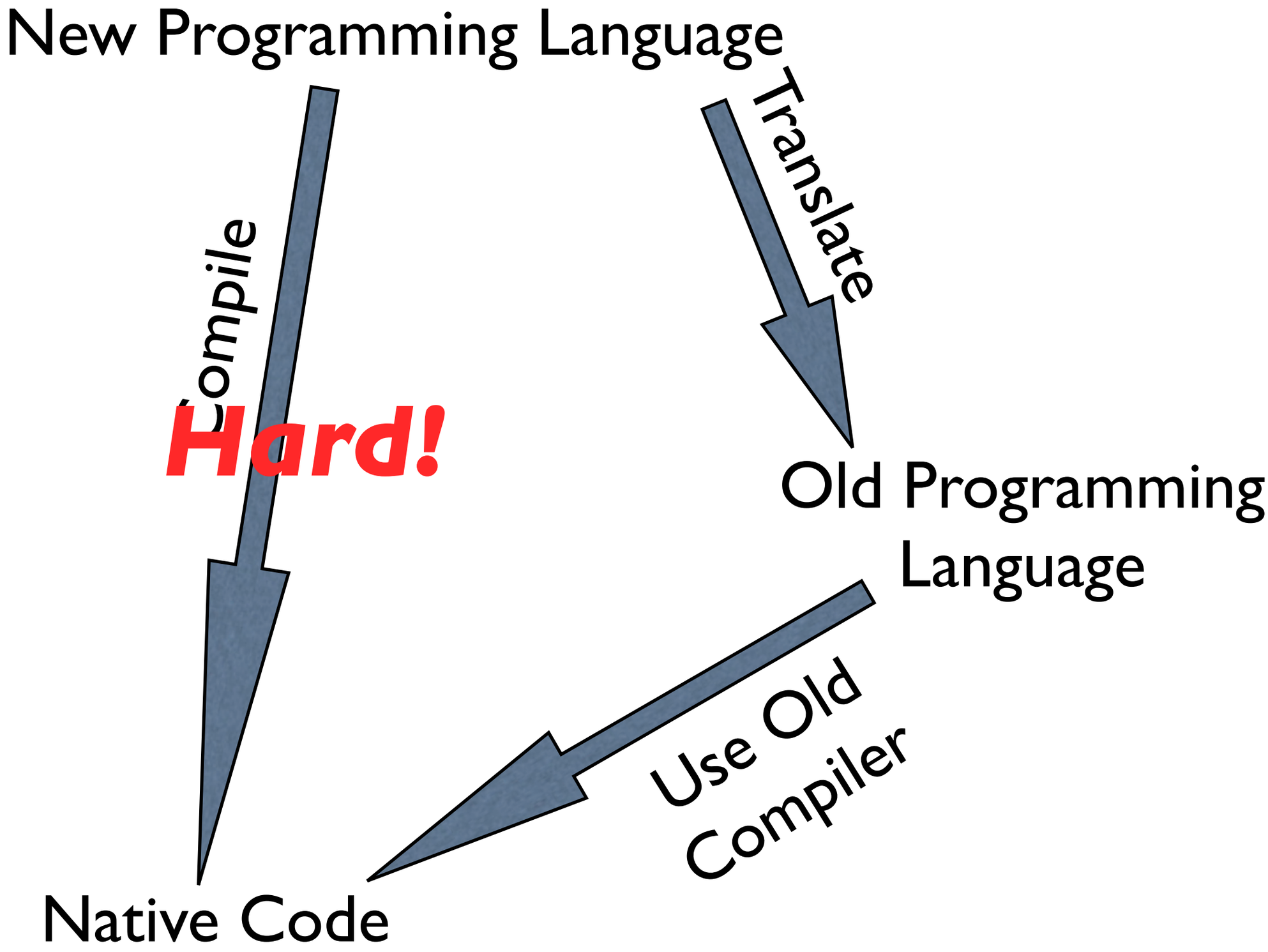
Accurate Garbage Collection *in* Uncooperative Environments *with* Lazy Pointer Stacks

Jason Baker, Antonio Cunei, **Filip Pizlo**, Jan Vitek
Purdue University


New Programming Language
Compile
Hard!
Translate
Old Programming Language
Use Old Compiler
Native Code

Which Old Language?

- Lots of systems translate to C or C++
- Can use freely available high quality compilers like GCC to target many platforms (x86, AMD64, PowerPC, SPARC, ARM, etc.)
- Portability and speed (almost) for free!

So what is the Problem?

- In three words: *accurate garbage collection*.
- Most new languages have some form of garbage collection.
- *Accurate* garbage collection is often preferred as it reclaims more memory and is more predictable than *conservative* garbage collection.
- However, accurate GC requires accurate stack maps.

```
void foo(void) {
    void *ptr = alloc();
    if (ptr == 0) error();
    bar(ptr);
}
```

```
_foo:
        mflr r0
        stmw r30,-8(r1)
        stw r0,8(r1)
        stwu r1,-80(r1)
        bl L_alloc$stub
        mr. r30,r3
        bne+ cr0,L2
        bl L_error$stub
L2:
        mr r3,r30
        bl L_bar$stub
        lwz r0,88(r1)
        addi r1,r1,80
        mtlr r0
        lmw r30,-8(r1)
        blr
```

- *Accurate GC requires accurate stack maps.*

- Most C/C++ compilers cannot provide accurate stack maps.

- ***We would like to scan the stack accurately while still using a stock C++ compiler as our back-end.***

Old Approaches

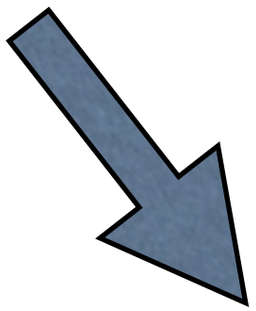
- Pointer Stacks
- Henderson's linked lists

Pointer Stacks

- Idea: put all pointer local variables into an array in the heap.
- Make accesses to these locals go to the array.
- To find pointers, just scan this array.
- We say that the array is an *explicit pointer stack* because it mimics the normal C stack but contains only pointers.

Pointer Stacks

```
void foo() {
    void *ptr = alloc();
    bar(ptr);
}
```



```
extern void **pStackTop;
void foo() {
    pStackTop++;
    pStackTop[-1] = alloc();
    bar(pStackTop[-1]);
    pStackTop--;
}
```

Henderson's Linked Lists

- See Henderson ISMM'02, or our paper, for details.

- Same basic idea as pointer stacks, but uses a linked list instead of an array.

Analysis of these approaches

- Both approaches are legal C (or C++) and so are portable - they will have the desired effect on any standards-compliant compiler.
- Both approaches make stack scanning very easy.
- *Neither approach allows register allocation of pointer locals.*
- *Both approaches add code to the prologue and epilogue*

Can we do better?

- The goal is to allow local pointers to be register allocated.
- Further, we wish to minimize the amount of additional code in the prologue and epilogue.
- *Is this possible?*

The Idea

- Keep pointers in local variables
- Allow C++ compiler to place pointers *anywhere*.
- Have a mechanism for moving the pointers from the C++ local variables to a well-known heap location *on-demand*.

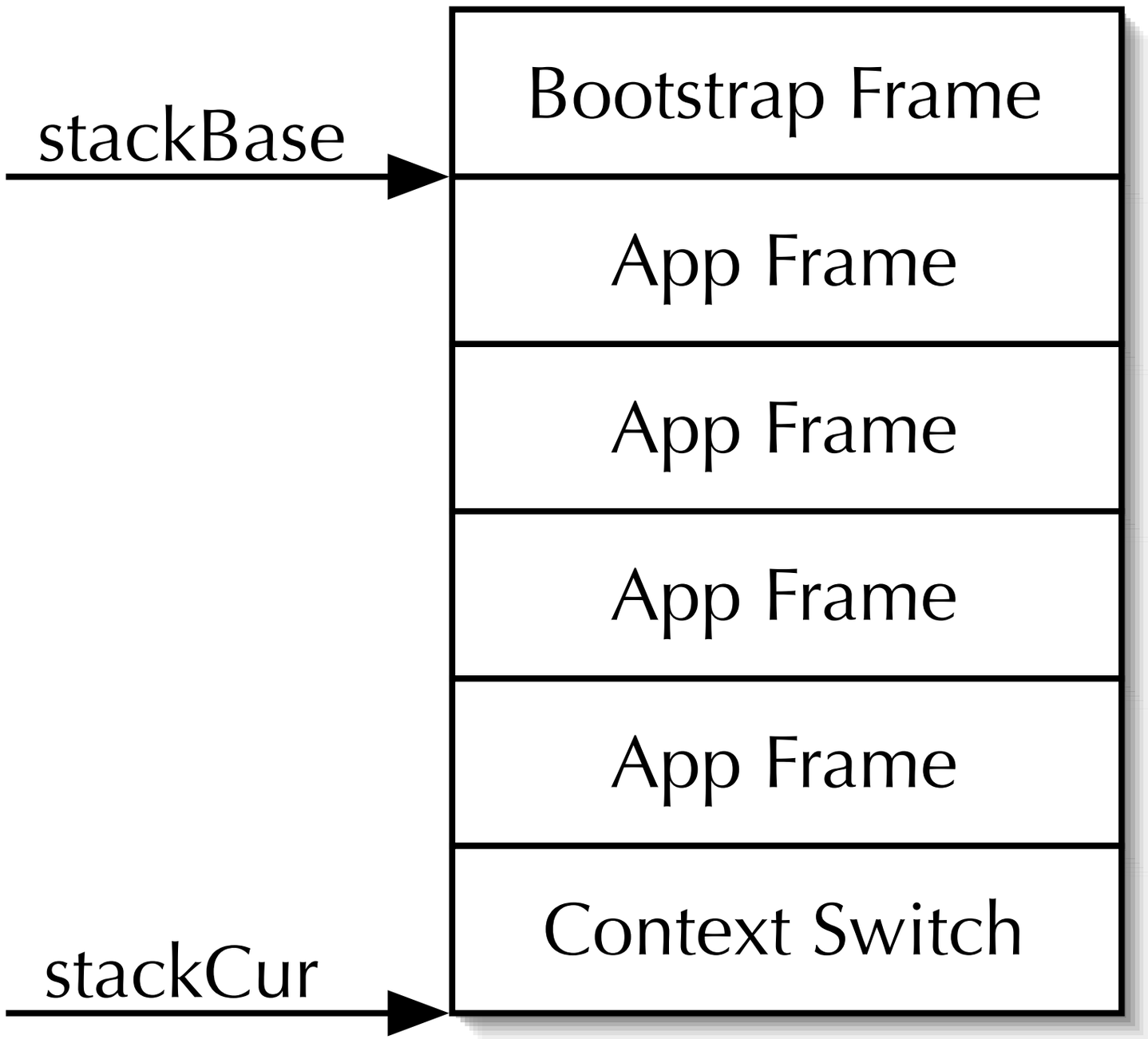
- When the collector wishes to scan the stack, it causes every thread to throw an exception.
- Transform each safe point to catch the exception *and save pointers to a pointer stack.*
- After pointers are saved, the exception is rethrown.
- When this process completes, two things will have happened:
  - *First, the collector will have accurate pointer information, and*
  - *second, all thread stacks will be destroyed!*

```
void foo() {
    void *ptr = alloc();
    try {
        bar(ptr);
    } catch (const StackScanException&) {
        lazyPtrStack->pushPtr(ptr);
        throw;
    }
}
```

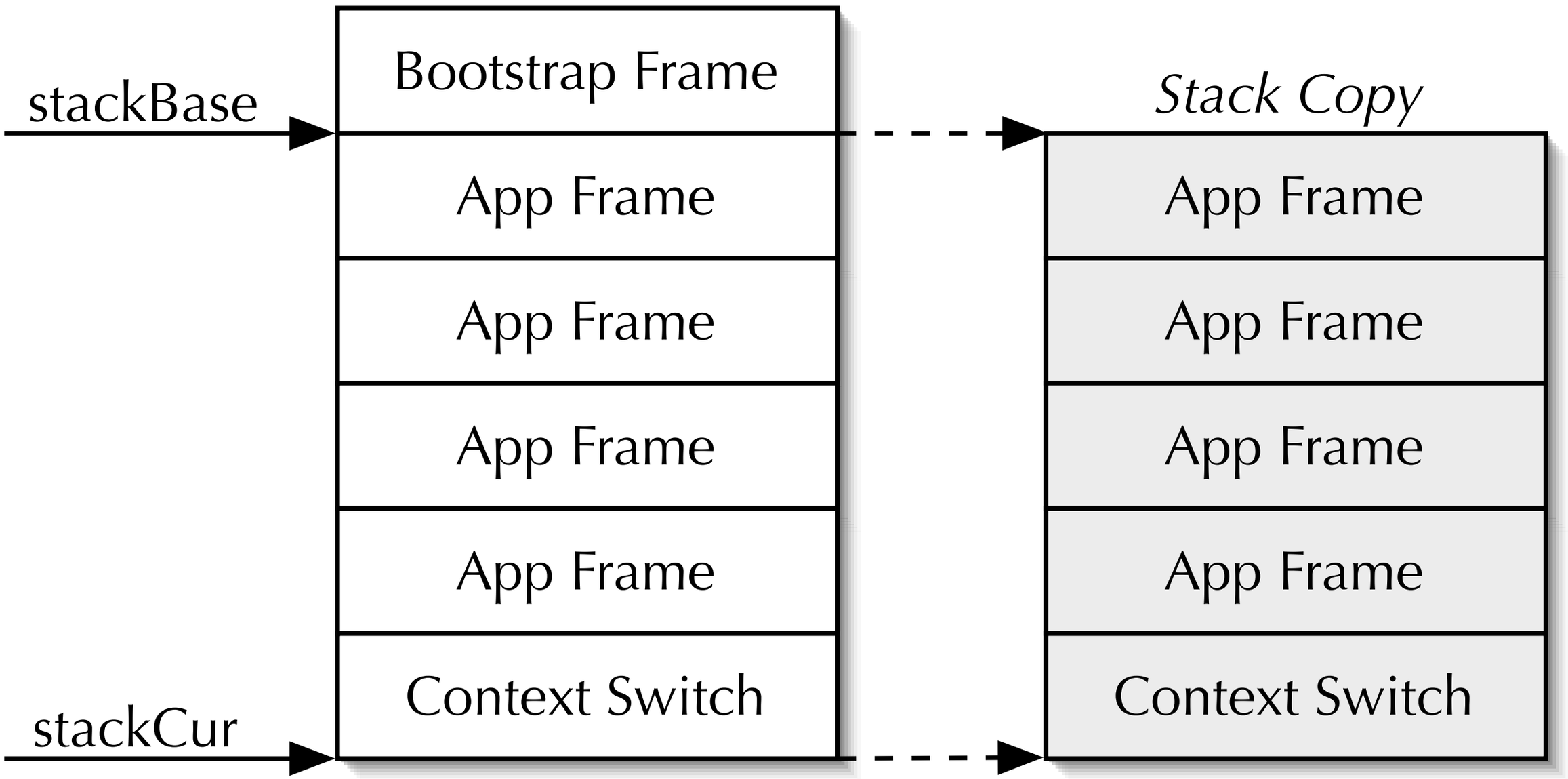
Two problems remain!

- First, we must find a way to restore the stacks to their previous state so that the program can execute, and
- second, we still need a way of allowing the collector to restore the pointers to *new values* (to support *moving collection*).

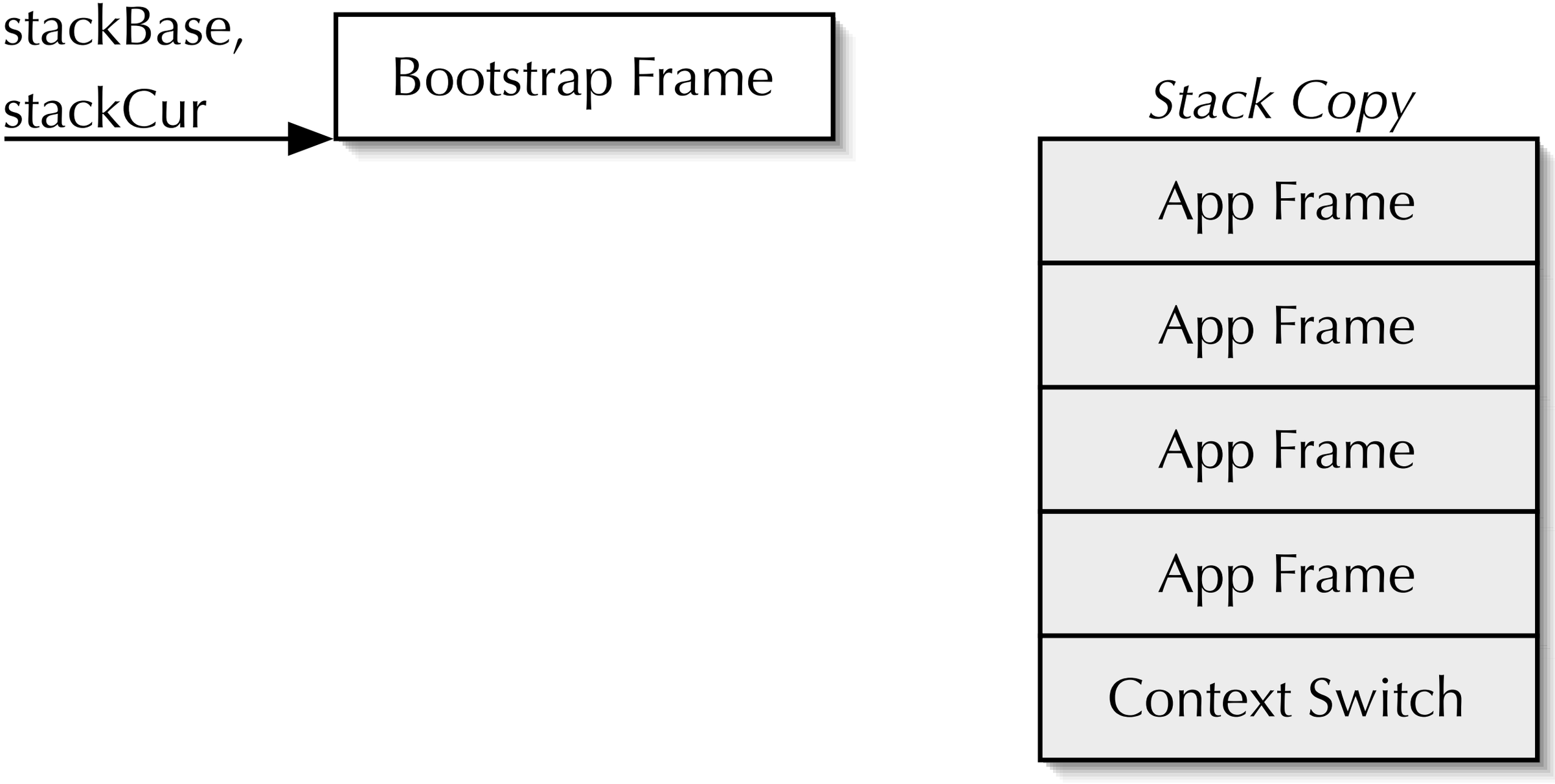
The solution to the first problem...



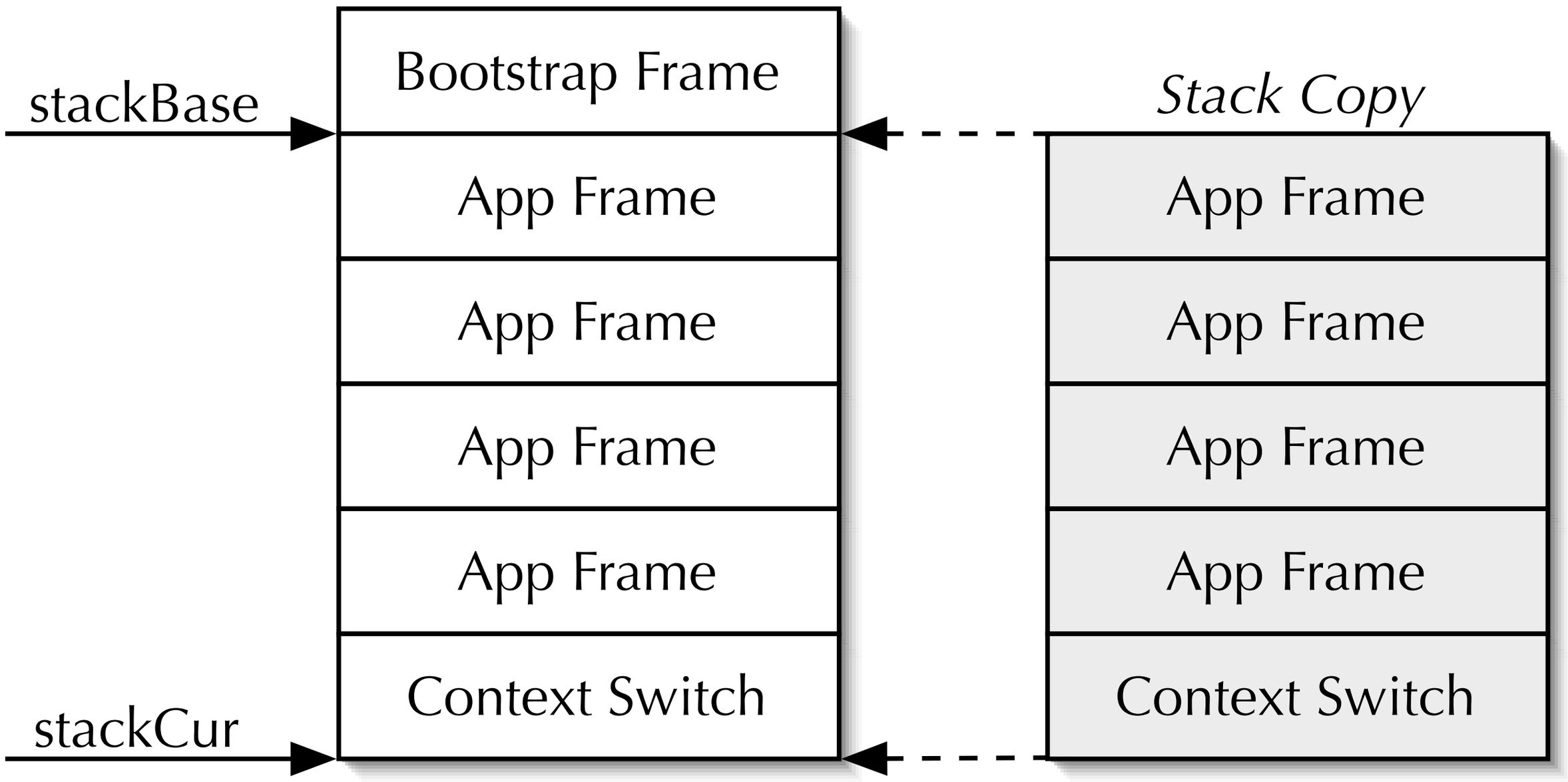
(a) Switch to a thread that needs stack walking.



(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).

Problem 2: Moving GC

What about pointer restoration?

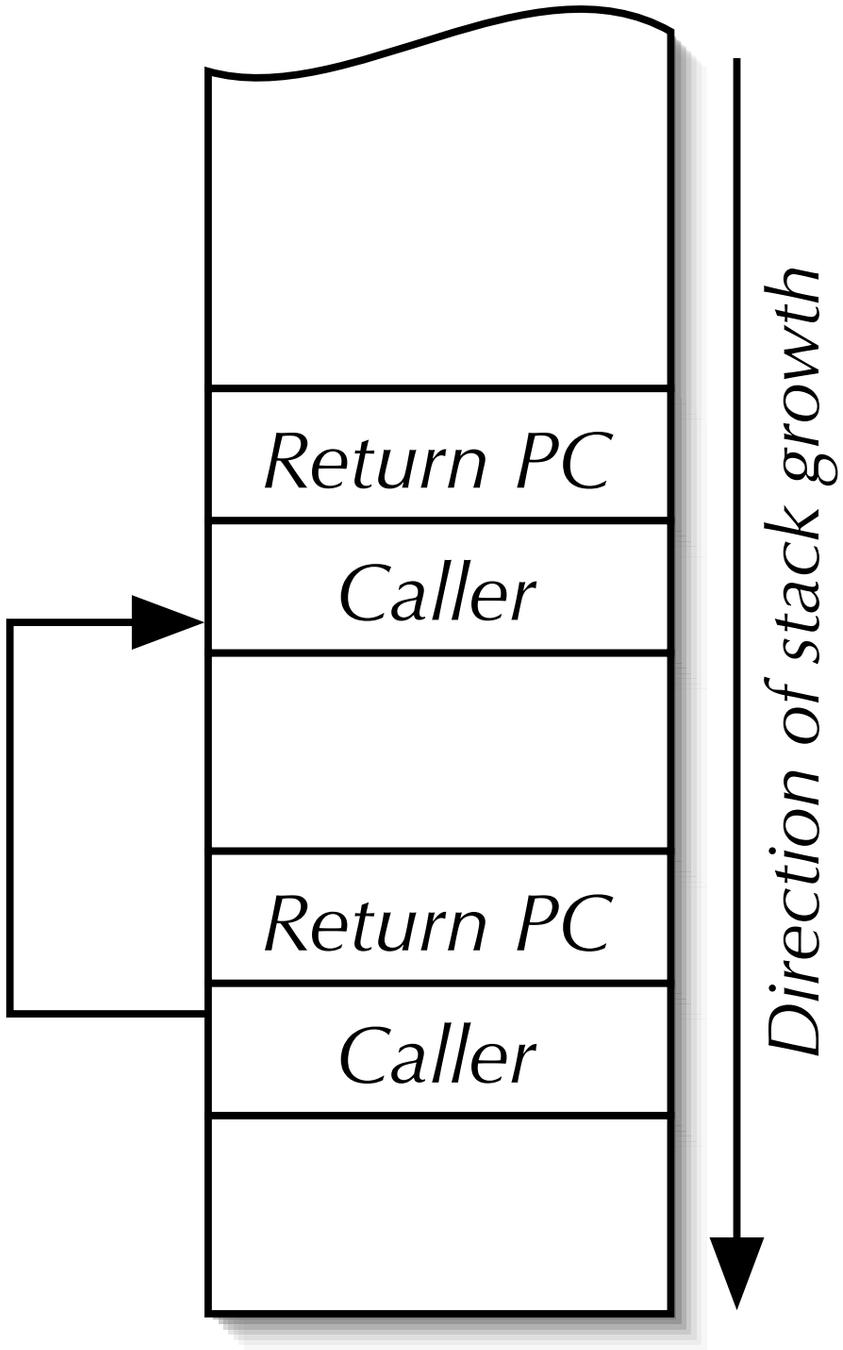
- We cannot directly modify the stack to update the pointers - because we still have no idea where the C++ compiler has placed pointers!
- All we can do is generate C++ code that performs pointer replacement *in the context of the affected frame*.
- Thus, we wish for some code to run at the safe point, but this time:
  - We want to run the code when the called function actually returns following GC,
  - and we want to take this opportunity to restore pointers.

- Assume for a moment that we can magically throw an exception when we return for the first time into a frame after GC.
- Then we can use the same strategy as before: a catch block that runs restoration code.

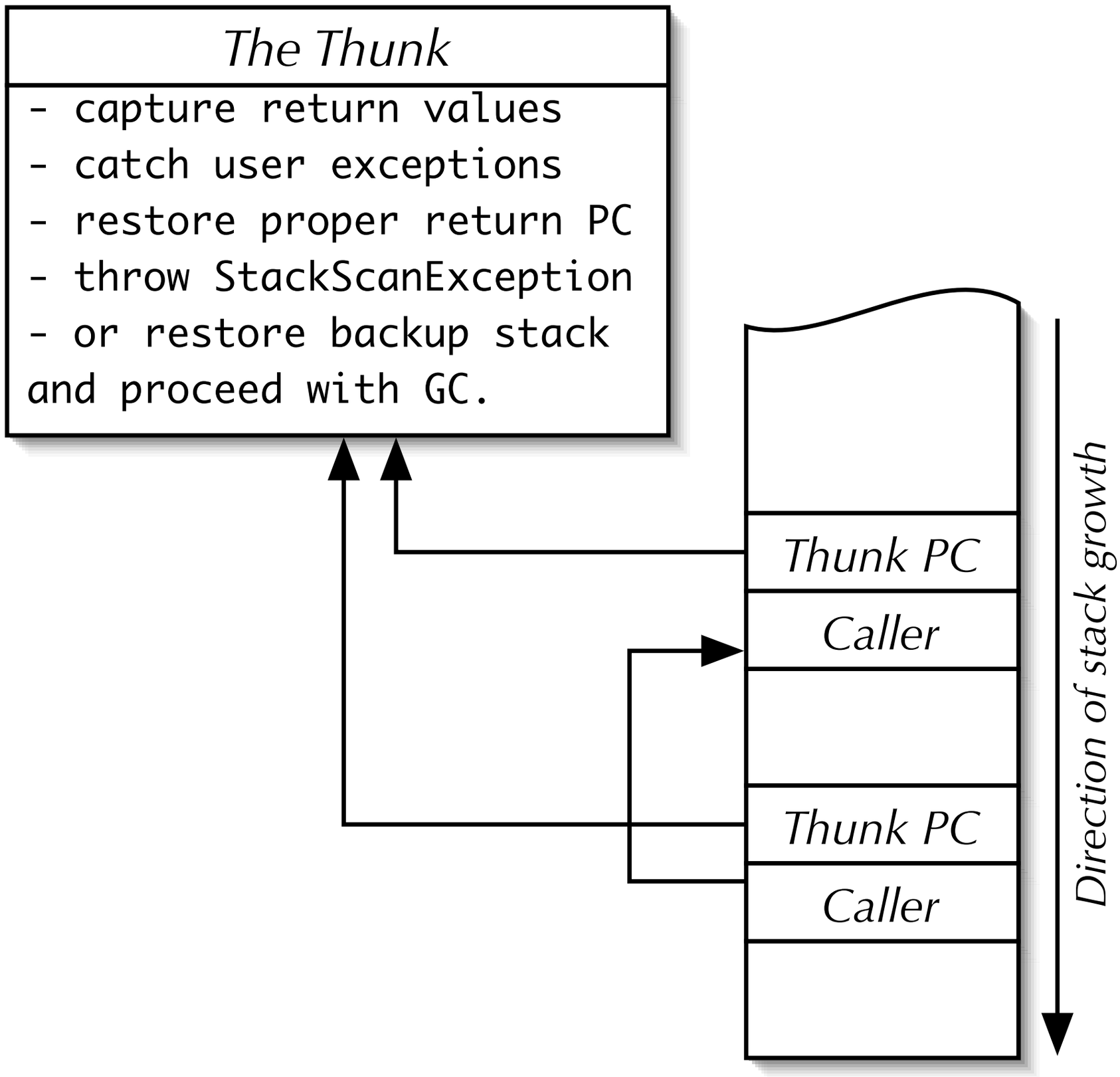
And the code looks like...

```
void *ptr;
try {
    functionCall();
} catch (const StackScanException&) {
    if (saving) {
        lazyPtrStack->pushPtr(ptr);
        throw;
    } else if (restoring) {
        ptr = lazyPtrStack->popPtr();
        if (returned normally) {
            restore return value;
        } else {
            throw app exception;
        }
    }
}
```

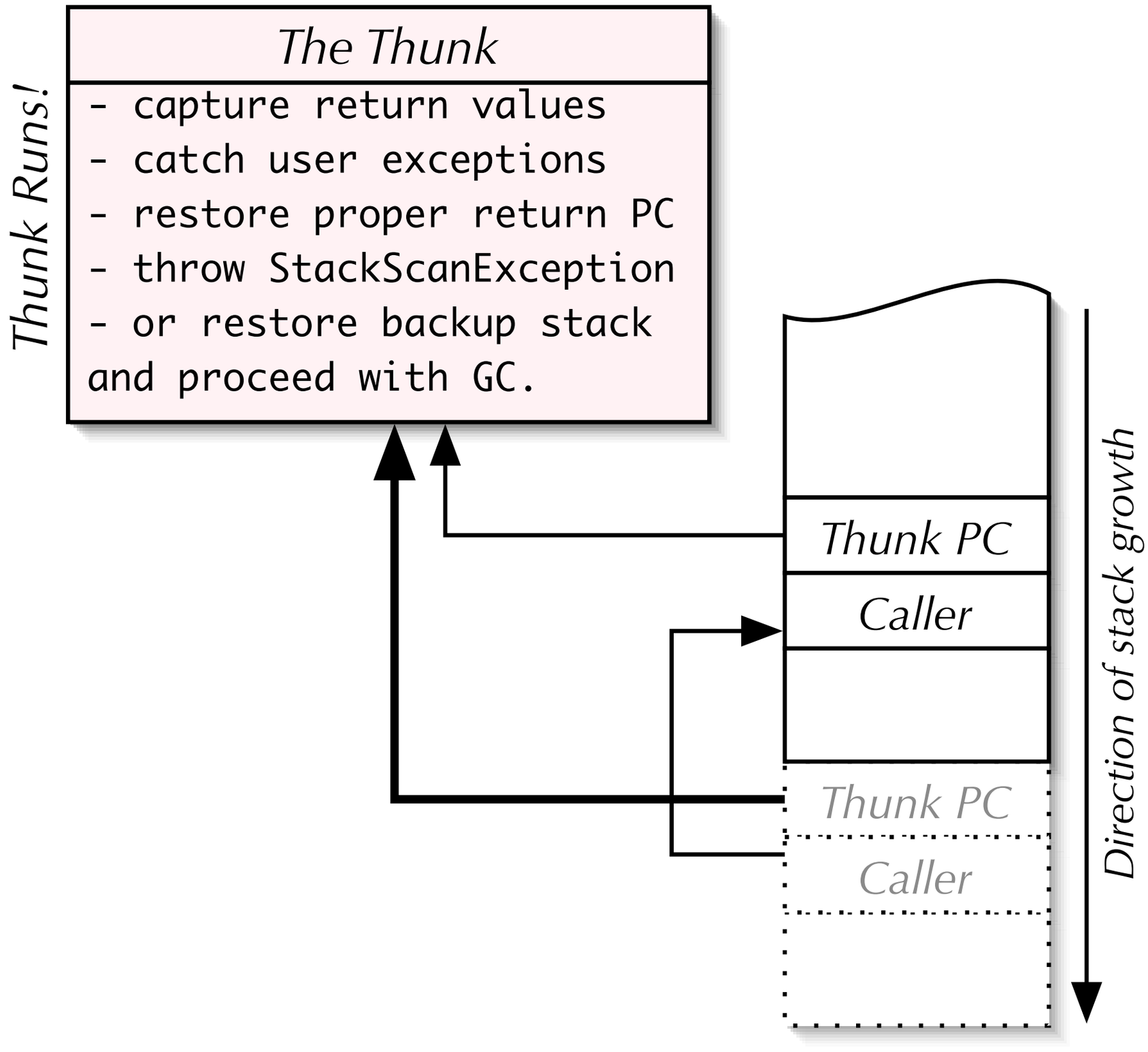
- *How to run the pointer restoration code at the right time?*
- When the GC runs, it updates pointers in its own pointer stack, and then installs *thunks* at every frame on the stack.
- The thunk throws the StackScanException when invoked.



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



(b) "Thunkified" callstack.



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

“Safe Point Catch And Thunk”

1. Throw an exception to trigger stack scanning.
2. Keep a backup copy of the original stack to allow the thread to continue as normal after stack scanning.
3. Install thunks that trigger pointer restoration after the GC runs.

- We have also experimented with using a *counting scheme* to emulate the exception and thunk scheme.

- Put simply, each callsite contains instrumentation that *dynamically checks* if pointers should be saved or restored, by using counters that keep track of stack height.

- Collectively, we call this class of mechanisms "lazy pointer stacks."

Implementation

- We have implemented explicit pointer stacks, Henderson's linked lists, safe point catch and thunk, and pointer frame counting in the Ovm and J2c compiler.
- Ovm is a real time Java virtual machine developed at Purdue.
- J2c is Ovm's ahead-of-time compiler. It generates C++ code, and GCC is used as the backend.

- By default, Ovm+J2c uses mostlyCopying, a Bartlett-style semispace garbage collector that performs conservative stack scanning.

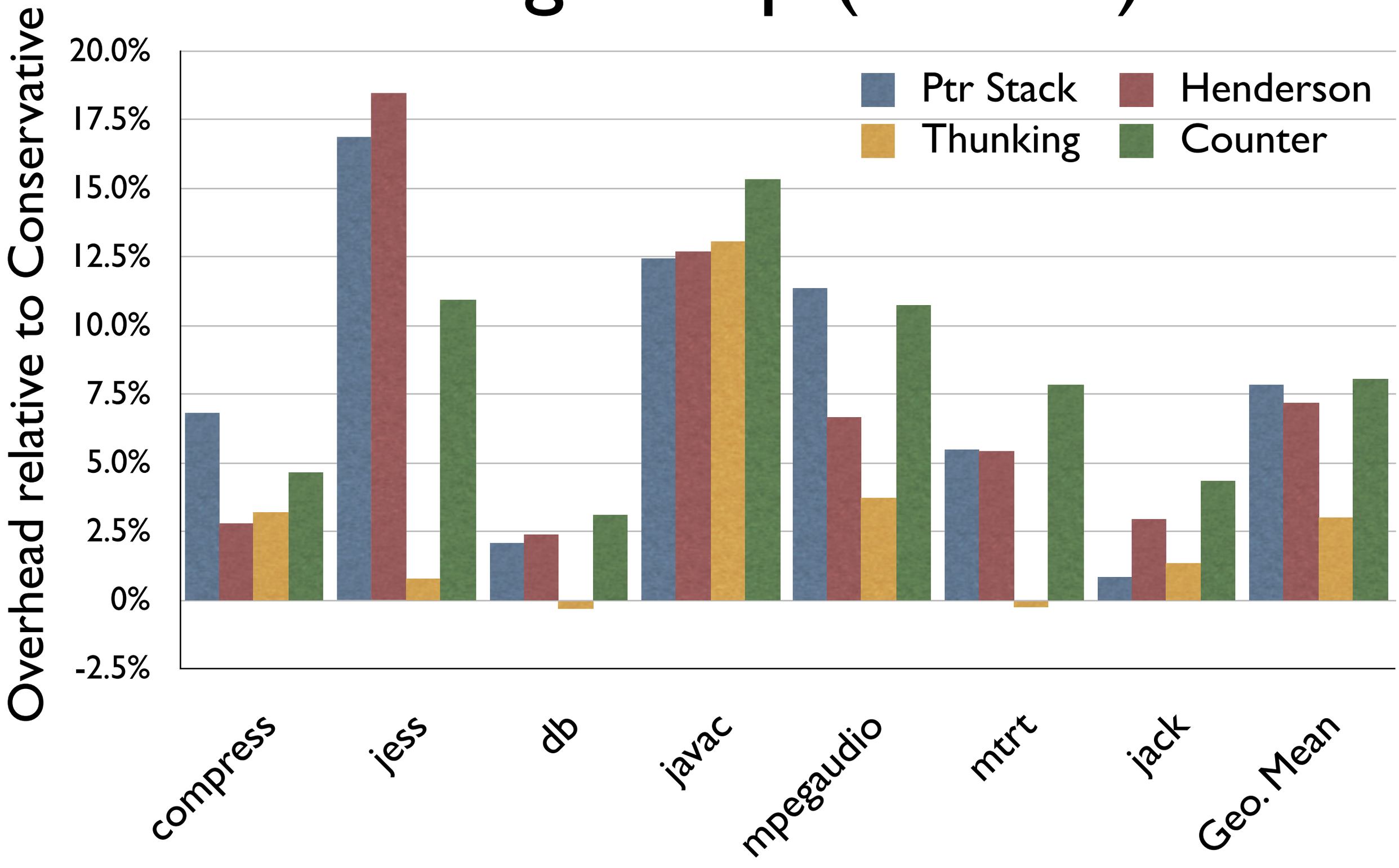
- We have added the ability to perform accurate stack scanning using the four techniques. The user is allowed to select the stack scanning style at compile time.

- The mechanism is modular - any of Ovm's collectors, including our RTGC, can select any of the stack scanning implementations.

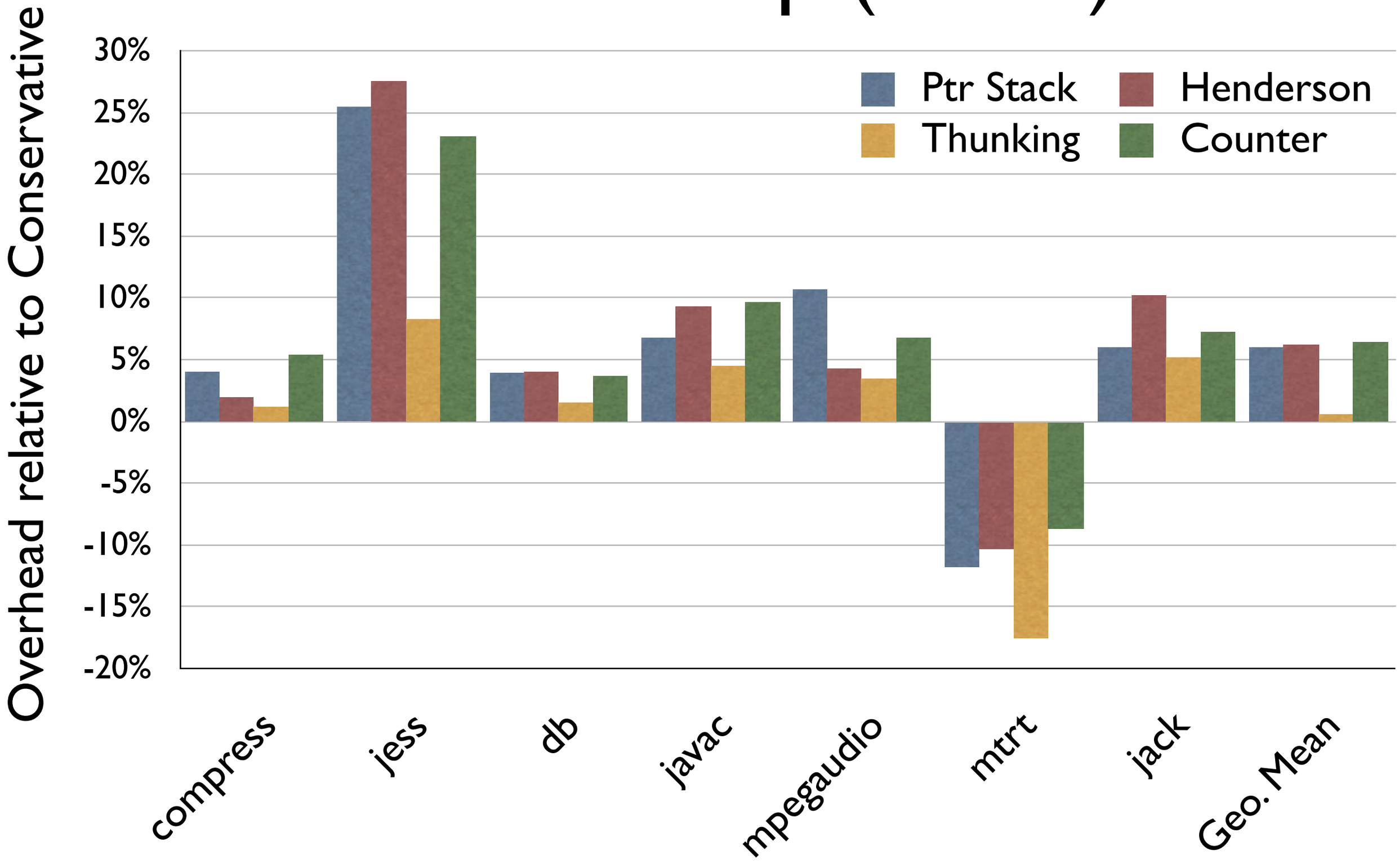
Experimental Evaluation

- We use the industry-standard SPECjvm98 benchmark suite.
- Each benchmark was run with the five stack scanning configurations (conservative, ptr stack, henderson, thunking, and counter) under Ovm+J2c+mostlyCopying at various heap sizes.
- We used a Pentium IV Linux machine with 512 MB of RAM for all runs.
- Additionally, we compared against the HotSpot JVM or GCJ (see paper).

Overhead relative to Conservative for Large heap (256MB)



Overhead relative to Conservative for Small heap (32MB)



Code Size for SPECjvm98 in KB

Conservative	3376
Ptr Stack	3857
Henderson	4031
Thunking	11081
Counter	9320

See the paper for more algorithmic details and more performance evaluation (different heap sizes, some profiling, etc.)

The End