Submitted to the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation

Featherweight concurrency in a portable assembly language

Norman Ramsey
Harvard University
nr@eecs.harvard.edu

Simon Peyton Jones
Microsoft Research Ltd
simonpj@microsoft.com

Abstract

What abstractions should a reusable code generator provide to make it easy for a language implementor to compile a highly concurrent language? The implementation of concurrency is typically tightly interwoven with the code generator and run-time system of the high-level language. Our contribution is to tease out the tricky low-level concurrency mechanisms and to package them in an elegant way, so they can be reused by many front ends.

1 Introduction

C-- is a compiler-target language intended to be independent of both source programming language and target architecture (Peyton Jones, Oliva, and Nordin 1997; Peyton Jones, Ramsey, and Reig 1999; Ramsey and Peyton Jones 2000). It acts as an interface between a *front end* and a reusable code generator. The idea is that the front end translates your favorite language into C--, leaving the C-- compiler to do the rest. C-- encapsulates compilation techniques that are well understood, but difficult to implement. Such techniques include instruction selection, register allocation, instruction scheduling, and optimization of imperative code with loops. The aim of the C-- project is not only to make it easy to reuse code generators, but to enable reusable code generators to provide almost as much flexibility and performance as custom code generators.

There is more to compilation than code generation. Many high-level languages require run-time services such as garbage collection, exception dispatch, and concurrency. These services are typically implemented through close collaboration between the code generator and a *run-time system*, but such close collaboration is impeded by the abstractions needed to make a code generator resuable.

Nowhere is this problem more difficult than in the design of mechanisms to support *concurrent* languages, which is the focus of our paper. Implementing concurrency often involves delicate and architecture-specific interactions between compiled code and the run-time system. For example, the details of changing the program counter and stack pointer during thread switching are precisely the sort of dark magic—well understood, but hard to implement—that we hope to encapsulate behind the abstractions offered by C--.

Here, then, is the problem we address: *we want to make it easy for a front end to support efficient user-level threads, in such a way that all the policy decisions are in the hands of the front end, while all the tricky mechanism is supported by C--.* Like many fine phrases, this goal is easier to state than achieve. The contribution of this paper is a design that extends C-- with clean, architecture-independent mechanisms that can be used to implement common concurrency policies efficiently. We identify three key mechanisms:

Switching contexts. We borrow from functional programming the idea of building concurrency on top of *continuations*. It is well known that exceptions can also be implemented using continuations, and that the full generality of first-class continuations is not needed for either exceptions or concurrency. Our contribution is to identify a single form of continuation that not only supports both exceptions and concurrency, but also is dirt cheap (Section 3).

Sharing data. Some data should be private to each thread, some should be shared among all threads running on a particular processor, and some should be shared among all threads running a particular program. Using C--'s *global registers*, we provide a simple way for the compiler writer to make these choices (Section 5).

Managing stacks. To manage stacks, we have developed a simple model that supports all common styles of stack use: execution on a single infinite stack, finite stacks with detected or undetected failure on overflow, contiguous stacks that are copied and enlarged at need, and segmented stacks (Section 7). Moreover, while our computational model *permits* execution on multiple stacks, it does not *require* multiple stacks, so for example, a garbage collector and mutator can share a single stack (Section 6).

There are other significant mechanisms that we do not have enough space to discuss, but we do mention issues in preemption and synchronization (Section 8).

Our work comes with a large caveat: it is not yet implemented. But it is clearly implementable; we believe that the ideas will interest the PLDI community; and the critical evaluation of that community will help refine our design.

2 What is C--?

C-- is a simple, procedural programming language, with a syntax reminiscent of C. To give a feel for C--, Figure 1 presents three C-- procedures, `sp1`, `sp2`, and `sp3`, each of which computes the sum and product of the integers $1..n$. C-- is designed to be as low-level as possible while still being

```
/* Ordinary recursion */
export sp1;
sp1( bits32 n ) {
  bits32 s, p;
  if n == 1 {
      return( 1, 1 );
  } else {
      s, p = sp1( n-1 );
      return( s+n, p*n );
  }
}

/* Tail recursion */
export sp2;
sp2( bits32 n ) {
  jump sp2_help( n, 1, 1 );
}

sp2_help( bits32 n, bits32 s, bits32 p ) {
  if n==1 {
      return( s, p );
  } else {
      jump sp2_help( n-1, s+n, p*n )
        /* "jump" = tail call */
  }
}
```

```
/* Loops */
export sp3;
sp3( bits32 n ) {
    bits32 s, p;
    s = 1; p = 1;

  loop:
    if n==1 {
      return( s, p );
    } else {
      s = s+n;
      p = p*n;
      n = n-1;
      goto loop;
}   }
```

Figure 1: Three procedures that compute the sum $\sum_{i=1}^{n} i$ and product $\prod_{i=1}^{n} i$, written in C--.

nearly[1] independent of the target architecture. C-- lacks many features that C has, such as user-defined types, because C is a *programming language* while C-- is a *compiler-target language.* C-- has a few features, critical for compilers, that C lacks. For example, C-- has proper tail calls, so procedure `sp2` in Figure 1 works in constant stack space.

So much for basic code generation. Implementations of high-level languages often require *run-time services* such as memory allocation and garbage collection, exception dispatch, and concurrency. It would be wrong for C-- to offer such services; no one semantics, object layout, and cost model could possibly satisfy all clients. Instead, we expect that each client of C-- includes not only a front end, but also a *front-end run-time system*, which implements whatever high-level services are needed. This front-end runtime must cooperate with C--, because C-- controls stack layout and register allocation. The *C-- run-time interface* provides the mechanism for this cooperation.

The run-time interface exports procedures that the front-end run-time system can call to *inspect and modify the state of a suspended C-- computation.* A garbage collector can walk the stack to find roots (Peyton Jones, Ramsey, and Reig 1999); an exception dispatcher can walk the stack to find a handler (Ramsey and Peyton Jones 2000); a scheduler can transfer control between C-- threads running on separate stacks (this paper); and so on. We expose the details of the interface gradually, but the fundamental idea is that *a C-- computation executes on a stack*; the run-time interface provides access to each activation and its local variables. We start by assuming that the stack is contiguous, unbounded, and grows towards increasing addresses; in Section 7 we discuss how to lift these unrealistic assumptions.

[1] C-- deliberately exposes a few architectural details, such as word size and alignment (Peyton Jones, Ramsey, and Reig 1999).

2.1 Implementations and policies for lightweight concurrency

We want to implement concurrent languages, but "concurrency" means different things to different people. To some it means processes or threads supported by the operating system. We do not address this form of concurrency; since the operating system handles all the details of multiplexing the processes or threads onto the available CPUs, this form of concurrency is outside the purview of a code generator.

OS threads and processes are heavyweight mechanisms. Creating and communicating between threads is typically expensive. For example, if Java is implemented by mapping a Java thread to an OS thread, programmers must not create too many threads, must not create them too often, and must not synchronize too often. To economize on thread creation, programmers sometimes write their own schedulers in the high-level language; e.g., "work crews" may pick up and execute work packets (Roberts and Vandevoorde 1989). For languages whose programming model is based on dirt-cheap thread creation, such as Concurrent ML, JoCaml, Concurrent Haskell, Pict, and others, using one OS thread for each language thread is a complete non-starter.

An attractive alternative *user threads*, which means multiplexing many Java (say) threads onto a single OS thread. Now the Java compiler and its run-time system must cooperate to share the computation power of a single OS thread among the Java threads. Everything gets much, much harder for the language implementor. For example, the OS thread provides only one stack, so now the Java compiler must itself allocate stack space for the Java threads, move control between stacks, manage stack overflow, etc. An implementation on a true multiprocessor may be even more complex, e.g., we may run an OS thread on each processor, each of which serves many user threads.

3 Continuations for concurrency

The first key mechanism needed to multiplex threads is the ability to transfer control between threads. In 1980 Mitch Wand noticed that one could support lightweight concurrency in a Scheme implementation simply by using `CATCH`, a precursor of "call with current continuation," also called `call/cc` (Wand 1980). The idea has been widely used in the Scheme community and also taken up by Concurrent ML (Reppy 1991). To take a very simple example, a thread that is ready to yield control might call a procedure `yield` in the scheduler. `yield` would capture the thread's continuation using `call/cc`, choose another thread (= continuation) to run, and throw control to it. The following implementation might be suitable for a uniprocessor:

```
(define yield ()
  (call/cc (lambda (k)      ; capture thread in k
    (put ready-queue k)
    ((get ready-queue)))))  ; throw to other thread
```

Of course one might wish for a more sophisticated scheduling policy, and on a multiprocessor, one would have to synchronize access to the ready queue, etc., but the point is that a *continuation*—a resumable computation—is a splendid encapsulation of a suspended thread; we need no other support for context switching. The win is that the scheduler is just another Scheme function, and the programmer can change the scheduling policy without any help from the language implementor. This is *exactly* the property we want for C--!

Should we simply add first-class continuations to C--? Unfortunately not. C-- is meant to provide an interface to a code generator, and that interface must present a clear cost model to the front end. There are too many strategies for implementing first-class continuations, with too many cost models (Clinger, Hartheimer, and Ost 1999); the client, not C--, should get to choose. Pleasantly enough, however, C-- already supports a weaker form of continuation, which is used to implement exception dispatch in constant time (Ramsey and Peyton Jones 2000). It turns out to be possible to implement Wand's idea using C-- continuations.

3.1 Continuations in C--

In C-- one captures a continuation using a syntactic construct somewhat reminiscent of an exception handler, rather than by using a procedure `call/cc`. Here is an example:

```
f( bits32 x ) {
    bits32 y;

    y = g( x, k ) also cuts to k;
    return( x-y );

    continuation k(y):
        return( x+y );
}

g( bits32 x, bits32 w ) {
  ...
  cut to w(3) ;
  ...
}
```

In this example, the phrase `continuation k(y)` declares the continuation `k`, binding `k` throughout the body of `f`.

- `k` is a value, which can be stored in memory, passed to `g`, and so on. `g` is free to "`cut to`" the continuation, cutting the stack back to the activation containing `k` and resuming execution at that point.
- The continuation can take parameters, to allow information to be conveyed from the `cut to` statement to the target continuation. In the example above, the declaration "`continuation k(y)`" specifies that `y` will be passed to the continuation. This is not a binding occurrence of `y`; on the contrary, `y` must be a local variable of `f`. How the value gets from `cut to` to `continuation` is a private matter for the C-- implementation.
- The phrase "`also cuts to k`" at the call site tells the C-- optimizer that in addition to a normal return, the call to `g` may terminate by transferring control to continuation `k`. The optimizer uses this information to determine liveness and to arrange that the call preserves any values needed in the continuation.
- Though `k` is a value, it is not a completely first-class value, because *its lifetime ends when* `f` *returns, or when any continuation captured by a (transitive) caller of* `f` *is cut to.* It is an *unchecked run-time error* to cut to a dead continuation. The benefit of this Draconian restriction is that such continuations can be implemented very efficiently (Section 3.3). C-- is an unsafe assembly language; it is up to the front end to make sure the C-- continuations are not misused.

Although restricted, these continuations are plenty powerful enough to implement constant-time exception dispatch (Ramsey and Peyton Jones 2000). They are also enough to transfer control between threads, as we see next.

3.2 Making new continuations

We already have nearly enough to implement a simple concurrency package. Suppose a C-- thread voluntarily calls `yield`, a procedure whose intent is to allow another thread to run. The Scheme version of `yield` from the previous page could be written this way in C--:

```
yield() {
  put(ready_queue, k);
  cut to get(ready_queue);
  continuation k:  return;
}
```

Here, `yield` captures a continuation `k`, puts it on the ready queue, decides which thread (continuation) to run next, and uses `cut to` to transfer control to that continuation. More sophisticated scheduling policies can be implemented by writing a different version of `yield`, or perhaps by having `yield` call an out-of-line scheduler.

But where are these continuations on the ready queue to come from? `yield` cannot safely cut to another continuation on its caller's stack, because transferring control to any such continuation would kill `k`. What we need is a C-- primitive to create a fresh continuation:

```
NewContinuation( bits32 stk, bits32 f, bits32 x )
```

`NewContinuation` receives a pointer to a chunk of memory `stk`, in which it creates a new call stack, initializes it, and returns a parameterless continuation, `k`. When this continuation is `cut to`, it runs `f(x)` on the new stack, incidentally killing `k`. It is an error for the procedure `f` to return, because there is nothing sensible to do "after" `f`. Instead, `f` will typically recycle the memory chunk and jump to the scheduler—but C-- neither knows nor cares about that. Section 4 has an example. We do not need to tell `NewContinuation` how big the memory chunk is, because dealing with stack overflow is the client's responsibility, as we discuss in Section 7.

3.3 Implementation notes and cost model

Because of their limited lifetime, C-- continuations are dirt cheap, and they come with a guaranteed cost model. A continuation value is represented by a single machine address, so it is cheap to pass and store. Exactly what the address points to is private to C--, but one good implementation is to represent the continuation by a pointer into the stack. `cut to` simply loads this pointer into the stack-pointer register, puts the actual parameters into standard locations, finds the program counter in some standard location on the stack, and branches to that program counter. Because `cut to` is much like a return, it's attractive to pass the actual parameters using the same convention we use for returning values from a procedure, but other conventions are equally valid. No matter what the value-passing convention, `cut to` takes constant time.

		How many calls	
		Arbitrarily many	At most one
Lifetime	Arbitrary	*Expensive* Wand (1980)	*Moderate* (Bruggeman, Waddell, and Dybvig 1996)
	Stack	*Cheap* (this paper)	

Figure 2: Design choices and costs for continuations

Capturing a continuation is extremely cheap; we need only save a stack pointer and program counter. Capturing a continuation has a subtle indirect cost, however. Consider the example program in Section 3.1. Because `cut to` is guaranteed to take constant time, *it cannot restore any callee-saves registers that are live at the call to* `g`. So the flow edge specified by the `also cuts to` annotation kills the callee-saves registers. That in turn makes the call to `g` slightly more expensive, because any variable live in `k` (for example, `x`), cannot be kept in a callee-saves register.

`NewContinuation` is a constant-time operation, involving a handful of memory writes to initialize the new stack chunk.

Figure 2 compares C--'s continuations with those in the literature. Fully first-class continuations are expensive, sometimes unpredictably so. Bruggeman, Waddell, and Dybvig (1996) observe, however, that many continuations, including those used to represent threads, are used at most once. In their Chez Scheme system, so-called *one-shot continuations* are restricted to at most one invocation, but they are much less expensive than first-class continuations, because neither capturing nor invoking a continuation require stack copying. On the other hand, their lifetime is dynamic: a one-shot continuation remains valid until used even if one throws to a continuation captured by a (transitive) caller. C--'s continuations occupy a different part of the design space; by compromising on lifetime, not number of uses, we get continuations that are cheaper still. We discuss this tradeoff further in Section 9.

4 Example: building threads on continuations

Using these continuations, the front-end runtime can support user threads by providing a library written in C--. We illustrate by defining `tfork`, a procedure that forks a new user thread. To show the flexibility of C--, we choose a more complex representation of threads than the simple continuations used to implement `yield`, above. Instead, we represent a thread by a *thread-control block* (TCB), which contains such information as the thread's saved continuation, its thread-local data, its priority, pointers used to put threads on queues, and so on. In particular, we assume that the very first word of a TCB is the (parameterless) continuation that should be `cut to` to resume execution of the thread. Different front ends could easily implement different design choices; this is the joy of Wand's idea.

We assume that we have defined these procedures as part of the front-end run-time system:

`alloc`	Allocate memory
`put`	Put a TCB on a queue
`release`	Deallocate memory
`reschedule`	Choose a ready thread and run it

Supposing that a TCB occupies 40 bytes, and we choose to put it at the base of a 4K stack, here is `tfork`:

```
tfork( bits32 f, bits32 x ) {
  bits32 tcb; /* TCB (and stack) of new thread */
  bits32 k;   /* Continuation for new thread */

  tcb = alloc(4096);   /* Allocate new TCB & stack*/
  k = NewContinuation( stk+40, run_thread, tcb );
                       /* Stack starts beyond TCB */

  /* Initialize and enqueue the TCB */
  bits32[tcb] = k; bits32[tcb+4] = f; bits32[tcb+8] = x;
  put( ready_queue, tcb );
  return;
}

run_thread( bits32 tcb ) {
   bits32 f,x;

   f = bits32[tcb+4]; x = bits32[tcb+8];
   f(x);               /* Do the job of this thread */
   release( tcb );     /* Deallocate the TCB+stack */
   jump reschedule();  /* Tail call the scheduler */
}
```

`tfork` allocates memory to hold the TCB and stack for the new thread, uses `NewContinuation` to initialize the stack, and stores `k`, `f` and `x` into the TCB (`bits32[a]` refers to the word of memory at address `a`). `tfork` then adds the new TCB to the ready queue and returns. The continuation `k` is stored in the first word of the TCB, where the scheduler expects to find it. The next two words hold `f` and `x` until the scheduler decides to run the new thread. It does so by cutting to `k`, which in turn calls the procedure `run_thread`. The latter retrieves `f` and `x` from the TCB, then runs `f(x)` (i.e., what the caller of `tfork` originally wanted) before releasing the storage for the thread and re-entering the scheduler.

You might wonder whether it is OK for `run_thread` to release the storage for the thread while still running on the stack held in that storage. Good question! This is exactly the kind of delicate question whose answer we want not to build into C--. In some situations it might be perfectly OK (e.g., the storage can't possibly be re-used until `Reschedule()` cuts to a new thread). In others, it might be a disaster. In the latter case, we can readily fix the problem by having a spare C-- continuation, with a stack of its own, that plays the role of a kernel stack. Then `run_thread` can cut to the kernel stack, which can release the storage held by the now-finished thread. The details are unimportant. What is crucial is that these decisions are made by the client; they are not built into C--.

5 Global variables and sharing

We now turn our attention to the second key problem from the Introduction: sharing. In general, some information should be shared among threads, while other information should be private. Furthermore, it is common to keep both kinds of information in registers, e.g., the allocation pointer, the exception stack, the current profiling bucket, or whatever. C-- provides direct support for such *global registers*, which are visible to all activations of a C-- computation. We devote this section to explaining how global registers interact with thread switching and concurrency, and in particular, how global registers can be made private or shared.[2]

[2] Given private global registers, it is easy for a C-- client to provide thread-local data; either the data itself or a pointer to it can be held in a global register.

5.1 Global registers

We begin by reviewing how global registers work in C--. Global registers are defined in C-- by a global register declaration. For example, this declaration specifies two global registers, **hp** and **tcb**, both of type **bits32**:

```
register bits32 hp;
register bits32 tcb;
```

The idea is that **hp** might be the allocation pointer, while **tcb** might point to the TCB of the currently-running thread. In the presence of such declarations, the globals can be used just like local variables. For example, we may write:

```
f( bits32 x ) {
  bits32[tcb+16] = x;
  return
}
```

This procedure stores **x** in the word beginning 16 bytes beyond where **tcb** points; it is probably a store to a thread-local location.

Global register declarations have the following Draconian constraints:

- Every C-- compilation unit in an application must contain identical global register declarations, in the same order. The use of global registers affects the calling convention generated by the C-- compiler, so it absolutely must be consistent.
- The programmer can assign global registers to particular machine registers by adding annotations, thus:

  ```
  register bits32 hp   "%eax";
  register bits32 tcb  "%ebx";
  ```

  Lacking such annotations, the C-- compiler puts as many globals as possible in registers, keeping the rest in memory. Variables declared first have highest priority for allocation to registers.
- You cannot take the address of a global register, so the C-- compiler can assume that they do not alias with each other or with any memory location.

5.2 Global registers and concurrency

What policies should be available to govern the sharing of global registers between threads? "Shared among all threads" and "private to each thread" are obvious policies, but for shared-memory multiprocessors finer distinctions are needed. For example, a common choice is to have a separate heap-allocation pointer for each *physical processor*. All the user threads that run on a particular processor use that processor's allocation pointer, which must be distinct from the allocation pointer used by user threads on other processors. This is clearly the sort of critical choice that should be up to the client. Accordingly, *C-- control transfers do not affect the global registers*. If the client wants to change the **tcb** global register, say, when switching threads, then he simply assigns a new value to it.

Unfortunately, at least some interactions between threads are likely to be mediated by the front-end run-time system, and we have already noted that much of the front-end run-time may be written in a proper programming language, which knows nothing about C-- global registers! To complete the story, we show how the front-end run-time system can get into the act *without* trashing the global registers.

5.3 Global registers and C code

The front-end run-time system should be written in a language designed for human programmers, but providing enough unsafe features to enable the implementation of garbage collectors, etc. Without loss of generality we assume this language is C. What we need, then, is a way to transfer control from C to C-- and back. In particular, it must be possible to define C-- procedures that are callable from C, and to call C from certain C-- procedures. Every transition between C and C-- must be identified explicitly, because C-- need not use C's calling convention.

To allow C to call C--, one defines a procedure in C-- using a **foreign** annotation:

```
foreign "C" f( bits32 x ) {
      ...
}
```

The **foreign "C"** part tells the C-- compiler to generate code for a procedure that expects to be called using C's calling convention. Such a procedure returns to C using **foreign "C" return(x)**.

Similarly, to allow C-- to call C, we provide a **foreign** version of the procedure call:

```
r = foreign "C" g( x, y );
```

So what happens to global registers? We cannot hope for them to survive a transition from C-- to C, because the C and C-- calling conventions need not be compatible. One possibility would be to automatically save and restore the globals across a **foreign** call, but that is not always right. For example, suppose that a user thread is suspended (its continuation captured) and later resumed. Some of its globals (e.g., the user-thread-local ones) should be restored just as they were when the thread was suspended. Others (e.g., the allocation pointer) should be shared among all user threads; or, on a multiprocessor, shared among the user threads bound to a particular OS thread.

These are clearly policy decisions! Here is how we expose them to the client:

- We provide C-- procedures that save and restore the global registers *en bloc*:

  ```
  SaveGlobals    ( bits32 gp )
  RestoreGlobals( bits32 gp )
  ```

  Each of these procedures takes as argument the address of a save area for the globals; this area must be provided by the front-end runtime. **SaveGlobals** stores the registers in the save area; **RestoreGlobals** loads the registers from the save area. It is an unchecked error to save the globals into a save area other than the one from which they were restored by the preceding **RestoreGlobals**.
- A **foreign** call kills all the global registers, and a **foreign**-callable procedure starts with all the global registers dead (but see also Section 5.4).
- The only C-- constructs that affect the values held in the global registers are **RestoreGlobals**, **foreign** call, and direct assignment to a global register. All the others, including call, return, **cut to**, **NewContinuation**, **SaveGlobals**, and so on, leave the global registers unchanged.

- The C-- run-time interface (see Section 2) includes a procedure that the front-end runtime can use to manipulate global registers:

  ```
  void *Cmm_FindGlobal( void *gp, int n )
  ```

  The call `Cmm_FindGlobal(gp,n)` returns the address of the `n`'th global in the save area `gp`. This allows the front end to get at the values saved by `SaveGlobals`.

Now the front end is in control. Typically, it will save the globals before exiting a C-- context and restore them before entering it again. Sometimes the address of the save area will itself be kept in a global register, so that it is conveniently available when the time comes to save the globals, but that choice is not built into C--. On a uniprocessor it might be perfectly OK to use a fixed memory location as the global save area, or on a multiprocessor there might be an OS-specific way to get at data local to an OS thread.

One question remains: how big should the client make the save area? C-- provides a static constant `%GlobalSize`, available in C-- code only, which gives the size in bytes. The client can, for example, implement a C-- procedure to return this value, or it can use C-- to allocate a suitable block of memory statically, thus:

```
section "data" {
  save_area : bits8[%GlobalSize]
}
```

Section 6 presents an example of the interactions between C-- threads and a garbage collector; this example illustrates the intended use of `RestoreGlobals` and `SaveGlobals`.

5.4 Atomic foreign calls

Some `foreign` calls invoke innocuous procedures, i.e, ones that cannot suspend, allocate, or do other strange things. In these cases we might like to avoid the expense of a complete `SaveGlobals`/`RestoreGlobals` pair, and instead simply ask the C-- implementation to save the globals across the foreign call. Purely as an efficiency enhancement, C-- offers an `atomic` call to a `foreign` procedure, thus:

```
r = atomic foreign "C" sin( x );
```

The keyword "`atomic`" tells C-- that the call is atomic with respect to thread switching, use of `Cmm_FindGlobal`, etc., *and* it instructs the C-- back end to save and restore any global registers that are considered volatile by the C calling convention. It can save them on the stack, or perhaps in C's callee-saves registers, but the globals are not available to `Cmm_FindGlobal` during the foreign call. Provided the foreign code keeps its promise of atomicity, the atomic foreign call is equivalent to a more expensive sequence: `SaveGlobals`, make foreign call, `RestoreGlobals`.

5.5 Implementation notes

C-- keeps as many as possible of the global "registers" in real, machine registers, but if there are too many globals, some must live in memory. In that case, a plausible implementation is to dedicate one machine register to point to the save area, and to access individual globals by indexing from this pointer. Even then, globals are better than arbitrary memory locations, because the code generator knows that the globals do not alias with other memory locations. Because we require that `SaveGlobals` be passed the same save area as `RestoreGlobals`, we can limit the cost of `RestoreGlobals` and `SaveGlobals` to the number of machine registers saved and restored.

```
void Cmm_FirstActivation(Cmm_cont k, Cmm_activation *a);
int  Cmm_NextActivation(Cmm_activation *a);
void *Cmm_FindLocal(Cmm_activation a, int n);

int Cmm_BytesToTop(Cmm_activation a);
Cmm_cont Cmm_CopyActivations (Cmm_activation *a,
                              void *stack, Cmm_ptr underflow);
```

Figure 3: Part of the C-- run-time interface, written in C

6 An example: garbage collection

As a concrete example of managing global registers, we show how to manage the control transfer between application code and a garbage collector. We use the same example to illustrate a second point: the same facilities that support concurrency are necessary to support garbage collection, *even in a non-concurrent system.* The garbage collector needs to inspect and modify the state of a suspended computation, including that computation's stack. In a non-concurrent setting, it is very desirable to do so *while using only one stack*, but this is a little tricky, because the garbage collector runs on the very same stack that it modifies. When we run on one stack, control starts in the front-end runtime, which initializes the heap, then calls the main C-- procedure of the compiled high-level program.

We assume that heap allocation takes place in a contiguous area whose extremities are pointed to by the global registers `hp` and `hlim`. At intervals, the compiled program must check for heap overflow, e.g., as in with the following C-- code, which allocates 20 bytes:

```
test:
  if (hlim - hp < 20) {
      call_gc();
      goto test:
  } else { hp = hp+20; }
```

The C-- procedure `call_gc` saves the global registers and calls the garbage collector, passing a continuation for the suspended computation. Here, `save_area` is assumed to be the label of a static global-register save area (Section 5.3):

```
gc() {
  SaveGlobals( save_area );
  foreign "C" gc( k );
  RestoreGlobals( save_area );
  return;

  continuation k:
    return;
}
```

The only reason to have `k` is to have a value to pass to `gc`, so that it knows where to start its stack walk.

The C-- run-time interface, part of which is summarized in Figure 3, provides access to the state of a suspended computation. The front end can look at an activation on the stack, find the locations of its local variables, find its caller's activation, and so on. The "handle" used to provide access to all this information is precisely the C-- continuation that is passed to `gc`, and which is used to resume execution of the thread. In particular, `Cmm_FirstActivation` initializes a data structure, of type `Cmm_activation`, that identifies an activation record on the stack. `Cmm_NextActivation` moves from one activation to the next one down, returning zero if there is none. `Cmm_FindLocal` returns the address of the `n`'th local variable in activation `a`. Other parts of the run-time interface, not presented here, allow the garbage collector to figure out which local variables hold pointers (Peyton Jones, Ramsey, and Reig 1999).

7 Stack management

The last of the key mechanisms identified in the Introduction is the management of stacks. C-- requires a computation to execute on a stack, but it is up to the client to decide how many stacks it wants and how big they should be. C-- is intended to support the following policies:

(A) The entire program, including both generated code and the front-end run-time system, executes on one big stack: the "system stack." This stack is effectively unbounded; the operating system arranges for it to grow until all of memory is exhausted, or until the size of the stack exceeds some limit assigned by the user. Stack exhaustion is a checked run-time error. This policy is suited only for single-threaded languages, but for those languages it is the policy of choice.

(B) A thread does not run on a stack, but instead uses continuation-passing style. This policy was popularized by Standard ML of New Jersey (Appel 1992) and Concurrent ML (Reppy 1991). It can be implemented as a special case of policy A, in which all threads share a degenerate stack containing only one frame, and every call is a tail call. Each thread yields only when its state is captured in a heap-allocated "client continuation", built and maintained entirely by the front end. C-- continuations are not used; instead, a thread yields by tail-calling the scheduler, passing the client continuation.

(C) Each thread runs on a bounded stack, the size of which is fixed at the time the thread is created. Stack exhaustion may be treated as a *checked* run-time error, as in SRC Modula-3, or as an *unchecked* run-time error, as in many threads packages for C. The client makes this decision by balancing the values of safety, efficiency, and ease of implementation.

(D) Each thread runs on a contiguous stack, which is enlarged as needed. Enlarging a stack may entail copying it to another memory location. The front-end run-time system and C-- must cooperate to ensure that stacks can be copied and that exhaustion is handled transparently. Several JVMs use this policy.

(E) Each thread runs on a segmented stack. New segments are added as needed and are returned to a common pool when no longer needed. For good performance, some hysteresis is needed to bound the overhead of "bouncing" across segment boundaries (Bruggeman, Waddell, and Dybvig 1996). Chez Scheme uses this policy.

(F) In variants of policies D and E, the client might want to move stacks around, or compact segments together, during garbage collection, rather than just moving a chunk on the top when stack overflow occurs.

The policies further down this list are extremely ambitious in the context of C--, because they involve such intimate cooperation between C-- and the client. In this paper we describe how to implement A-E, leaving F for further work.

To get off the ground, we need to know three things: in which direction the stack grows (easy, Section 7.1), how to detect stack overflow (moderately easy, Section 7.2), and what to do when the stack overflows (difficult, Section 7.3).

7.1 Stack direction

For each architecture there is usually a "preferred" direction of stack growth; that is, one that is better supported by the hardware. We deal with this question in a simple but brutal way: C-- assumes that the client knows the direction of stack growth. As we mentioned in the Introduction, C-- is not a "write-once, run-anywhere" language. There are a few architectural features that are very hard for C-- to hide without losing efficiency, but are very easy for the client to adapt to: word size, byte order, alignment requirements, direction of stack growth, and synchronization style (Section 8.2). C-- is dramatically simplified by exposing these features to the client, and we claim this simplification comes at very little cost to the client.

The direction of stack growth determines the specification of `NewContinuation`. If the stack grows up, `NewContinuation` must be given the address of the lowest-address byte of the stack; if the stack grows down, it must be given the address one byte beyond the highest-address byte of the stack.

7.2 Detecting stack overflow

To implement policy C, we need only the ability to detect stack overflow. This is a delicate matter, because we want the front end to decide whether and how to check for stack overflow, but the back end has critical information needed to perform such checks.

The primary policy question about detecting stack overflow is "how much should it cost?" The cost is paid at every call, so it must be low. Here are some common policies:

Omit the stack-overflow check.

Compare the stack pointer with a stack limit, obtained

(a) from a dedicated register,

(b) from some other well known location, or

(c) by masking out the least significant n bits of the stack pointer (for downward-growing stacks).

Guard page. Place an inaccessible page of memory just beyond the end of the stack. The compiler arranges for the first store into each procedure's activation record to be at the extreme young end of the frame; if the stack is exhausted, this store is guaranteed to cause a fault. This works unless the procedure's activation record is larger than a guard page (a statically-known fact), in which case the compiler must either test against a limit or generate multiple stores, one into each page of the new activation record.

The remaining policy decision that must be left to the front end is that only the front end knows whether it should be possible to recover from stack overflow.

The front end cannot implement these policies by itself, because only the C-- implementation knows these facts:

- in what register the stack pointer is stored
- how big the activation record is
- how much "headroom" is needed to save the state that is needed for recovery from stack overflow
- how to make a store into the extreme young end of the frame do useful work
- how to save the state needed to recover after a stack overflow

We need such close cooperation between front and back ends that it seems necessary to introduce a new C-- primitive.

The `limitcheck` primitive, which checks for stack overflow, is a bit like a conditional `cut to`; if the stack is exhausted, it cuts to an overflow handler provided by the front end. The syntax of this primitive is as follows:

> `limitcheck` [*limit*] [`guarded` *size*]
> `fails to` *continuation*
> [`recovers with` *continuation*];

Optional elements appear in brackets; the meanings of the elements are as follows.

- The *limit* is an expression used to compute the stack limit. This expression may refer to the C-- pseudo-variable `%sp`; this variable stands for the current value of the stack pointer, which is needed for limit-check policy (c) above. By making the front end provide the *limit* expression, we enable the front end to choose the cost of computing the stack limit.
- `guarded` *size*, if present, indicates that the front-end run-time system has placed an inaccessible "guard page" of size *size* just beyond the end of the stack. The *size* must be a compile-time constant expression. The default is size 0, i.e., no guard pages.
- The front end must provide a continuation to `cut to` if the stack overflows. It receives as its argument a continuation for the thread whose stack has overflowed.
- Using the optional `recovers with` clause, the front end may provide a continuation to `cut to` after the overflow has been resolved. Normally this continuation will be in the procedure that has just overflowed, either just before or just after the test. If the front end omits the `recovers with` continuation, the back end knows that stack overflow is an unrecoverable error. The back end can adjust its notions of "headroom" accordingly, and it can also avoid emitting the code needed to save state.

The front end must provide either a *limit* or a nonzero guard-page size; if it provides both, C-- uses the guard page for small frames and an explicit limit check for large ones.

Here, for example, is one way to test for stack overflow, at the entry point to procedure `f`:

```
import stack_overflow;
register bits32 stacklimit;
f( bits32 x ) {
  continuation test:
    limitcheck stacklimit
      fails to bits32[stack_overflow]
      recovers with test;
    ...rest of f...
}
```

The limit check requires that the current activation record fits below `stacklimit`, a global register. If not, it immediately cuts to a continuation stored in memory at address `stack_overflow`, passing continuation `test` as an argument. After the overflow is dealt with, the front-end run-time system can resume the thread by cutting to `test`.

Of course, if the limit check fails, the back end will need enough stack space to spill the parameters (`x` in this case) and the callee-saves registers (since `cut to` does not restore them) before transferring control to `stack_overflow`. The amount of space needed is bounded by the number of registers; if there are dozens of parameters, the ones that do not fit in registers will be on the stack already. The calling convention therefore requires that there always be enough space to dump the registers live at the call, and it is up to the *caller* to provide this space. So `f`'s `limitcheck` includes space for `f`'s callees to dump their registers if necessary. It is therefore unsafe for a C-- procedure that does not recover from overflow to call a C-- procedure that does recover from overflow.

An ambitious C-- compiler might use call-graph information to economize on stack-overflow checks, e.g., a procedure might check stack space for its (non-recursive) callees. Call-graph information might come from the compiler's own analysis or might be provided by a front end, using an annotation (`targets`) not otherwise mentioned in this paper. It is not obvious that the gain in efficiency would be worth the pain of complexity.

7.3 Dealing with stack overflow

If the stack overflows, we may `cut to` a continuation in the front-end runtime, which can either abort (the thread or the whole program) or can rearrange memory to increase the size of the stack. In general, memory can't be rearranged without copying all or part of the stack to a new block of memory, which is our topic for the rest of this section.

The recovery routine can use the run-time interface (Figure 3) to walk the stack of the thread whose stack has overflowed. At any time during this stack walk, it can call `Cmm_BytesToTop` to find how many bytes lie between the current activation and the youngest end. At some point—when the bottom of stack is reached (policy D), or when there are enough bytes to copy (policy E)—the stack walker will decide to copy all or part of the stack to somewhere new. It does this by calling `Cmm_CopyActivations`. `Cmm_CopyActivations` copies the segment of stack from, and including, the activation `a` up to the youngest end; that is, up to and including the continuation from which `a` was originally initialized with `Cmm_FirstActivation`. It copies this chunk of stack into a new block of memory pointed to by `stack`, and it returns a new continuation that can be used to resume the computation on its new stack.

When the computation resumes, it may eventually *underflow* the new stack, unless we copied all the activations on the old stack. `Cmm_CopyActivations` arranges that, when underflow occurs, the C-- procedure `underflow` is called. This procedure is passed as a parameter to `Cmm_CopyActivations`. If zero is passed then the following C-- procedure is used as default:

```
std_underflow (bits32 k, bits32 stack) {
       cut to k;
}
```

The first parameter to the underflow procedure is a zero-parameter continuation that resumes the computation on the old stack segment. The second parameter points to the block of memory originally given to `Cmm_CopyActivations`. It is ignored by the default underflow routine, but the client may supply an underflow routine that (for example) returns the block to a free pool.

7.4 Redirecting pointers

A serious complication with moving stacks around is that the client may hold pointers into the stack. These pointers include not only *C-- continuations* but also *client data* that is allocated on the stack using a mechanism not presented in this paper. There are many policies one might like to support for stack-allocated data.

- If stack-allocated data is immutable, it is safe to copy.
- If stack-allocated data is mutable, but the garbage collector never moves stacks, a good plan is to use the original activation, not any copies, as the "master" location for the data. The cost is that such data must be addressed indirectly, via a C-- local variable, rather than by a fixed offset from the stack pointer. Chez Scheme uses this policy for "heap objects" allocated on the stack. This policy is easily expressed in C--, but we omit the details.
- If stack-allocated data is mutable and the garbage collector moves stacks, we need support for redirecting pointers into the stack. This support presents no great difficulty in principle, but it adds to the complexity of `Cmm_CopyActivations`, and the details are beyond the scope of this paper.

Continuations are a little more delicate than client data. Consider the following example:

```
f( bits32 x ) {
      bits32 v ;
      v = x;

      g( x, k ) ;
      v = v+1 ;

      h( x ) also cuts to k ;
      return(v) ;

      continuation k:
        return( v ) ;
 }
```

Here, `g` presumably stores `k` somewhere, and `h` finds that pointer and `cuts to` it. But since `g` (or its callees) may provoke stack overflow, by the time `g` returns, `f` may resume execution in a *copy* of its original activation, and *it is in the copy that* `v` *will be modified.*[3] It would be an error for `h` to cut to the old stack, because that won't have an up-to-date copy of `v`.

One solution to this problem is to require the client to find all the continuations it has stored away, and redirect them to the new stack segment. This may be acceptable during garbage collection, when the store is being traversed anyway, but it is not acceptable when expanding a stack. A better solution is to require `Cmm_CopyActivations` to replace all continuations in the copied activations by indirections to their new locations. Now, if `h` throws to the old `k`, the old `k` will transparently throw to the new `k`.

We hope this story gives you an uneasy feeling. How complicated is this going to get? How do we know when we are done? We do not have a completely satisfactory answer. You may not be surprised; systems that support fully-mobile stacks and interior pointers, even when front and back ends are tightly integrated, are rare, complicated, and seldom well described. Nevertheless, the design we have presented can support the most common policies, and we see no insuperable obstacles to extending it to richer regimes.

7.5 Implementation notes

The abstractions described in this section are undoubtedly complicated to implement, which is precisely why we want to abstract over them. The main complications are these:

- To enable continuations to be forwarded transparently, the simplest strategy is to allocate a pair of words on the stack for each continuation. At allocation time, one word is initialized, to the program counter for the relevant continuation. If the activation is moved by `Cmm_CopyActivations`, the second word is made to point to the new copy of the continuation, and the program-counter word is changed to a trivial C-- procedure which throws to that continuation.
- When underflow occurs, C-- must arrange to dump the return values (there may be many) into the lower stack segment and construct a zero-parameter continuation to complete the return, which it then passes to the underflow routine. If the user has not supplied an underflow routine, then it is possible to return directly to the lower stack segment without dumping the return values. The `std_underflow` routine given above is a specification, not an implementation.

8 Pre-emption and synchronization

A full treatment of concurrency in C-- would cover pre-emption and synchronization, but we do not have space for that here, so we content ourselves with brief summaries.

8.1 Pre-emption

Pre-emption is typically triggered by an asynchronous event, e.g., a timer interrupt. The interrupt handler may simply make a small change in the state of the machine, for which each thread must poll frequently. Alternatively, the interrupt handler may suspend the interrupted thread, taking the "signal context" or "interrupt frame" supplied by the operating system and packaging it as a C-- continuation. The latter technique is well understood (Reppy 1990) but tricky to implement; it is a perfect candidate for C--.

When a thread can be interrupted and suspended anywhere, it is unclear what operations can be performed on a suspended thread. Clearly one can `cut to` such a thread, but can one safely walk the stack? Run the garbage collector? It may be too expensive to make a thread "collectible" at every instruction, although some recent results are encouraging (Stichnoth, Lueh, and Cierniak 1999). If not, and if it is necessary to run the collector, suspended threads must be rolled forward or rolled back to a "safe point." These are current research problems, as is whether or how the C-- back end should support safe points.

[3] We assume some hysteresis in the stack to prevent "bouncing".

8.2 Synchronization

For pre-emptive or multiprocessor implementations, C-- must support synchronization. Operating-system synchronization is easy but expensive; the sensible solutions for a custom code generator appear to be atomic hardware instructions or (for a uniprocessor only) restartable atomic sequences (Bershad, Redell, and Ellis 1992; Shivers, Clark, and McGrath 1999). We discuss only instructions here.

We have identified two families of atomic hardware instructions. The *explicit-protection* family, exemplified by the *test and set* instruction, requires that all access to shared data be protected by an explicit "lock bit." The *implicit-protection* family, exemplified by the *compare and swap* or *load-linked/store-conditional* instructions, can provide atomic operations on a word in memory even when some threads write to that word using ordinary store instructions. It is pointless to try to hide this distinction from the C-- client, because when using explicit protection, the front end must allocate a lock bit to protect every piece of shared data, whereas when using implicit protection, the front end can usefully manipulate a protected word without additional lock bits.

We have identified a useful abstraction for implicit-protection instructions: *weak compare and swap*. It is like an ordinary compare and swap, except it may nondeterministically fail even if the comparison succeeds. It can be used to solve a variety of synchronization problems, and it can be implemented easily and efficiently using either hardware compare and swap or a load-linked/store-conditional pair. It is therefore a good candidate for addition to C--.

9 Related work

The basic ideas about continuations, exceptions, context switching, and stack overflow are well known to those interested in compilers and run-time systems, who have implemented many variations on these themes. We have, however, found it hard to track down published descriptions. Even among what we have found, space limitations preclude a comprehensive treatment; we give only an illustrative sample of references.

Wand (1980) is the seminal paper on using continuations for concurrency. The technique has been used widely in languages with first-class continuations; Reppy (1991) presents an interesting example. Reppy (1990) makes a connection between pre-emption and continuations.

The trade-off between the Chez Scheme's one-shot continuations (Bruggeman, Waddell, and Dybvig 1996) and C--'s continuations is interesting. Chez Scheme's design requires a segmented stack, which C-- supports but does not require. Furthermore, capturing a one-shot continuation requires sealing an existing stack chunk and beginning a fresh one. This is a constant-time operation, but it is not nearly as cheap as saving a single program counter on the existing stack, which is what C-- does to capture a continuation (Section 3.3). For example, `call/1cc` would be a relatively expensive way to set up an exception handler. A C-- client could, however, implement Chez Scheme's design by using `NewContinuation` to allocate a new stack whenever a continuation was captured.

There are many "lightweight threads packages" for C; Cormack (1988) describes one interesting early example. Many of the techniques developed for these packages could be used to implement some of the primitives discussed here, but most of them embody policy decisions concerning stack overflow and thread scheduling, and none of them provides the crucial ability to inspect and modify the state of a suspended computation. In short, C-- is—by design—at a lower level; it should be possible to implement a C user-threads package using C--, but not vice versa. The package that is nearest in spirit to C-- is Keppel's QuickThreads package, which shares the explicit goal of providing minimalist mechanisms, leaving policy to the client (Keppel 1993). However, C-- separates the notions of capturing a continuation and cutting to one, whereas QuickThreads, whose objectives are more limited, combines both into a thread-switch operation. QuickThreads provides no support for global registers, stack walking, or stack-overflow checking and recovery.

Segmented stacks appear not to be widely covered in the literature, but Bruggeman, Waddell, and Dybvig (1996) repays careful scrutiny.

10 Conclusions and further work

We set ourselves the task of encapsulating well-known implementation techniques for concurrency in a way that enables the *client* to choose policies and cost tradeoffs, without having to rebuild the implementation. No other work known to us tries to abstract this particular interface, and it has proved surprisingly challenging.

We are working on an implementation of C--, in which we can test our design.

Acknowledgements

We gratefully acknowledge helpful feedback on this work from Andrew Black, Olivier Danvy, Kent Dybvig, Robert Ennals, Cedric Fournet, Benjamin Pierce, Todd Proebsting, John Reppy, and Martin Rinard.

References

Appel, Andrew W. 1992. *Compiling with Continuations.* Cambridge: Cambridge University Press.

Bershad, Brian N., David D. Redell, and John R. Ellis. 1992 (October). Fast mutual exclusion for multiprocessors. *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems,* in *SIGPLAN Notices*, 27(9):223–233.

Bruggeman, Carl, Oscar Waddell, and R. Kent Dybvig. 1996 (May). Representing control in the presence of one-shot continuations. *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 31(5):99–107.

Clinger, William, Anne H. Hartheimer, and Eric M. Ost. 1999. Implementation strategies for first-class continuations. *Higher-Order and Symbolic Computation*, 12(1):7–45.

Cormack, Gordon V. 1988 (May). A micro-kernel for concurrency in C. *Software—Practice & Experience*, 18(5):485–491.

Keppel, David. 1993. Tools and techniques for building fast portable threads packages. Technical Report UWCSE 93-05-06, Computer Science Department, University of Washington.

Peyton Jones, Simon L., Dino Oliva, and T. Nordin. 1997. `C--`: A portable assembly language. In *Proceedings of the 1997 Workshop on Implementing Functional Languages*, Vol. 1467 of *LNCS*, pages 1–19. Springer Verlag.

Peyton Jones, Simon L., Norman Ramsey, and Fermin Reig. 1999 (September). `C--`: a portable assembly language that supports garbage collection. In *International Conference on Principles and Practice of Declarative Programming*, Vol. 1702 of *LNCS*, pages 1–28. Springer Verlag.

Ramsey, Norman and Simon L. Peyton Jones. 2000 (May). A single intermediate language that supports multiple implementations of exceptions. *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 35(5):285–298.

Reppy, John H. 1990 (August). Asynchronous signals in Standard ML. Technical Report TR90-1144, Cornell University, Computer Science Department.

———. 1991 (June). CML: a higher-order concurrent language. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.* ACM.

Roberts, Eric S. and Mark T. Vandevoorde. 1989 (April). WorkCrews: An abstraction for controlling parallelism. Technical Report Research Report 42, Digital Systems Research Center, Palo Alto, CA.

Shivers, Olin, James W. Clark, and Roland McGrath. 1999 (September). Atomic heap transactions and fine-grain interrupts. *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '99),* in *SIGPLAN Notices*, 34(9):48–59.

Stichnoth, James M., Guei-Yuan Lueh, and Michal Cierniak. 1999 (May). Support for garbage collection at every instruction in a Java compiler. *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 34(5):118–127.

Wand, Mitchell. 1980. Continuation-based multiprocessing. In *Proceedings of the 1980 LISP Conference*, pages 19–28. Reprinted in *Higher-Order and Symbolic Computation*, 12(3):285-299, October 1999.