Explicit Stack Management in C--

Russ Cox
rsc@eecs.harvard.edu

ABSTRACT

The portable assembly language C-- currently assumes that program execution takes place on a one-way infinite stack. We propose modifications to the language to allow the front end to implement the stack management policy of its choice. The modifications are complicated by C--'s tail call.

We do not address the implementation of C--'s `yield` call, except to note that it cannot use the program's stack pointer.

Introduction

The portable assembly language C-- [1, 2] assumes that programs run on a one-way infinite stack. This assumption implicitly affects the standard C-- calling convention and stack frame management; these effects in turn make the implementation of other stack management policies (e.g., the segmented copy-on-write stacks used to implement first-class continuations in Chez Scheme [3, 4]) difficult if not impossible.

To remedy this situation, we propose that C-- rely on the front end to provide a function prologue and epilogue to handle stack frame allocation and deallocation. We also propose that C-- use a front-end-provided ''argument frame'' allocator to create space in which to store overflow parameters for function calls. After providing justification for these interfaces, we propose additions to the C-- runtime to allow programs management of the stacks from the front end run-time systems.

Goals

We would like C-- to enable the following stack management methods.

- *One-way infinite stack*. The current stack protocol should still be possible, and should be no more expensive than it currently is.
- *''No stack''*. A compiler such as the one used by Standard ML of New Jersey [5, 6] should be able to use heap-allocated objects for function arguments and activation records.
- *Segmented stacks*. The implementation of first-class continuations in Chez Scheme uses a linked list of stack segments to implement continuations efficiently.

We want to allow the front end to be able to implement a policy such as these but still be isolated from as many details (hopefully all!) as possible about the target architecture.

Background

Ramsey and Peyton Jones identify three different parameters that describe the space in use by an instance of a function. Specifically, the space consists of an *n*-byte activation record and an *i*-byte incoming overflow parameter area; if the function is preparing to do a call or a jump, there will be an *o*-byte outgoing overflow parameter

area as well. Ramsey and Peyton Jones, working on a one-way-infinite stack, note that considering parameters to be part of the caller's or callee's activation record is mostly a question of point of view. Unfortunately, this question becomes more precise and more hairy when we break this model; in fact, the overflow parameters will have to be treated differently than the activation records.

Front end provisions

To make the stack management explicit, we propose that the C-- back end inline ''calls'' to helper routines provided by the front end to handle activation record and overflow argument management in the generated code. We assume three pseudoregisters[1]: `fp` points at the activation frame, `ip` points at the incoming overflow parameter frame, and `op` points at the outgoing overflow parameter frame. The helper routines are used as follows.

- `fn_prologue(`*n*`, `*i*`, ...)`: allocate an activation record of size *n*, storing a pointer to its base in `fp`; also set `ip` to point at the *i* bytes of incoming overflow arguments, if any, using the helper parameters returned by `fn_findargs`. The old value of `fp` must be recoverable.
- `fn_epilogue(`*n*`, `*i*`, ...)`: in preparation to return, deallocate the activation record to which `fp` points, which is of size *n*; additionally, deallocate the argument frame to which `ip` points, which is of size *i* and was found using the given helper parameters. The old value of `fp` (i.e., its value when `fn_prologue` was called) must be restored.
- `fn_jumpepilogue(`*n*`, `*i*`, `*o*`, ...)`: Like `fn_epilogue`, but in preparation for a tail call.
- `fn_allocargs(`*o*`)`: allocate an *o*-byte overflow argument frame, storing a pointer to its base in `op`; return some number of values (possible zero) that can be used in a call to `fn_prologue` to locate the argument frame.

For the mechanism to work efficiently, the C-- compiler must inline the calls and perform optimizations such as constant propagation and unreachable and dead code elimination. The number of values returned by `fn_allocargs` must be the number of extra parameters taken by `fn_prologue` and `fn_epilogue`; this restriction is easily checked by the compiler. (The number of values used to locate the overflow frame influences the number of overflow parameters, so the compiler must find this anyway).

As an example, consider the following tail recursive C-- function to compute elements of the Fibonacci sequence. We assume we must spill all function parameters but that the return value can be kept in a register.

[1] The pseudoregisters could go in favor of more values passed and returned to the ''calls'', assuming the compiler implements a sufficiently aggressive register coalescing algorithm. The C-- compiler is free to treat these pseudoregisters as variables that can be dead at given points in the function; for example a function with no incoming overflow parameters would not need that register. Further, if the compiler deduces that two registers have a constant difference and one is only used as a base register for loads and stores, it seems reasonable to expect that the compiler could rewrite those loads and stores to use the other register.

```
fib(n) {
        m = fibhelp(1, 1, n);
        return m;
}

fibhelp(f0, f1, n) {
        if(n == 0)
                return f0;
        jump fibhelp(f1, f0+f1, n-1);
}
```

A C-- compiler might translate this into the following pseudo-assembly (before inlining) for a 386.

```
TEXT fib
        /* find argument, load into register */
        fn_prologue(0, 1*4)
        MOV 0(ip) -> AX

        /* prepare arguments and call */
        fn_allocargs(3*4)
        MOV $1, 0(op)
        MOV $1, 4(op)
        MOV AX, 8(op)
        CALL fibhelp

        fn_epilogue(0, 1*4)
        RET
TEXT fibhelp
        /* find arguments, load into registers */
        fn_prologue(0, 3*4)
        MOV 0(ip) -> AX
        MOV 4(ip) -> BX
        MOV 8(ip) -> CX

        DEC CX
        JGE tailcall    /* jump if CX >= 0 */
        RET

tailcall:
        ADD BX -> AX

        /* prepare arguments and jump */
        fn_allocargs(3*4)
        MOV BX -> 0(op)
        MOV AX -> 4(op)
        MOV CX -> 8(op)

        fn_jumpepilogue(0, 3*4, 3*4)
        JMP fibhelp
```

Because the front end provides the inlined functions, it retains control over allocation policies and the like.

Justification

Before continuing, it likely seems startling that function call arguments are allocated and managed separately from activation records. This is dictated by C--'s unrestricted tail calls.

An activation record is created and destroyed by the function that uses it. This is

necessary mainly for independent compilation of modules: it is unreasonable to expect every caller to know how large a stack frame to allocate or destroy for each function it calls. Function arguments are fundamentally different, as they anticipate the callee's stack frame but can outlive the caller's stack frame (as is the case in a tail call). Thus we must allocate and reclaim them separately from stack frames. Techniques developed to handle tail calls in the case of a one-way infinite stack can still be expressed efficiently via this model, as will be shown.

Example: One-way infinite stacks

We must ensure that the implementation of a simple one-way infinite stack does not become inefficient. The simplest way to do this is to keep the frame pointer pointing at the bottom of the stack frame, and write outgoing arguments under it. It will be up to the callee to move the frame pointer down to acomodate the arguments and any desired extra stack space, and to restore it upon return.

The prologue, epilogue, and argument allocator are as expected:

```
fn_prologue(bits32 n, bits32 i) {
        ip = fp-i;
        fp -= n+i;
}
fn_epilogue(bits32 n, bits32 i) {
        fp += n+i;
}
fn_allocargs(bits32 o) {
        op = fp-o;
}
```

For a jump, we will copy the arguments up over our own stack frame.

```
fn_jumpepilogue(bits32 n, bits32 i, bits32 o) {
        move(fp+n+i, fp, o);
        fp += n+i+o;
}
```

The only arguably inefficient part of the resulting scheme over a hand-coded one is the block copy for the outgoing arguments at a jump. In the general case, though, the stack frame might contain enough vital information, and the number of outgoing arguments might be so large, that the copy would be necessary anyway. The details of `move` could be such that the small cases were picked off, if desired. While algorithms do exist to handle such argument transformations, they would need to be implemented by the front end, which would require intimate knowledge of the stack layout; the issue does not seem pressing enough to consider bringing the front and back ends so close together.

On a RISC machine, this is the whole story, but on a CISC machine like the 386, we need to worry about the return address, which is kept on the stack, and could potentially be wiped out by the epilogues. Since the return address for a call never outlives the caller, the C-- back end can arrange for the return address to go in the stack frame rather than be treated a true argument. If we map `fp` to the stack pointer, then we can reserve the bottommost word of the stack frame for the return address. Then a call to `foo()` translates to

```
ADD $4, SP
CALL foo
SUB $4, SP
```

while a jump remains a `JMP` instruction. In the case where no calls are made by the function, the extra word need not be allocated.[2] Alternate schemes might treat the

[2] We might consider using the return address as an argument and simply jumping to it rather than

return address as the first or last argument, but that would involve the epilogues in machine-dependent operations and thus hinder their portability. The benefit of this scheme is that the front end need not worry at all about the fact that it is compiling for a 386.

Example: Heap allocated activation records

We note that a compiler employing heap allocated activation records has an easy time of providing this interface. We assume that a copying collector is in use, that "hp" points and the heap frontier, and that "hlimit" points at the end of the heap. We punt the implementation of garbage collection to the next half of this paper, and ignore synchronization completely.

The prologue and argument allocator are both inlined calls to the heap allocator; the prologue need not save the current frame pointer, since we will not return.

```
fn_prologue(bits32 n, bits32 i, bits32 passed_ip) {
        if(n > 0) {
                fp = hp;
                hp += n;
                if(hp > hlimit)
                        yield();
        }
        ip = passed_ip;
}
fn_allocargs(bits32 o) {
        op = hp;
        hp += o;
        if(hp > hlimit)
                yield();
        return op;
}
```

The two epilogues are empty functions, leaving the garbage collector to pick up the stack and argument frames. We also assume that garbage collection will fix `fp` and `op` when `yield` is called. Note the dependence upon constant propagation and code elimination to trivialize `fn_prologue` in the case where there is no stack frame to be allocated, or no arguments to be located (in which case `ip` will not be used in the rest of the function).

The prologue and argument allocation could be folded into one, to avoid the extra instance of `yield`; however, it seems to me a case of premature optimization, since the usual case in which such allocation patterns arise is in a continuation-passing compiler, which will make exactly one call per function, so at best we save a factor of two (assuming no computation whatsoever is being performed).

Code Restrictions

Clearly arbitrary code cannot be placed in the prologue and epilogue functions, since there is no activation record in which to spill temporaries and the like. It is hard, if not impossible, to give a useful definition of what might be expected to be a valid sequence of code to inline. Precisely, a sequence is valid if and only if the back end compiler is able to compile it using no spilled temporaries. This is horribly target and compiler dependent. To the front end writer we can only give the advice to keep the

---

using the native return instruction. In fact, failure to use the native call and return instructions results in enormous performance hits on the Pentium, due to conflict with the instruction prefetch and branch prediction units. For a doubly-recursive Fibonacci computation, the cost of using POP AX; JMP AX rather than RET is an approximately 60% increase in wall clock time. To put things in perspective, POP AX; PUSH AX; RET results in only a 10% increase.

prologues simple, perhaps envisioning a target machine with four to six registers in which to work.

Back end provisions

Since stack and argument frames are now heap allocated, they are subject to garbage collection. We have deferred that issue up to this point because it requires extra functions to be provided by the back end, specifically by the C-- run-time system.

The activation walking mechanisms need the front end to provide a function that, given a frame pointer, returns the saved frame pointer in that frame, or zero if this is the last frame:

```
void *GetNextFP(activation *a, void *fp);
```

This is necessary because only the front end knows where the prologue squirrels away the old frame pointer.

Further, the front end needs the back end to provide information[3] about the frame sizes (*i*, *n*, and *o* in the previous discussion) for a given activation.

```
int GetFrameSize(activation *a);
int GetInputSize(activation *a);
int GetOutputSize(activation *a);
```

When the suspended function has been stopped between `fn_allocargs` and the corresponding function call, `GetOutputSize` returns the size of the overflow area; otherwise it returns zero. Finally, the function `FindReg` returns a pointer to the value of a pseudoregister like `fp`, `ip`, or `op`.

```
void *FindReg(activation *a, int reg);
```

Example: Heap allocated activation records, continued

It seems a safe assumption that pointers into the stack frames do not exist; certainly in a continuation-passing compiler, the only use of stack variables should be to spill temporaries or local variables which would otherwise be in registers. Given this assumption, the copying of the C-- stack frame is simple.

[3] It is not reasonable to put these in spans because only the back end knows their value at various points in the code; in fact, the span ranges would not even correspond to line boundaries in the C-- source.

```
void
CopyStack(void)
{
        int i, n, o;
        void **p;
        activation *a;

        a = FirstActivation();
        if(n = GetFrameSize(a)) {
                p = FindReg(a, FP);
                *p = copy(*p, n);
        }
        if(i = GetInputSize(a)) {
                p = FindReg(a, IP);
                *p = copy(*p, i);
        }
        if(o = GetOutputSize(a)) {
                p = FindReg(a, OP);
                *p = copy(*p, o);
        }
}
```

Since there is no next activation, we need not look for it. Note that the C-- run-time interface must use the locations returned by `FindReg` when it needs to look at the values therein, so that changes are accurately reflected.

Example: Segmented stacks

As a final example, we consider the implementation of segmented stacks *à la* Chez Scheme. The run-time system keeps a linked list of segment descriptors, each of which contains a pointer to a stack segment and the size of the segment. The top frame (assuming downward stack growth) has as its return address a special stack underflow handling routine rather than a real return address. The original return address for that frame is also kept in the segment descriptor. We need a mechanism both for rewriting this return address and jumping to the old one. *A priori*, it is not clear that the ability to make such modifications would not run afoul of the optimizer. For instance, the ability to arbitrarily set the program counter to which a thread resumes would certainly invalidate the large majority of optimizations performed by the compiler. However, it does seem reasonable to be able to find and write to the return address in an activation; since pointers to functions, especially exported functions, enable their calling from arbitrary locations, the optimizer cannot depend on properties of the return address. So let us posit that `FindReg` can treat the return address as a special register. Since we are not overly concerned with the efficiency of handling stack underflow, the following works portably, if not elegantly. As in Chez Scheme, we have a handler function called `underflow`:

```
export underflow;
underflow() {
        underflowyield();
}
export underflowyield;
underflowyield() {
        yield();
}
```

`Underflowyield` is exported so that C-- does not consider inlining it. Now when the stack underflows, we will ''return'' to `underflow`, which will call `underflowyield`, which will yield to the run-time system. Then the run-time system can modify the environment as necessary, changing the return address of

`underflowyield` to be the intended return address, clearing the underflowed stack in the process. Of course, these hoops are necessary only for machines that keep the return address on the stack; RISC machines will work with slightly simpler methods, but this scheme will work as well.

Conclusions

We believe that the use of inlined code fragments and the extra runtime calls outlined here enable a variety of stack-management policies, leaving the ultimate decision in the hands of the front end. Further, they do so in a way that is almost completely machine independent. One of the few machine details directly exposed is stack direction; a magic direction multiplier would be sufficient to remove this dependency. The problem of what code is allowed to appear in a prologue or epilogue is trickier (and, we believe, insoluble).

The current scheme does not allow the allocation of all overflow arguments along with the activation record; we might envision passing a *maxo* parameter to the `fn_prologue` routine; if desired, the prologue could allocate the overflow area and then `fn_allocargs` would simply be a no-op. Such a scheme (reusing argument space for multiple calls) is only viable in the absence of first-class continuations.

The twisting to handle CISC machines like the 386 seems unfortunate but unavoidable; future work might address cleaner ways to deal with such a machine, since it is not likely to go away soon.

References

1. Norman Ramsey and Simon L. Peyton Jones. ''Machine-independent support for garbage collection, debugging, exception handling, and concurrency (draft)''. Technical Report CS-98-19, Department of Computer Science, University of Virginia, August 1998.
2. Norman Ramsey, Simon L. Peyton Jones, and Fermin Reig. ''C--: a portable assembly language that supports garbage collection''. *International Conference on Principles and Practice of Declarative Programming*, LNCS 1702, September 1999.
3. Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. ''Representing Control in the Presence of First-Class Continuations''. *ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, June 1990.
4. Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. ''Representing Control in the Presence of One-Shot Continuations''. *ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, June 1996.
5. Andrew W. Appel and Trevor Jim. ''Continuation-passing, closure-passing style''. *Proc. Sixteenth ACM Symposium on Principles of Programming Languages*, pp. 293-302, January 1989.
6. Andrew W. Appel *Compiling with Continuations*. Cambrige University Press, New York, 1992.