic garbage collection. In *International Conference on Compiler Construction (CC)*, pages 304–318, 2001.
28. Fridtjof Siebert. The impact of realtime garbage collection on realtime java programming. In *ISORC*, pages 33–40, 2004.
29. Frederick Smith and J. Gregory Morrisett. Comparing mostly-copying and mark-sweep conservative collection. In *Proceedings of the ACM International Symposium on Memory Management*, volume 34, pages 68–78. ACM, March 1998.
30. Taichi Yuasa. Real-time garbage collection on general-purpose machines. 11(3):181–198, 1990.