# Accurate Garbage Collection in Uncooperative Environments with Lazy Pointer Stacks

Jason Baker, Antonio Cunei, Filip Pizlo, and Jan Vitek

Computer Science Department
Purdue University
West Lafayette, IN 47906, USA
{baker29,cunei,filip,jv}@cs.purdue.edu

**Abstract.** Implementing a new programming language 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 the 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 many platforms, and good portability. For garbage-collected languages, however, they are not a perfect match as they provide no support for accurately discovering pointers to heap-allocated data. We evaluate the 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++ compiler and GCC as a back-end, and found that our technique outperforms existing approaches.

## 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 portability and extendibility. One popular implementation technique is to use a language translator to transform the code into an existing language, thus leveraging existing technology for part of the implementation. A time tested road has been to use C or C++ as a form of portable assembly language. This approach takes advantage of the portability of C++ and offloads 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++ compiler such as GCC[1] will not provide support for automatic memory reclamation. It is up to the language implementer to bridge the semantic mismatch between the features of the high-level language and what is available in the low-level language. In the case of garbage collection, implementers end up programming *around* the C++ compiler to ensure that garbage can be reclaimed.

The most straightforward solution to the problem is to use a *conservative* garbage collection algorithm. A conservative collector does not require cooper-

ation from its environment–it will traverse the stack and heap, and treat every value that could possibly be a pointer as a pointer. However, there are drawbacks. Unused memory may not be freed because of a non-pointer word that looks like a pointer to a dead object. Memory defragmentation is less effective, as values that may or may not be pointers cannot be freely updated when objects are moved. In the domain of real-time systems, all application deadlines must be met even in the presence of collector-induced pauses. For this, the garbage collector has to be predictable–a trait not found in conservative collectors.

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, the discussion generalizes to other high-level language translators. We evaluate several approaches to generating idiomatic 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 stack:** We present a class of new techniques for maintaining accurate information on the call stack. It promises lower overheads than previous work because the information is only materialized when needed leaving the native compiler free to perform more optimizations.
- **Catch & thunk stack walking:** We propose an efficient technique for saving and restoring the pointers on a call stack, which extends lazy pointer stacks by exploiting the exception handling features of the C++ language.
- **Implementation:** We implemented our technique in the Ovm framework. We report on our implementation and describe our optimizations.
- **Evaluation:** 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.
- **Validation:** We report on the implementation of a real-time garbage collector within Ovm using lazy pointer stacks to obtain accurate stack roots.

## 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 [4]. 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. Sources and documentation for Ovm are available from our website [5]. The reader is referred to [6, 4, 7] for further description of the framework.

*The J2c Compiler.* 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 [4]). 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++, which is then processed by the GCC compiler.

*The Ovm Threading subsystem.* Ovm uses user-level threading. 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. Our current poll check insertion strategy leads to less than 2.5% overhead. 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 $6\mu s$. For a detailed description of these results the reader is referred to [4].

We leverage poll checks in our implementation of memory management. Context switches only occur at poll checks and a small well-understood set of scheduler actions. The garbage collector can only run while a thread is blocked at a poll check, calling the memory allocator or invoking a scheduler operation. This makes for a simple definition of garbage collection safe points: in Ovm the only safe points are method calls and poll checks.

## 3 Previous Work: Accurate Stack Scanning

While it is often possible to assume that heap-allocated data structures have accurate type-descriptors that can be use by the garbage collector. However, while the native C/C++ compiler knows which locations in the call stacks contain pointers and which don't, but this knowledge is normally lost once the executable has been produced.

We found two previously used techniques for accurately scanning C call stacks. The simpler of the two uses an explicit stack of live pointers. The other approach, presented by Henderson [8], involves building a linked list of frames that contain pointers. This section describes both techniques in detail.

*Explicit Pointer Stacks.* While a C 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 language 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.

```
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

**Fig. 1.** Example of previous techniques for accurate stack traversal in C++ code. 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.

*Henderson's Linked Frames.* Henderson proposed a different approach, taking advantage of the fact that in C a local variable's address may either be passed to another function or stored in the heap. A C or C++ compiler handles these variables specially, to ensure 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 which 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). places The frame is placed into a linked list which can be subsequently traversed by the garbage collector.

Both techniques pin local variables into specific memory location that cannot easily be optimized by the C/C++ 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 allocator is limited, as pointers can not be moved around or stored in register.

## 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. We investigate here techniques that constructs the equivalent of a pointer stack on demand. We refer to this approach as *lazy pointer stacks*.

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 each thread. We shall assume that safe points are associated to call sites, *de facto* the case in Ovm as GCs are triggered by calls to the memory allocator. Other granularities are however possible.

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

**Fig. 2.** Lazy pointer stack construction: generated C++ code.

For every safe point, the language 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 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, we already have a copy of the register file, otherwise, we save it using `setjmp`. To be able to restore the stack, we simply save the original C++ call stack before unwinding and replace the unwound stack with its copy afterwards.

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. We developed two original solutions for this purpose: *pointer frame counting*, and the *safe point catch and thunk*.

### 4.1 Pointer Frame Counting

Updating pointers held in local variables can also be done lazily, as in Fig. 3. After collection, when each thread resumes execution, we cause each frame to perform pointer restoration as control returns to it, thanks to an additional post-safe-point guard which retrieves the possibly updated pointers. So, when 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.

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

**Fig. 3.** Prototype of lazy pointer stack with frame counting guard: generated C code.

```
// pointers in locals
void *ptr1, *ptr2, *ptr3, ...
height++;
functionCall();
height--;
if (save()) {
   if (height < auxHeight) {
      stop unwinding, restore the stack;
   } else {
      lazyPointerStack->pushFrame(nptrs);
      lazyPointerStack->pushPtr(ptr1);
      lazyPointerStack->pushPtr(ptr2);
      lazyPointerStack->pushPtr(ptr3);
      ...
      return;
   }
} else if (height < auxHeight) {
   ptr3 = lazyPointerStack->popPtr();
   ptr2 = lazyPointerStack->popPtr();
   ptr1 = lazyPointerStack->popPtr();
   ...
   lazyPointerStack->popFrame();
   auxHeight--;
}
```

(a) Function call with pointer frame counting

```
// pointers in locals
void *ptr1, *ptr2, *ptr3, ...;
try {
   functionCall();
} catch (const StackScanException&) {
   if (save()) {
      lazyPointerStack->pushFrame(nptrs);
      lazyPointerStack->pushPtr(ptr1);
      lazyPointerStack->pushPtr(ptr2);
      lazyPointerStack->pushPtr(ptr3);
      ...
      throw;
   } else {
      ptr3 = lazyPointerStack->popPtr();
      ptr2 = lazyPointerStack->popPtr();
      ptr1 = lazyPointerStack->popPtr();
      ...
      lazyPointerStack->popFrame();
      if (had application exception) {
         throw application exception
      } else {
         retrieve return values
      }
   }
}
```

(b) Function call, safe point catch and thunk

**Fig. 4.** Lazy pointer stack techniques.

The restoration logic has two key aspects. First, `restore()` must only evaluate to true 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 only after several 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.

We illustrate these issues in the following example. The program is composed of four method $M, A, B, C$, and $G$, which is the invocation of the memory allocator which triggers garbage collection. A frame is said to be *dirty* if it contains references to objects that were not updated after a collection (these reference are stale if the objects were moved). We denote a dirty 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 dirty as they contain pointers to the old locations of objects.
(f) $[\mathbf{M} \rightarrow \mathbf{A} \rightarrow \mathbf{B}]$ We return to a dirty 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) [$\mathbf{M} \rightarrow \mathbf{A} \rightarrow B \rightarrow G$] Consider what happens if $B$ 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.

We see that there is a frontier between dirty and clean frames. For dirty frames, the lazy pointer stack has correct pointers. For clean frames, the lazy pointer stack has no information. The *pointer frame counting* technique for accurate stack scanning is shown in Fig. 4(a). We keep track of the frontier between dirty frames and clean frames by using two counters: `height` is the height of the stack below the current frame; `auxHeight` is the height of the lazy pointer stack managed by the garbage collector, and it keeps track of the frontier between dirty and clean frames. We only restore pointers in a frame when the `height` becomes smaller than `auxHeight`. After stack scanning, the collector resets the `height` to its previous value, and `auxHeight` to the same value as `height`.

Non-local returns can interfere with this scheme. In our case the language translator uses C++ exceptions, so we have to handle them appropriately, providing a way to maintain the correct value of `height`. The solution is to compile `try` blocks as shown in Fig. 5. Before entry into the `try` block we save the value of `height`, and we restore it when an exception is caught. Because the exception may have traversed multiple dirty frames, we need to pop those from the lazy pointer stack. This is the purpose of the `while` loop. Finally, we check if the current frame is dirty; if so, we restore the pointers.

```
unsigned savedHeight = height;
...
try {
   ...
} catch (const ApplicationException&) {
   // restore counts
   height = savedHeight;
   while (height < auxHeight-1) {
      // ignore pointers in frame
      lazyPointerStack->popFrame();
      auxHeight--;
   }
   if (height < auxHeight) {
      ptr = lazyPointerStack->popPtr();
      lazyPointerStack->popFrame();
      auxHeight--;
   }
   // handle application exception
   ...
}
```

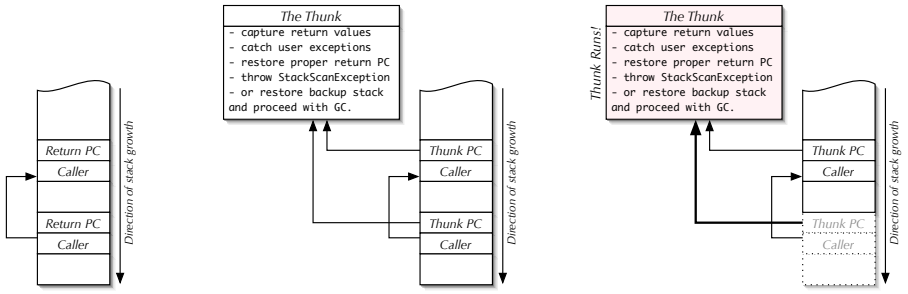
**Fig. 5.** Compiling try blocks to restore the pointer frame counts.

This gives us a complete system, with all the features necessary to accurately scan the stack and find pointers. However, this solution still has some overheads. In particular, it is necessary to execute code that counts the stack height before and after each function call.

### 4.2 Safe Point Catch and Thunk

On most systems, enclosing a group of instructions in a `try/catch` block does not require any additional code to be executed when no exceptions are thrown. The fast path has virtually no overhead. Instead of using a costly conditional to protect entry to the save and restore sequences, therefore, we can obtain better performance by implementing the accurate pointer guard using C++ exceptions.

To scan the stack, we simply throw a distinguished `StackScanException`. That exception is caught by the `catch` block in Fig. 4(b) and, if the `save()` predicate is set, the pointers in the receiver frame are saved. The exception propagates until all pointers are saved. During the traversal, for every dirty frame we install a helper routine, a *thunk*, by modifying the return address in the C++ call stack. After GC, whenever control would return to a function with a dirty frame, the thunk runs instead, throwing again a `StackScanException`.




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

(b) "Thunkified" callstack.

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

**Fig. 6.** Installing thunks in a C++ call stack.

That causes the pointers in the corresponding frame to be restored before normal execution resumes. At all times, thunks delimit the frontier between clean and dirty frames. This approach is illustrated in Fig. 4(b).

Thunks are invoked on both normal and exceptional returns. During the normal return sequence, the ordinary return sequence of the target frame results in the thunk being called. Fig. 6 shows this process. During exceptional returns, the C++ runtime unwinds the stack using the return PCs to determine if a frame is able to handle exceptions of a given type. We replace the return PC with the address of our thunk; therefore, we simply have to enclose the entry point of our thunk within a suitable exception handler. As a result, the thunk also automatically runs as a result of exceptional returns as well.

```
void thunk() {
   if (unwinding stack) {
      stop unwinding, restore the stack
   } else {
      if (target frame threw exception) {
         save exception
      } else {
         save return values
      }
      restore proper return PC
      throw StackScanException;
   }
}
```

**Fig. 7.** Thunk algorithm.

The thunk algorithm is shown in Fig. 7. If we are unwinding the stack in preparation for GC, it means we hit the frontier between clean and dirty frames. We stop (all pointers have been copied), restore the original stack, and proceed with GC. Otherwise, we save the current exception or the value returned by the routine from which we were returning when control was assumed by the thunk, we restore the original return PC, and throw a `StackScanException`, triggering the pointer restoration code in Fig. 4(b) and the retrieval of the original saved exception or return value. 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 in any case bounded by the stack height at the time of the collection.

### 4.3 Practical Considerations

Henderson [8] argues that his approach is fully portable as it uses only standard C. The same holds for explicit pointer stacks. Our approach uses some platform-

specific knowledge in order to save and restore the stack. The safe point catch and thunk technique also requires some knowledge of the C++ calling convention. While the implementation of thunks does require some platform specific code, we argue that the such dependencies are small. The platform specific code in our thunking implementation amounts to just 30 lines of code, supporting both IA32 and PPC architectures.

## 5 Compiler Optimizations

We have described four methods for accurate stack scanning: explicit pointer stacks and Henderson's frame lists, and two new techniques, frame counting and catch & thunk. All four have been implemented in the Ovm virtual machine. We found that the overheads imposed by those techniques can be reduced by carefully applying a number of optimization strategies.

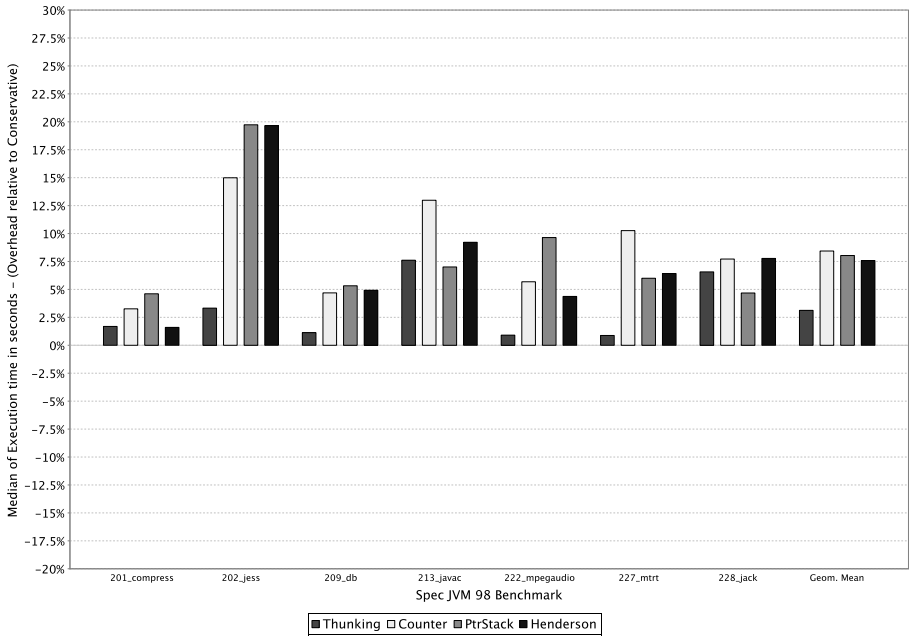
*General Optimizations.* The adoption of certain optimization techniques proved to have a positive effect on the performance of both eager and lazy stack scanning techniques. Inlining at the bytecode level, within our compiler, produced smaller code than relying exclusively on the GCC inliner, as we can rely on a higher-level knowledge of the program structure. Method devirtualization is more effective after bytecode inlining, as our analysis can rely on additional context. Refining our liveness analysis also proved beneficial. By reducing as much as possible the set of variables known to be live at each call site, we can both reduce the size of the added code, and improve the efficiency of the garbage collector at runtime.

*Fine Tuning* Further optimizations, specifically concerning our new techniques, also proved very valuable. When lazy pointer stacks are used, we treat safe points where no pointers are live as a special case. With catch & thunk, a function call with no live pointers does not require a *try/catch* block at all. In the pointer frame counting approach, empty safe points just require a simple guard, as show in below. In SPECjvm, 26% of all safe points are empty.

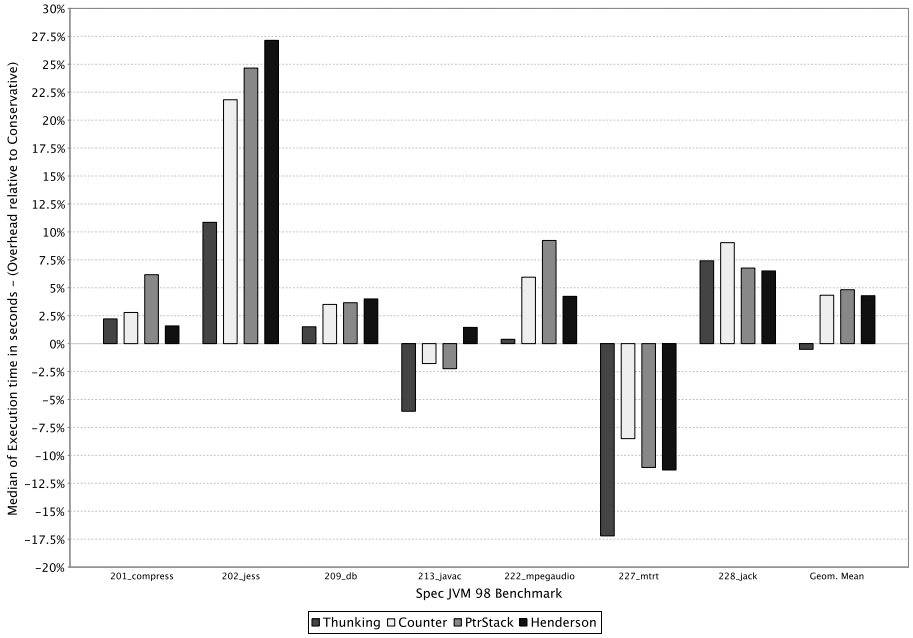
If no variables are live across any safe point in a method, that method can avoid using the pointer stack entirely. Because we only emit poll checks on backward branches, many methods fall into this category (roughly 34% of all methods after bytecode inlining). Certain function calls do not need a guard even if pointers are live at the call. This includes function calls that are known not to return normally, for which the set of exceptions thrown is known, and whose 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. Only 5% of our runtime checks include non-empty safe points. In certain cases, we can also coalesce multiple exception handlers within each method. That allows us to further contain the code size overhead.

## 6 Experimental Evaluation

Our experimental evaluation was performed on a Pentium 4 machine at 1600 MHz, 512 MB of RAM, running Linux 2.6 in single-user mode. All results are based on the SPECjvm98[9] benchmark suite. The results reported here are




(a) 256MB heap




(b) 32 MB heap

**Fig. 8. Overhead of accurate collection.** Median of 60 runs, with Ovm and bytecode inlining. Results normalized with respect to the Ovm Conservative collector. The numbers represented in the graphs are shown in Fig. 9.

| Median of Execution time in seconds - (Overhead relative to Conservative) | Thunking | Counter | PtrStack | Henderson |
|---|---|---|---|---|
| 201_compress | 1.69% | 3.26% | 4.61% | 1.60% |
| 202_jess | 3.33% | 14.99% | 19.73% | 19.65% |
| 209_db | 1.13% | 4.69% | 5.33% | 4.92% |
| 213_javac | 7.61% | 12.98% | 7.01% | 9.22% |
| 222_mpegaudio | 0.91% | 5.68% | 9.64% | 4.37% |
| 227_mtrt | 0.88% | 10.26% | 6.00% | 6.42% |
| 228_jack | 6.57% | 7.72% | 4.68% | 7.77% |
| Geometric Mean | 3.13% | 8.44% | 8.03% | 7.58% |

Median of Execution time in seconds - Spec JVM 98 Benchmark vs. Ovm Configuration - Ovm inlining, 65000 vars, copy propagation, Heap Size: 256m, Compiler: gcc-4.0.2, 60 runs - (Overhead relative to Conservative)

(a) 256MB heap

| Median of Execution time in seconds - (Overhead relative to Conservative) | Thunking | Counter | PtrStack | Henderson |
|---|---|---|---|---|
| 201_compress | 2.21% | 2.79% | 6.17% | 1.59% |
| 202_jess | 10.86% | 21.82% | 24.66% | 27.14% |
| 209_db | 1.51% | 3.51% | 3.66% | 4.00% |
| 213_javac | -6.05% | -1.78% | -2.24% | 1.45% |
| 222_mpegaudio | 0.39% | 5.95% | 9.23% | 4.23% |
| 227_mtrt | -17.21% | -8.50% | -11.08% | -11.31% |
| 228_jack | 7.41% | 9.04% | 6.76% | 6.51% |
| Geometric Mean | -0.51% | 4.33% | 4.83% | 4.29% |

Median of Execution time in seconds - Spec JVM 98 Benchmark vs. Ovm Configuration - Ovm inlining, 65000 vars, copy propagation, Heap Size: 32m, Compiler: gcc-4.0.2, 60 runs - (Overhead relative to Conservative)

(b) 32 MB heap

**Fig. 9.** Percent overhead of accurate garbage collection in Ovm. The overhead of accurate stack walking when using the safe point catch and thunk is only 3% for a large heap. For a small heaps, safe point catch and thunk actually improves performance, due to the conservative collector retaining too many objects.

| Median of Execution time in seconds | Conservative | Thunking | Counter | PtrStack | Henderson |
|---|---|---|---|---|---|
| 201_compress | 16.478 | 16.756 | 17.016 | 17.238 | 16.742 |
| 202_jess | 5.187 | 5.360 | 5.964 | 6.210 | 6.206 |
| 209_db | 18.468 | 18.677 | 19.334 | 19.452 | 19.378 |
| 213_javac | 12.747 | 13.718 | 14.402 | 13.641 | 13.922 |
| 222_mpegaudio | 12.007 | 12.116 | 12.690 | 13.165 | 12.532 |
| 227_mtrt | 5.067 | 5.112 | 5.587 | 5.372 | 5.393 |
| 228_jack | 14.394 | 15.340 | 15.505 | 15.068 | 15.513 |

Median of Execution time in seconds - Spec JVM 98 Benchmark vs. Ovm Configuration - Ovm inlining, 65000 vars, copy propagation, Heap Size: 256m, Compiler: gcc-4.0.2, 60 runs

(a) 256MB heap

| Median of Execution time in seconds | Conservative | Thunking | Counter | PtrStack | Henderson |
|---|---|---|---|---|---|
| 201_compress | 16.439 | 16.803 | 16.897 | 17.453 | 16.700 |
| 202_jess | 4.937 | 5.473 | 6.014 | 6.154 | 6.277 |
| 209_db | 30.172 | 30.628 | 31.231 | 31.277 | 31.379 |
| 213_javac | 18.202 | 17.100 | 17.877 | 17.793 | 18.466 |
| 222_mpegaudio | 12.047 | 12.094 | 12.764 | 13.160 | 12.557 |
| 227_mtrt | 7.003 | 5.799 | 6.408 | 6.227 | 6.211 |
| 228_jack | 14.253 | 15.310 | 15.541 | 15.217 | 15.181 |

Median of Execution time in seconds - Spec JVM 98 Benchmark vs. Ovm Configuration - Ovm inlining, 65000 vars, copy propagation, Heap Size: 32m, Compiler: gcc-4.0.2, 60 runs

(b) 32 MB heap

**Fig. 10.** Execution time in seconds of SpecJVM benchmarks on Ovm. The corresponding overheads, relative to the conservative collector, are shown in Fig. 9.

the median of sixty individual runs of tests from the SPECjvm98[9] benchmark suite (a first batch of ten, plus an additional fifty). We have run all the tests with both a large heap (256 MB) and a small heap (32 MB). 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.

*Overhead of Accurate Techniques.* Figure 8(a) shows the percent overhead of using the four accurate stack scanning techniques with a large heap (256MB). catch & thunk is significantly better than other techniques. It has a geometric mean overhead of 3% over a conservative collector, while the others are no better than 7%. 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. We also look at smaller heap configurations, which place more emphasis on the cost of stack scanning and collection time. Results using small heaps are shown in Fig. 8(b).

*Time Spent in GC.* 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, and verified that the time used for GC in `javac` and `mtrt` is shorter in the accurate configuration, consistently with the speed-ups shown in Fig. 8(b).

*Comparing VMs.* Our goal in implementing Ovm was to deliver a competitive Java implementation. We compare Ovm and stack walking configurations (conservative and thunking) against HotSpot Client and Server version 1.5, and GCJ version 4.0.2, with a 256 MB heap. Fig. 12 shows the results. Ovm's performance is highly competitive with that of the other systems, therefore our overhead results are likely not to be due to implementation inefficiencies.

| | |
|---|---|
| Conservative | 3,376 |
| Explicit Pointer Stack | 3,857 |
| Henderson | 4,031 |
| Counter | 9,320 |
| Thunking | 11,081 |

**Fig. 11. Code Size in KB.**

*Code Size.* All accurate techniques increase code size. In the case of Ovm with `j2c` we can measure the code component of the Ovm executable image. Fig. 11 shows the image sizes in KBytes for the SPEC benchmark executable image (includes the application as well as the VM code, and approximately 30MB of data.)

### 6.1 Understanding the Overheads

We used `gprof`[10] to obtain profiling information for the `javac` benchmark in both the conservative and catch and thunk configurations, and found three main sources of overhead in these methods.

*Exception Dispatch Code.* Up to two call-preserving registers may be used by exception dispatch code generated by GCC. This appears to be the dominant

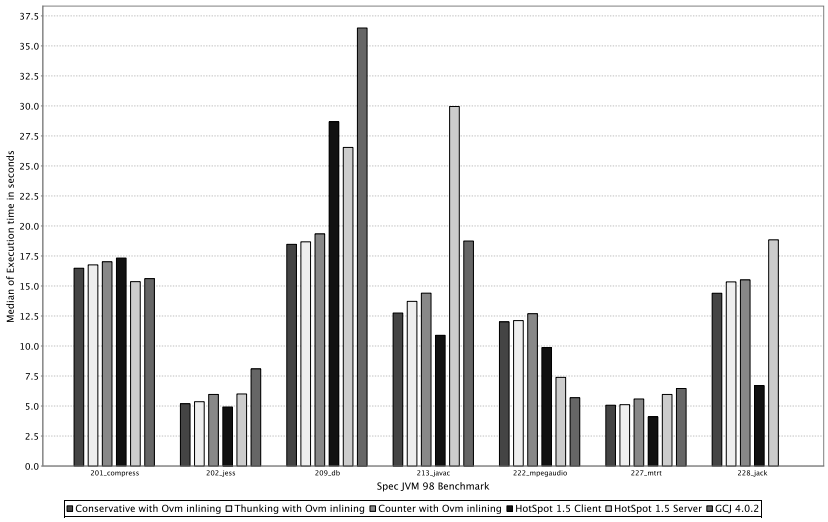



**Fig. 12. Comparing Virtual Machines.** 256MB heap, median of 60 runs. Comparing two Ovm configurations (conservative and thunking) with HotSpot Client 1.5, HotSpot Server 1.5 and GCJ 4.0.2.

cost in *ScannerInputStream.read*, where the presence of catch and thunk 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 occasionaly performs extra work to ensure that variables that are not used by safe point code are available inside the safe point catch clause.

## 7 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 [11]. The success in this endeavor increased our confidence in the general applicability of the techniques introduced here.

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 thus used 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. When memory usage increases beyond a user-specified threshold, the collector thread is scheduled. Because of its priority, it immediately preempts any application threads. It then accurately scans the stack, the Ovm boot image, which contains immortal objects used by the VM, and then incrementally traverses the heap reclaiming unused objects. Accurate stack scanning takes less than $250\mu s$ for the `mtrt` benchmark, and the maximum collector pause time for this benchmark is $1.022ms$. Further details on our real-time collector are in [11].

## 8 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 [14, 8]. Popular techniques for conservative garbage collection include the Boehm-Weiser collector[2] and various incarnations of mostly-copying collectors[15–17].

JamaicaVM uses explicit pointer stacks [14], 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[18, 19] 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.

It possible to modify, with some effort, the GCC compiler to support accurate garbage collection. Diwan, Moss, and Hudson [20] describe changes to GCC version 2.0 to support accurate garbage collection in Modula-3. A further effort in this area is described in [21]. 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.

## 9 Conclusions

We have extended the state of the art for accurate garbage collection in uncooperative environments. The lazy pointer stacks technique shows less than 10% overhead in any given benchmark, and 3% overhead overall. Further, we demonstrated 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, we 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++ as portable low-level representation.

## References


1. Free Software Foundation: Gnu compiler collection. (`http://gcc.gnu.org/`)
2. Boehm, H.J., Weiser, M.: Garbage collection in an uncooperative environment. Software—Practice and Experience **18**(9) (1988) 807–820
3. Boehm, H.J.: Space efficient conservative garbage collection. In: Proceedings of the ACM Conference on Programming Language Design and Implementation. Volume 26. (1991) 197–206
4. Baker, J., Cunei, A., Flack, C., Pizlo, F., Prochazka, M., Vitek, J., Armbuster, A., Pla, E., Holmes, D.: A real-time Java virtual machine for avionics. In: Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2006), IEEE Computer Society (2006)
5. Vitek, J., Baker, J., Flack, C., Fox, J., Grothoff, C., Holmes, D., Palacz, C., Pizlo, F., Yamauchi, H.: The Ovm Project. (`http://www.ovmj.org`)
6. Palacz, K., Baker, J., Flack, C., Grothoff, C., Yamauchi, H., Vitek, J.: Engineering a common intermediate representation for the Ovm framework. The Science of Computer Programming **57**(3) (2005) 357–378
7. Flack, C., Hosking, T., Vitek, J.: Idioms in Ovm. Technical Report CSD-TR-03-017, Purdue University Department of Computer Sciences (2003)
8. Henderson, F.: Accurate garbage collection in an uncooperative environment. In: Proceedings of the ACM International Symposium on Memory Management. Volume 38., ACM (2002) 256–263
9. SPEC: SPECjvm98 benchmarks (1998)
10. Free Software Foundation: Gnu binutils. (`http://www.gnu.org/software/binutils/`)
11. Pizlo, F., Vitek, J.: 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)
12. Baker, H.G.: List processing in real time on a serial computer. Communications of the ACM **21**(4) (1978) 280–294
13. Bacon, D.F., Cheng, P., Rajan, V.T.: A real-time garbage collector with low overhead and consistent utilization. In: Conference Record of the ACM Symposium on Principles of Programming Languages. Volume 38. (2003) 285–298
14. Siebert, F.: Constant-time root scanning for deterministic garbage collection. In: International Conference on Compiler Construction (CC). (2001) 304–318
15. Bartlett, J.F.: Compacting garbage collection with ambiguous roots. Research Report 88/2, Western Research Laboratory, Digital Equipment Corporation (1988)
16. Smith, F., Morrisett, J.G.: Comparing mostly-copying and mark-sweep conservative collection. In: Proceedings of the ACM International Symposium on Memory Management. Volume 34., ACM (1998) 68–78
17. Bartlett, J.F.: Mostly-copying garbage collection picks up generations and C++. Technical Note TN-12, Western Research Laboratory, Digital Equipment Corporation (1989)
18. Jones, S.P., Ramsey, N., Reig, F.: C--: a portable assembly language that supports garbage collection. In: International Conference on Principles and Practice of Declarative Programming. (1999)
19. C--: (http://www.cminusminus.org)
20. Diwan, A., Moss, J.E.B., Hudson, R.L.: Compiler support for garbage collection in a statically typed language. In: Proceedings of the ACM Conference on Programming Language Design and Implementation. Volume 27. (1992) 273–282

21. Cunei, A.: Use of Preemptive Program Services with Optimised Native Code. PhD thesis, University of Glasgow (2004)