Accurate Garbage Collection in Uncooperative Environments with Lazy Pointer Stacks

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++ programming language is the target of choice due to its portability and the availability of efficient compilers. For garbage-collected languages however, C++ is not an ideal match because it provides no support for accurately discovering pointers to heap-allocated data. This poster proposes a new technique, referred to as lazy pointer stacks, for performing accurate garbage collection in such uncooperative environments. We have implemented this technique in the Ovm Java virtual machine with our own Java-to-C++ compiler and using GCC as a back-end. When compared against an efficient mostly-copying conservative collector our technique has a mean overhead of 3% for large heaps (256MB) and no overhead for smaller heaps (32MB).

```
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

Review of previous techniques. In (a) we see a function **Foo** compiled to C without any instrumentation for accurate stack scanning. This code will require conservative stack scanning by the collector. In (b) we see explicit pointer stacks, and in (c) we have Henderson's linked lists. Both (b) and (c) require additional code to be executed upon entry and exit of every function. Further, no pointer access can be register allocated. ***The goal of this research is to eliminate these overheads.***

```
// pointers in locals
void *ptr1, *ptr2, *ptr3, ...;
functionCall();
if (save()) {
    lazyPointerStack->pushFrame(nptrs);
    lazyPointerStack->pushPtr(ptr1);
    lazyPointerStack->pushPtr(ptr2);
    lazyPointerStack->pushPtr(ptr3);
    ...
    return;
}
```

(a) Function call guarded with the accurate pointer guard

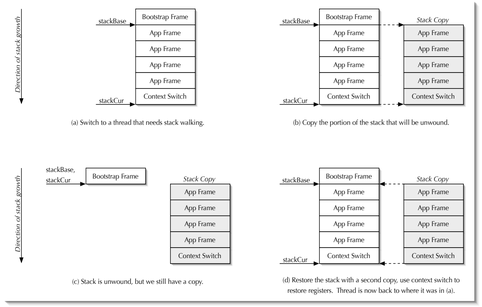
```
// pointers in locals
void *ptr1, *ptr2, *ptr3, ...
height++;
functionCall();
height--;
if (save()) {
    if (height < auxHeight) {
        stop unwinding, restore the stack;
    } else {
        lazyPointerStack->pushFrame(nptrs);
        lazyPointerStack->pushPtr(ptr1);
        lazyPointerStack->pushPtr(ptr2);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

(b) 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 (bad application exception) {
            throw application exception
        } else {
            retrieve return values
        }
    }
}
```

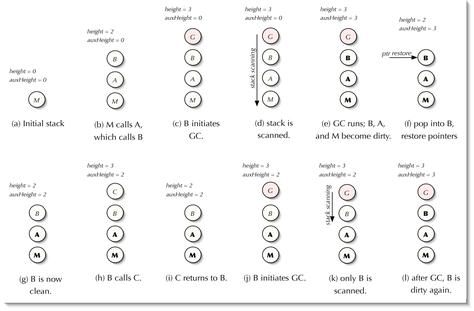
(c) Function call with safe point catch and thunk

Our proposed techniques. In (a) we see the basic technique: every function call is followed by an accurate pointer guard, which saves all pointers onto the lazy pointer stack when the function returns and the **save()** predicate is *true*. This predicate is *false* during normal operation. When the collector starts, it sets the predicate to *true* and returns. Because every function call is guarded, the effect will be that the stack is destroyed and all pointers are saved. This naive solution has two problems: 1) the stack is destroyed, and 2) we cannot move objects referenced from the stack because we have no way of modifying the pointers after GC. The solution to (1) is that we make a copy of the thread stack and register file prior to stack scanning; we restore these after the stack is destroyed. We have two solutions to (2); see the approaches shown in (b) and (c). Both approaches cause pointer restoration code to run upon the first return into a frame after the collector completes. Pointer frame counting in (b) does this by using an extended predicate based on counters, resulting in significant overhead. In (c) we show the safe point catch and thunk mechanism, which uses exceptions and thunks instead of predicates and counters: to trigger stack scanning the collector throws a special exception; to restore pointers we install thunks of function return addresses in the stack.



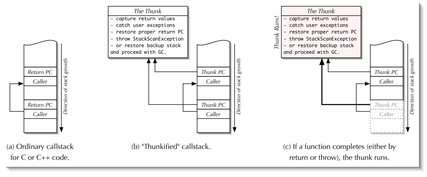
(a) Switch to a thread that needs stack walking.
(b) Copy the portion of the stack that will be unwound.
(c) Stack is unwound, but we still have a copy.
(d) Restore the stack with a second copy, use context switch to restore registers. Thread is now back to where it was in (a).

An illustration of the stack copying technique used to restore the stack after scanning. In (a) we show the thread before scanning. In (b) we copy the stack. In (c) the stack is scanned, but in (d) the destroyed stack is restored from the copy.



(a) Initial stack
(b) M calls A, which calls B
(c) B initiates GC.
(d) stack is scanned.
(e) GC runs; B, A, and M become dirty.
(f) pop into B, restore pointers
(g) B is now clean.
(h) B calls C.
(i) C returns to B.
(j) B initiates GC.
(k) only B is scanned.
(l) after GC, B is dirty again.

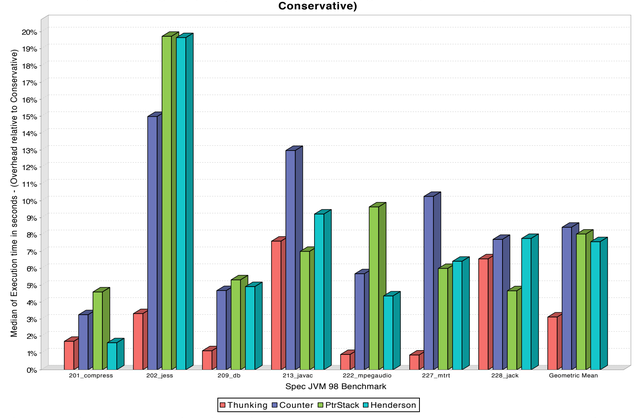
Pointer frame counting in action. We keep track of the program stack and the lazy pointer stack in real time using the *height* and *auxHeight* counters, respectively. When the collector is invoked, all frames for which *height* > *auxHeight* are scanned. After GC, any frames that cause *height* < *auxHeight* after function return do not proceed until pointers are restored. This facilitates the two operations that the collector needs: retrieving pointers from the mutator, and then modifying them after objects are moved.



(a) Ordinary callstack for C or C++ code.
(b) "Thunkified" callstack.
(c) If a function completes (either by return or throw), the thunk runs.

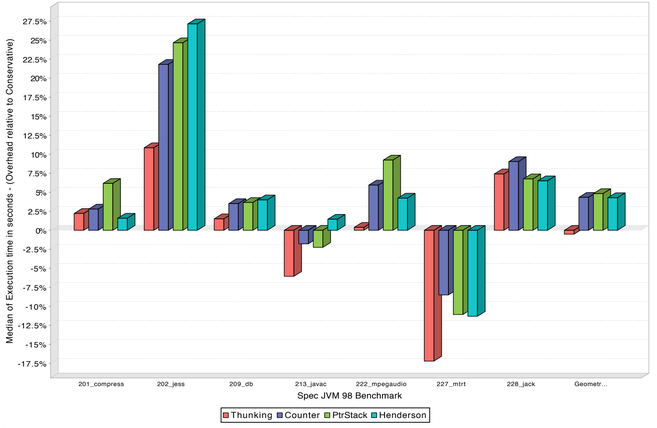
The performance of counters is quite poor because of the overhead of counting on function entry and exit, and comparing the counters on exit. We can improve performance by placing stack scanning code in a C++ catch statement; throwing an exception will be used to trigger scanning. This means that no predicate is needed. But what about pointer restoration? We implement this by installing thunks on function returns; these thunks will throw an exception that triggers restoration. This figure shows thunks in action. In (a) we see an ordinary C++ call stack. In (b) we see the thunk installed. In (c) we see a function return causing the thunk to run. The thunk then performs bookkeeping and throws the exception that signals restoration.

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)



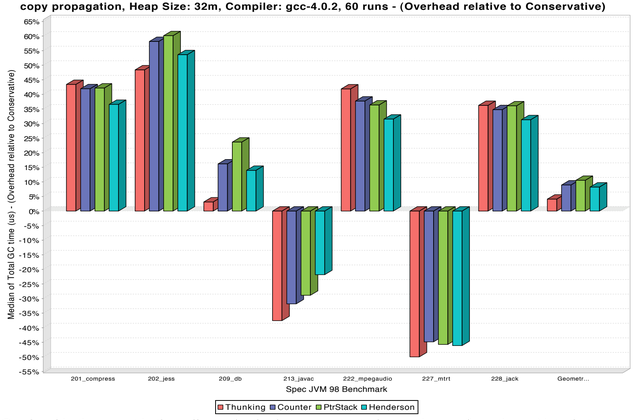
Performance of the four mechanisms for achieving accurate stack scanning, measured as overhead relative to the conservative collector for a large heap size (256 MB). Note that in the geometric mean we see that the overhead of the *Thunking* (safe point catch and thunk) configuration is substantially better than the other three. It is within three percent of the conservative configuration, while the Henderson configuration (the best of the other three) seven percent level.

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)



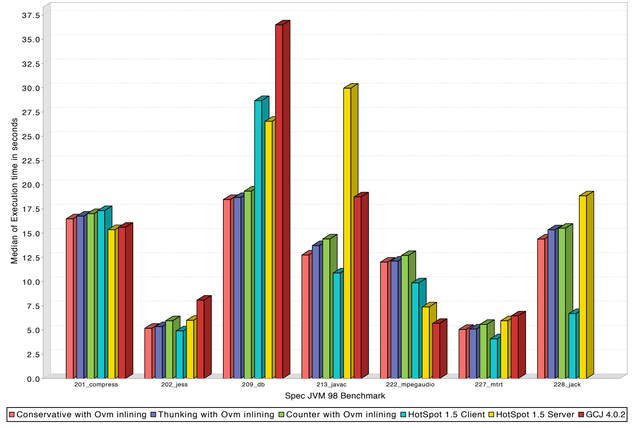
Performance of the four mechanisms for achieving accurate stack scanning, measured as overhead relative to the conservative collector for a small heap size (25 MB). Note that in the geometric mean we see that the overhead of the *Thunking* (safe point catch and thunk) configuration is negative—this is because the overhead cost of accurate stack scanning with thunking is outweighed by the gains from having unambiguous roots in the collector. Hence, for small heaps and our collector, better performance tends to be achieved with safe point catch and thunk than with using a conservative collector.

Median of Total GC time (us) - 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)



Overhead in time spent in the collector for the four accurate stack scanning configurations. Note that scanning the stack using any of these mechanisms tends to be slower than scanning the stack conservatively because of two factors: 1) the collector is written in Java and so has all of the same accurate instrumentation, and 2) conservative stack scanning always involves a trivial walk of the thread stacks and register files, while the other mechanisms are typically more involved. However, we see speed-ups for some benchmarks, likely due to the collector having to pin too many pages in the conservative configuration.

Median of Execution time in seconds - Spec JVM 98 Benchmark vs. VM - Comparing VMs, Heap Size: 256m, 60 runs



Comparison of two Ovm accurate configurations and the conservative configuration to HotSpot and GCJ. Note that Ovm with thunking or conservative stack scanning is competitive with the other systems. This shows that it is possible to have a fast VM that translates Java bytecodes to C++, without loss of accurate stack scanning.

Research by Jason Baker, Antonio Cunei, Filip Pizlo, and Jan Vitek of the S3 Lab at Purdue University.