Representing Control in the Presence of First-Class Continuations *

Robert Hieb, R. Kent Dybvig, Carl Bruggeman
Indiana University
Computer Science Department
Lindley Hall 101
Bloomington IN 47405

Abstract

Languages such as Scheme and Smalltalk that provide continuations as first-class data objects present a challenge to efficient implementation. Allocating activation records in a heap has proven unsatisfactory because of increased frame linkage costs, increased garbage collection overhead, and decreased locality of reference. However, simply allocating activation records on a stack and copying them when a continuation is created results in unbounded copying overhead. This paper describes a new approach based on stack allocation that does not require the stack to be copied when a continuation is created and that allows us to place a small upper bound on the amount copied when a continuation is reinstated. This new approach is faster than the naive stack allocation approach, and it does not suffer from the problems associated with unbounded copying. For continuation-intensive programs, our approach is at worst a constant factor slower than the heap allocation approach, and for typical programs, it is significantly faster. An important additional benefit is that recovery from stack overflow is handled gracefully and efficiently.

1 Introduction

Stacks have traditionally been used to implement both activation records and local environments in languages that support recursive procedure calls [5]. Stacks allow rapid allocation and deallocation of call frames and efficient linkage on calls and returns. On modern architectures with hierarchical memory, stacks also help maintain locality of memory operations.

*This material is based on work supported in part by the National Science Foundation under grant number CCR-8803432.

Traditional stack management techniques are inadequate for some modern languages, however. If a language is to allow arbitrarily deep recursion, some means for detecting and recovering from stack overflow is necessary. Since multiple control threads in the same address space require multiple control stacks, it must be possible to scatter stacks throughout memory and each must occupy a bounded amount of space. Further complications result when a language provides a means for capturing, storing and reinstating control stacks, such as Scheme's first-class continuations [14] and Smalltalk's context objects [9].

The need to support continuations or contexts as objects with indefinite extent precludes the use of a simple stack-based implementation of call frames. A continuation represents the rest of the computation from a given point. Since the "rest of the computation" is determined by the current chain of activation records, when a continuation is captured, the current chain of activation records must be preserved, and when a continuation is reinstated, the current chain of activation records must be replaced with a previously captured chain so that the computation can continue from the point at which the continuation was captured. Since the same continuation may be reinstated more than once, reinstating a continuation cannot be accomplished by simply switching control back to the old stack area, since this would overwrite the saved activation records.

Stack overflow has traditionally been handled by locating the stack in an area of memory that can be extended indefinitely. A standard approach is to locate the heap at one end of memory and the stack at the other end of memory, and let them grow toward each other. Often, the memory management system can then be used to trap stack overflow. However, multiple execution threads require multiple stacks; such stacks cannot be easily extended if overflow occurs, and it may not be possible to use the memory system to trap stack overflow. Simply restricting the size of the stack area for a given control thread, either automatically or by programmer control, is unsatisfactory in languages that support and encourage the use of recursive programs. Since stack overflow and underflow can be thought of as continuation capture and reinstatement, it is not surprising that a method that allows efficient continuation operations also provides the means for handling stack overflow and underflow efficiently.

The simplest way to allow continuation operations and multiple control threads, and at the same time avoid stack overflow problems, is to allocate activation records as a linked list in the heap. Such an approach has the virtue of simplicity, but the price is increased allocation and more expensive linkage at procedure call and return. In this paper we show how stacks can be used to implement activation records in a way that is compatible with continuation operations, multiple control threads, and deep recursion. Our approach allows a small upper bound to be placed on the cost of continuation operations and stack overflow and underflow recovery. Since we do so while retaining the benefits of traditional stack management, ordinary procedure calls and returns are not adversely affected. Although the cost of continuation operations is greater than it would be in a heap model, the increased cost is defrayed by the less expensive procedure call interface. One important feature of our method is that the stack is not copied when a continuation is captured. Consequently, capturing a continuation is very efficient, and objects that are known to have dynamic extent can be stack-allocated and modified since they remain in the locations in which they were originally allocated. By copying only a small portion of the stack when a continuation is reinstated, reinstatement costs are bounded by a small constant.

In the next section we provide some background for our work. We discuss the use and importance of first class continuations and review other methods for implementing them. In Section 3 we describe our stack model. In Section 4 we use this model to develop techniques for continuation capture and restoration. In Section 5 we show how these techniques can be used to handle stack overflow and underflow, and we present an efficient method for detecting stack overflow in the absence of hardware and operating system support.

2 Background

We have used our stack management techniques for implementing Scheme on several machines and operating systems. Scheme is a good test bed for these techniques because it relies heavily on procedure calls and provides access to continuations as first-class objects. In fact, conditionals, procedure calls, and continuations are the only control operations provided by Scheme. Looping is accomplished by tail-recursive procedure calls, and support for exception handling and "gotos" is provided by continuations.

Continuations in Scheme are procedure objects that represent the remainder of a computation from a given point in the computation. The procedure *call-with-current-continuation*, commonly abbreviated *call/cc*, allows a program to obtain the current continuation. When given a procedure of one argument, *call/cc* creates a continuation procedure and passes it to the argument procedure. The procedure created by *call/cc* represents the continuation of the call to *call/cc*.

When the continuation procedure is invoked, it returns its argument to the continuation of the call to *call/cc* that created it. In essence, the argument passed to the continuation procedure is returned as the result of the call to *call/cc*. If control has not otherwise passed out of the call to *call/cc*, invoking the continuation merely results in a nonlocal exit. If control has already passed out of the call to *call/cc*, the continuation may still be invoked, but the result is to restart the computation at a point from which the system has already returned. This feature may be used to implement many interesting control structures, including loops, nonblind backtracking [16], coroutines [8], and engines [10, 7].

The continuation of a procedure call is nothing more than the control stack of procedure activation records. If continuations were used only for nonlocal exits, as in Common Lisp [15], then the essence of a continuation object would be a pointer into the control stack. However, because continuations can outlive the context of their capture, continuation objects have indefinite extent and a pointer into the stack is not sufficient. If control passes out of the context where the continuation was created, the stack may be overwritten by other procedure activation records, and the information required to return to the continuation will be lost.

The simplest way to support continuation operations is to abandon the use of a reusable stack to store activation records and to maintain activation records as a linked list in the heap (see Figure 1). In this model, previous activation records are never overwritten; instead, a new activation record is allocated for each call. On return, the activation record is not automatically deallocated, since if a continuation has been captured it may be needed later. Instead, a storage manager reclaims the record when it is no longer reachable. The chief advantage of this approach is that the capture and invocation of a continuation is quick and easy. A continuation may be captured or reinstated for little more than the cost of an ordinary procedure call. Also, there is no need for a separate stack overflow detection and recovery mechanism; stack overflow is simply a special case of heap overflow. Multiple control threads are also easily

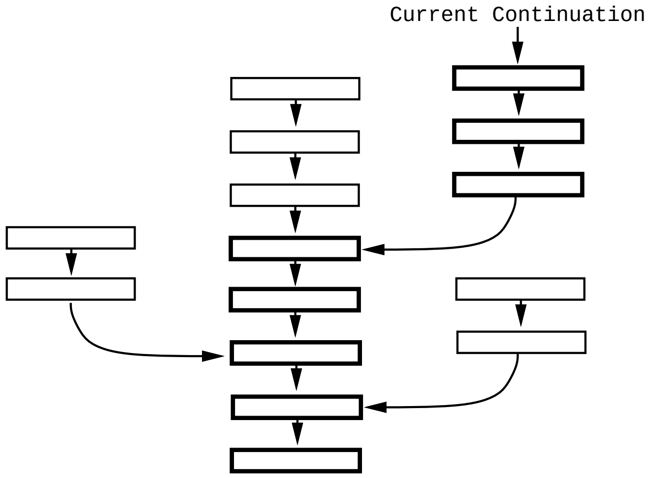

Figure 1. The heap model provides the simplest method for supporting constant-time continuation operations. An activation record is allocated in the heap and linked to the current activation record before a call is made. The called procedure uses the link to restore the old frame pointer before returning. Continuation operations involve saving or restoring a pointer to the current frame.

accommodated. The disadvantage is that ordinary procedure calls may be slowed down by the increased overhead caused by allocating the activation records in the heap and by more complicated activation record linkages. Furthermore, the storage manager must do more work to reclaim abandoned activation records.

Appel [1] points out that heap allocation and the associated cost of garbage collection can be made competitive with stack allocation by using large physical memories. The argument is based on the fact that a copying collector takes time proportional to the amount of retained data rather than discarded data. Thus, if memory is sufficiently large in comparison to the amount of retained data, the cost of garbage collection becomes insignificant. In fact, assuming that each stack frame must be explicitly deallocated (popped), the overall cost of heap allocation operations can actually be made less than that of stack allocation operations. Unfortunately, this argument may not apply to all memory systems, since it assumes that arbitrarily large amounts of memory can be used without penalty, whereas hierarchical memory systems that use caches and virtual memory penalize programs that use large amounts of memory without a high degree of locality.

Since in the heap model frames are not contiguous in memory, the frame pointer must be saved and restored on each call, resulting in an extra memory write and read for each recursive call. The stack model (see Figure 2), on the other hand, can combine frame allocation, deallocation and linkage by adjusting the frame pointer by a small constant on procedure call and return. Also, since the heap model must assume that a frame may be captured as part of a continuation, the frame cannot be reused or modified. With the stack model, on the other hand, portions of a frame may be reused for local storage or subordinate calls.

In order to preserve the benefits of stack management of activation records, some implementors have used a copy strategy. The copy strategy uses ordinary stack management techniques until a continuation is captured or invoked. When a continuation is captured, the stack is copied into the heap and a pointer to the heap copy is stored in a continuation structure. When a continuation is invoked, the stack image in the heap is copied into the stack area, where it is treated as an ordinary stack of activation records. The first reference we have found to this approach is by McDermott [12], who suggests copying continuations to and from a control stack so that only programs that actually use first class continuations need pay for the cost of supporting them.

Unless continuation operations are relatively rare or the size of the stack is usually quite small, the cost of copying stack images makes continuation operations inordinately expensive. It is possible to construct programs that cause the naive copy model to behave very poorly, since the cost of a continuation operation is proportional to the size of the stack. Furthermore, since many copies of an arbitrarily large continuation may be retained if the same continuation is captured and saved repeatedly, a large amount of memory may be wasted, resulting in much worse memory usage than the supposedly memory-intensive heap model.

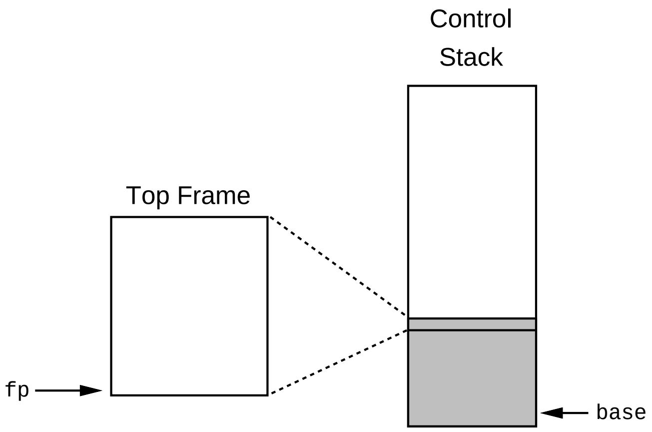

Figure 2. The traditional stack model provides the least expensive frame linkage. Since frames are physically adjacent, frame links can be maintained by simple register adjustments. However, since continuation operations require time proportional to the size of the active portion of the stack, the amount of time required is effectively unbounded.

Since allocation and reclamation on a stack is inexpensive, objects that are known to have dynamic extent, that is, do not survive the call frame in which they are allocated, are often allocated on the stack as part of the call frame. However, under the copy model this sort of stack allocation is not likely to be useful. It is not possible, in general, to retain pointers to such objects or to modify their contents, because the stack in which they are allocated may be moved out of the stack area and into the heap, perhaps more than once.

Much recent work has been devoted to developing techniques that allow the stack model to be used without making the use of continuations too expensive. For instance, Bartley and Jenson [2] "optimistically" stack-allocate control frames, but temper their optimism by using a stack cache of limited size. This places a bound on the worst-case costs of continuation capture and reinstatement, since a bounded amount of memory is copied. Since all but the top frame of the stack cache can be copied into the heap on stack overflow—essentially forcing a continuation capture—deep recursion is still possible. However, there is a direct relationship between the bound on the cost of continuation operations and the bound on the depth of recursion without stack overflows. Since handling stack overflow and underflow is expensive compared with the cost of ordinary procedure calls, a small stack size can lead to a substantial decrease in the performance of recursive programs. In the worst case, a "bouncing" phenomenon may occur. If a program makes just enough recursive calls to place the stack on the verge of overflow and then enters a loop that causes the stack to repeatedly overflow and underflow, the worst-case cost of recursive procedure calls can become the average-case cost, making calls as expensive as continuation operations.

Our method for representing control threads also limits the amount of memory copied by continuation operations without requiring the small stack size that results in increased overflow and underflow overhead for programs that do not use continuations.

3 The Control Stack

In our model, the control stack is represented as a linked list of *stack segments* (see Figure 3). Each stack segment is structured as a true stack of *frames* (*activation records*), with one frame for each procedure call. A *stack record* associated with each stack segment contains information about the stack segment, including:

1. a pointer to the base of the stack segment,
2. a pointer to the next stack record,
3. the size of the stack segment, and
4. the return address for the topmost frame.

Each frame consists of a sequence of machine words. The first word at the base of the frame is the return address of the current active procedure. The next n words contain the n actual parameters of the procedure, or pointers to cells in the heap containing the actual parameters if the parameters are assignable [6, 13]. (It is also possible to pass the return address and the first few arguments in registers, leaving a hole in the frame in which the return address can be placed if the called

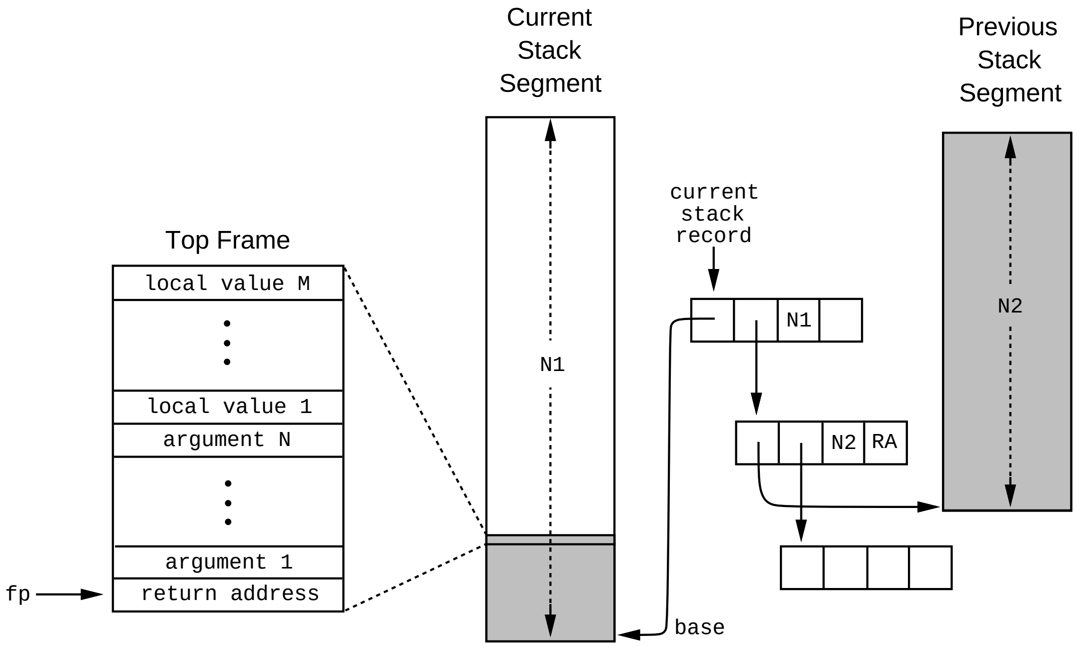

Figure 3. The segmented stack model is a simple generalization of the traditional stack model. By implementing the control stack as a linked list of stack segments, continuation operations are bounded by the size of the top segment instead of the size of the entire control stack.

routine itself makes a recursive call.) The remaining words in the frame contain the values of local variables, compiler temporaries, and partial frames for procedure calls initiated but not yet completed. A frame pointer register, *fp*, points to the base of the current frame, which is always in the topmost stack segment.

No separate stack pointer is maintained to point to the topmost word on the stack, so there is often a gap between the frame pointer and the topmost word. This does not create any difficulties as long as this stack is not used for any other purpose (such as asynchronous interrupt handling). Using a frame pointer instead of a stack pointer simplifies argument and local variable access and eliminates register increments and decrements used to support stack "push" and "pop" operations. This savings is more important on architectures, such as RISC architectures, that do not support auto-increment and auto-decrement addressing modes.

No explicit links are formed between frames on the stack. Many compilers place the current frame pointer into each stack frame before adjusting the frame pointer to point to the new frame. This saved pointer, or *dynamic link*, is used by the called routine to reset the frame pointer and by various tools, *e.g.*, exception handlers and debuggers, to "walk" the stack. In our model, the frame pointer is adjusted just prior to a procedure call to point to the new frame, and is adjusted after the called routine returns to point back to the old frame. In order for this to work, the frame pointer must still (or again) point to the called routine's frame on return. The compiler generating code for the calling procedure must keep track of the displacement between the start of the calling procedure's frame and the start of the called procedure's frame in order to adjust the frame pointer both before and after the call. In both cases, the adjustment is performed by a single instruction to add (subtract) the displacement to (from) the frame pointer.

Exception handlers, debuggers, and other tools that need to walk through the frames on the stack must have some way to get from each frame to the preceding frame. Our continuation mechanism also requires this ability in order to find an appropriate place at which to split the stack (see Section 4). In the place of an explicit dynamic link, the compiler places a word in the code stream that contains the size of the frame. This word is placed immediately before the return point so stack walkers can use the return address to find the size of the next stack frame (see Figure 4). If the return address itself is always placed in a known frame location, the frame size effectively gives the offset from the return address of the current frame to the return address of the preceding frame. For Scheme, it is useful to have the return address stored at the base of the frame so that it need not be moved for tail recursive calls.

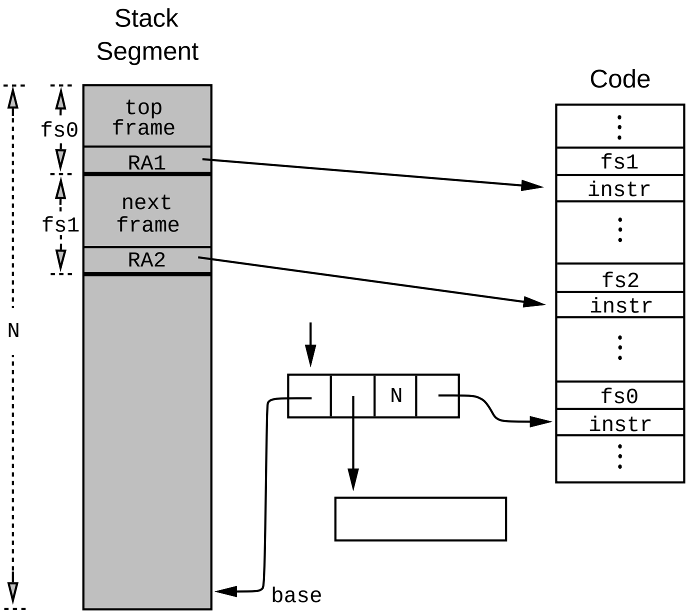

Figure 4. Walking backwards through a stack segment is straightforward. The return address field of a continuation stack record points to an instruction in the code stream, which is preceded by a data word containing the frame size. This frame size is used to to find the base of the top frame, where its return address is stored. This return address is used to find the frame size of the next frame down, which is used to find the next return address, *etc.*

Assuming that the compiler always generates an instruction to reset the frame pointer immediately at the point of return, the stack walker could disassemble this instruction to determine the frame size and we could thereby avoid storing the frame size explicitly in the code stream. This would, however, complicate the stack walker and unnecessarily constrain the compiler, which would otherwise be able to move the frame pointer directly to the base of the frame for the next procedure call in many cases. The constraint that the return address be placed at a constant offset in the frame can also be relaxed by storing the actual offset in the code stream along with the frame size.

4 Continuation Operations

When the system is initialized, a large stack segment and an associated stack record are created. The initial stack segment is large for two reasons: first, so that stack overflow for deeply recursive programs is less likely, and second, because continuation captures shorten the stack. Each time a continuation is captured (see Figure 5), the occupied portion of the current stack segment is sealed and the current stack record is converted into a continuation object by adjusting the size field and storing the current return address in the return address field. The return address in the current frame is replaced by the address of an underflow handler (see below). A new stack record is allocated to serve as the current stack record. Its base is the address of the next word above the occupied portion of the old stack segment, its link is the address of the old stack record (the continuation), and its size is the number of words remaining in the old stack segment. The stack is thus shortened each time a continuation is captured. Creating a continuation, therefore, does not entail copying the stack, but it does shorten the current stack object, which eventually results in stack overflow and the allocation of a new stack object (see Section 5). If a continuation were captured before each recursive procedure call, each saved stack segment would contain exactly one frame, and the resulting list of continuation objects would be essentially equivalent to a heap-based control stack.

If the current stack segment is empty when a continuation is captured, no changes are made to the current stack record and the link field of the current stack record serves as the new continuation. This is necessary to implement tail recursion properly, *i.e.*, so that no growth in the control stack occurs when continuations are created repeatedly in a tail-recursive situation.

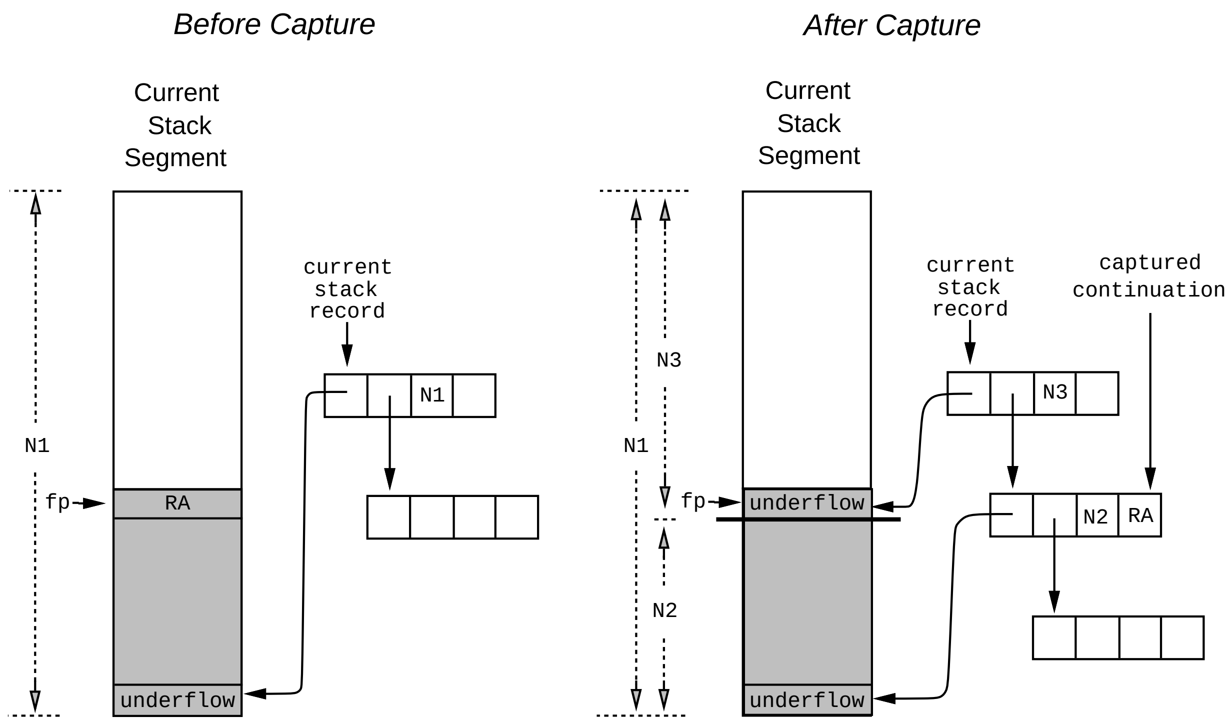

Figure 5. Capturing a continuation is a constant-time operation with the segmented stack model. The current stack segment is divided into two segments at the top frame. The bottom segment is the stack segment for the captured continuation, and the top segment becomes the current stack segment.

For instance, the following Scheme function should loop indefinitely since it calls itself tail-recursively:

(**define** *looper*
  (**lambda** ()
    (*call/cc*
      (**lambda** (k)
        (*looper*)))))

If a new link were added to the control stack at each iteration of *looper* because of the call to *call/cc*, the control stack would grow progressively longer and the program would eventually run out of memory.

Reinstating a continuation is more complex (see Figure 6). In the simplest case, the current stack segment is overwritten with the stack segment from the continuation, and the frame pointer is adjusted to point to the top frame of the copied segment. If the current stack segment is not large enough a new one is allocated.

Unfortunately, the size of a saved stack segment is bounded only by the size of the initial stack segment. Since stack segments are allocated in large chunks to reduce the frequency of stack overflows, if the whole segment were copied at once, the cost of continuation reinstatement would be bounded only by this large amount. This can be prevented by placing an upper bound on the amount copied. If the size of a saved stack segment is less than or equal to this bound, the entire segment is copied. Otherwise, the segment is first split into two segments such that the size of top stack segment is less than the copy bound. Although it would be sufficient to split off a single frame, it is more efficient to split off as much as possible without exceeding the bound because of the overhead of splitting the continuation and initiating the copy. An appropriate bound for a given machine can be determined only by experimentation.

Finding an appropriate splitting point entails walking backwards through the continuation stack segment (see Figure 4) until adding another frame would exceed the copy bound. The stack segment is then split in much the same way the stack is split when a continuation is captured (see Figure 7). The base and link pointers from the continuation stack record and the return address from the frame above the splitting point are stored in a newly allocated stack record. The size field of the new stack record is set to the size of the stack segment below the splitting point. The new stack record becomes the stack link for the old stack record. The old stack record's base pointer is set to the splitting point and its size field is set to the size of the stack segment above

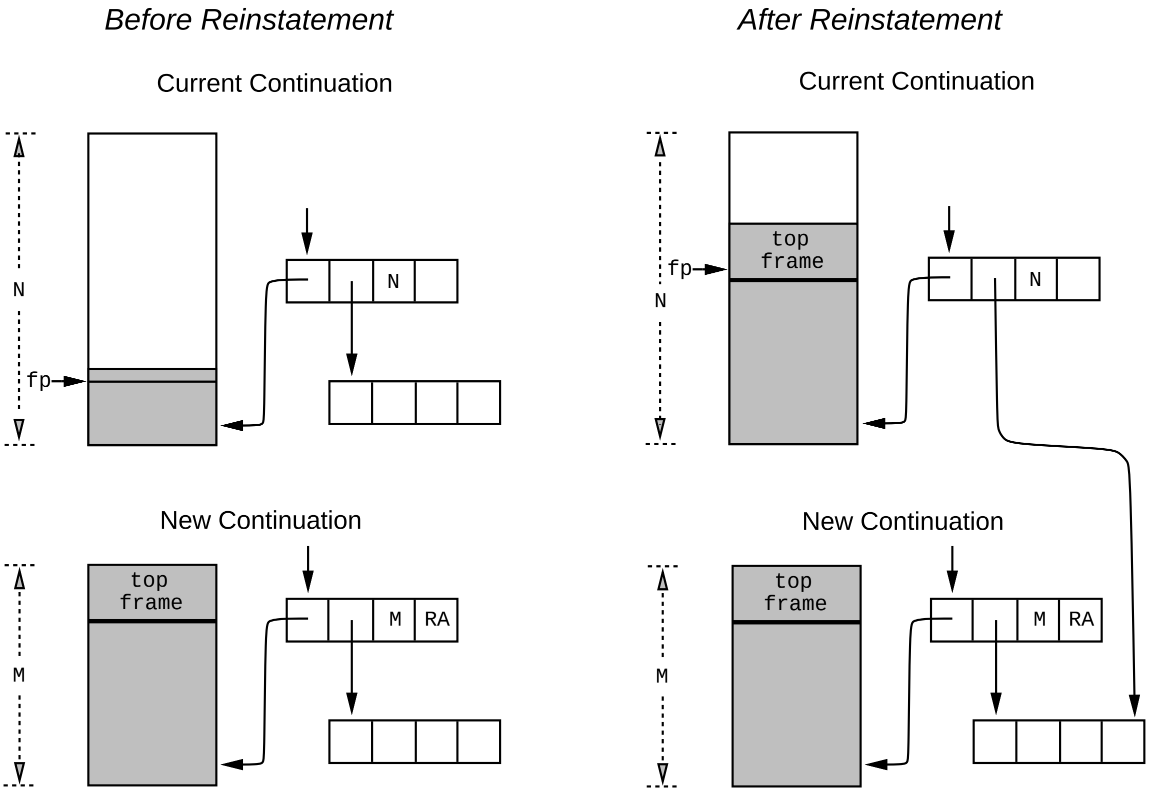

Figure 6. When a continuation is reinstated, the contents of the stack segment of the continuation is copied into the current stack segment. If the size of the stack segment is greater than a predetermined limit, the segment is first split into two segments (see Figure 7). If the current stack segment is not large enough to hold the contents of the reinstated stack segment, a new stack segment is allocated.

the splitting point. The return address in the frame is replaced with the address of the underflow handler.

Since at least one frame must be copied when a continuation is reinstated, if the amount of copying is to be bounded the size of a frame must be also bounded. This bound can be the same as the bound used in splitting continuations, but in practice it is reasonable to make it larger if frames larger than the optimum splitting size are not unusual. The frame bound then determines the worst-case cost and the copy bound determines the average-case cost of continuation invocations. In order to maintain a bound on frame size, the number of arguments to a procedure and the amount of storage necessary for local bindings and intermediate results must be limited. Extra arguments can be passed in a auxiliary data structure and the number of local bindings can be limited by converting local binding blocks into unnamed procedures. Intermediate results for pending calls and other operations can be stored in a linked list in the heap. In practice, with a reasonably large frame bound, these conversions are rarely necessary.

It is necessary to do something special when a return is attempted from a call frame that is at the bottom of a stack segment. The initial stack segment has as its return address at the base of the segment the address of a routine that exits to the operating system. All other segments have the address of the underflow handler stored at the base of the segment. The underflow handler simply reinstates the continuation in the link field of the current stack record.

5 Stack Overflow

When the heap model of continuation allocation is used, the depth of recursion is limited only by the amount of available heap memory. However, with a stack-based implementation of the control stack, some method for stack overflow detection and recovery is necessary to allow indefinite depth of recursion. If stack overflow can be detected while the system is in a known state, overflow can be treated as an implicit continuation capture.

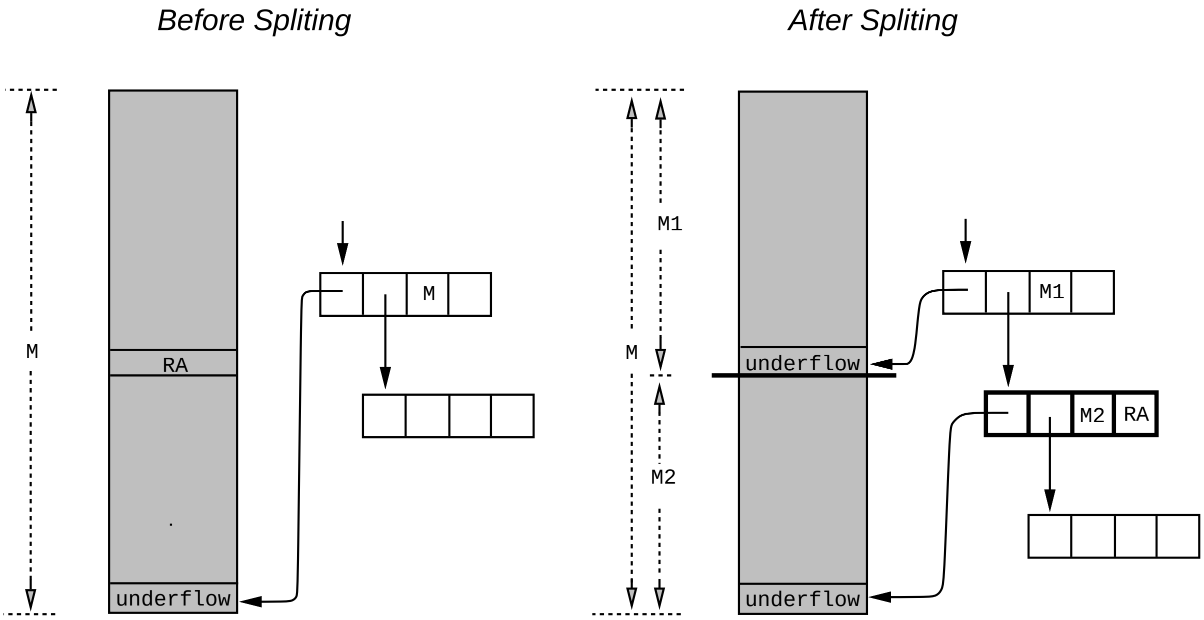

Figure 7. Large stack segments must be split before being reinstated. A splitting point is found by walking the stack to find the frame that gives the largest value for $M1$ without exceeding the predetermined limit on copying. The return address at the splitting point is stored in a new stack record and the address of an underflow handler is stored in its place.

Unfortunately, detecting stack overflow inexpensively is not a simple matter, particularly if the goal is a portable implementation model. Furthermore, detecting overflow is only part of the problem, since the state of the system at the time of overflow must be known completely so that the computation can be continued after a new stack area is allocated and linked to the continuation containing the previous area.

On some architectures, stack overflow detection can be made virtually cost-free with the help of the memory management system. The stack can be located adjacent to an area of memory that is not writable and an exception generated when the stack attempts to grow into the unwritable area. Such an approach is especially compatible with the copy-in, copy-out model of continuation management, since one can permanently place the stack cache next to a suitable region of memory. However, our model requires that new stack areas be allocated on demand, so it is essential that the hardware and operating system allow areas of memory to be selectively marked read-only.

Even on machines on which it is possible to reliably generate memory faults as a means for detecting stack overflow, it still may not be possible to recover from the stack overflow if the hardware and operating system do not preserve enough of the state of the system. In particular, it may not be possible to determine which instruction caused the overflow or what the contents of the registers were when the overflow occurred.

If all requirements can be met, then the memory management system can be used to create a no-cost solution to the stack overflow detection problem. The process of recovering from stack overflow does not in itself need to be extremely efficient. Since our system allows arbitrarily large stack areas and does not suffer from the danger of "bouncing" back and forth between overflow and underflow, arbitrarily large amounts of computation can be done between stack overflows to amortize their cost. Consequently, the efficiency of the host system's memory management and exception handlers is not an issue.

Unfortunately, it has been our experience that memory exceptions are not a tenable means for detecting stack overflow on many of the machines on which we have implemented Scheme. Either we cannot find a reasonable way to generate them or we cannot restore the state of the system after they have been detected. Furthermore, even on machines for which one could use the memory management system, the programmer time necessary to implement and maintain a special system for each machine may not be reasonable. As a result, we have turned to explicit checks for stack overflow detection.

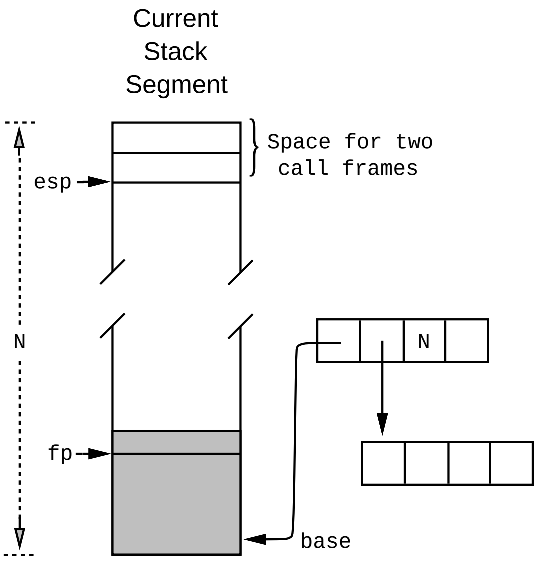

Figure 8. The end-of-stack pointer, *esp*, always points to a region before the actual end of the stack. This region must contain enough space for two call frames. Reserving room for two frames simplifies the overflow check for procedures that make recursive (non-tail) calls and eliminates the overflow check for procedures that do not make recursive calls.

The challenge is to implement the overflow checks with minimal impact on the speed of procedure calls. Our implementation uses two tactics to minimize the cost of overflow detection. The first is to make the test as inexpensive as possible. The second is to avoid the test whenever possible.

The most important part of making the test inexpensive is to avoid memory references by keeping an end-of-stack pointer (*esp*) in a register (see Figure 8). Since the frame pointer (*fp*) is already in a register, checking for stack overflow requires a simple register compare and branch. We point the *esp* a constant amount before the actual end of the stack area so that the comparison does not have to take into account the expected frame size as long as it does not exceed the *esp* offset. In fact, if the size of a frame is bounded (see Section 4), the *esp* offset can be set so that the overflow check need never take into account the frame size.

In some cases, we can avoid the stack overflow check entirely. First we make the *esp* offset even larger—large enough so that a procedure that only uses a bounded amount of stack space need not check for overflow. If space for an extra frame is maintained at the end of the stack by procedures that do make recursive (as opposed to tail recursive) calls, then procedures that do not make recursive calls need not check for overflow. The result is that leaf routines and routines that form a tight, tail-recursive loop need not check for overflow.

Additional stack overflow checks can be eliminated by static analysis of the code. First, some procedures contain paths that meet the criteria for eliminating stack checks, even if the procedure as a whole does not. Thus it is desirable to delay the stack check until it is known that a recursive call will be made, but not so long that it becomes necessary to repeat the check. Second, the compiler can determine how much stack space is used by some recursive calls. If the called procedure uses a known, bounded amount of stack space and the sum of its space requirements and those of the current frame size is less than the space reserved for non-checking procedures, then the calling procedure need not check for stack overflow.

6 Conclusions

Our approach to stack management supports bounded-time continuation operations and stack overflow recovery without adversely affecting the efficiency of procedure calls and returns. Creating a continuation is efficient, requiring only the creation of a small stack record and the adjustment of a small number of fields in an existing stack record. Reinstating a continuation requires copying a saved stack segment over the current stack segment, perhaps after first splitting the continuation to limit the size of the copied segment. Stack overflow and underflow recovery are essentially identical to continuation creation and reinstatement. Various problems with naive stack copying approaches to supporting continuations are solved by our approach. Continuation operations are bounded, stack overflow occurs infrequently, overflow/underflow "bouncing" is avoided, and stack allocation is possible for data objects with dynamic extent.

The main advantage of our approach over the approach of heap allocating a linked list of call frames is that procedure calls and returns do not have to maintain explicit frame linkage information. In addition, our method is less memory-intensive, consuming less heap space and exhibiting greater locality of reference. As a result, our approach results in smaller indirect costs from garbage collection, cache misses, and page faults. Our approach does not suffer from some of the limitations of the heap-based approach, such as the inability to reuse frames and the inability to stack-allocate objects with dynamic extent.

Clinger, et. al. [3], argue that a hybrid stack/heap mechanism may be most appropriate for Scheme and Smalltalk. Their mechanism provides for the frames to be allocated on a stack and moved into a heap-allocated linked list when a continuation is created. This list remains in the heap indefinitely and the frames in the list are never copied back onto the stack. Procedure returns must check whether or not they are returning from a frame on the stack, which requires adjustment of the stack pointer, or on the heap, which requires following the frame link. The hybrid stack/heap model suffers from some of the disadvantages of the pure heap model. In particular, a small additional cost is paid for procedure returns (but not calls) and objects with dynamic extent cannot generally be stack allocated because they move if a continuation is created. In addition, the stack must be kept small so that the cost of creating a continuation is bounded, which results in more frequent stack overflows. The primary advantage of the hybrid stack/heap mechanism is that there is never more than one copy of a given frame. They were motivated by Danvy [4], who pointed out that multiple continuation copies can lead to unbounded allocation. While our approach does not avoid duplication of stack frames, the bound we place on stack segment size on continuation reinstatement places a bound on the amount of duplication, and the amount of memory resulting from this duplication is at worst a constant factor more than with the stack/heap approach.

We have implemented the continuation and overflow mechanisms described in this paper and incorporated them into the implementation of *Chez Scheme*. We have not modified the compiler to enforce the frame size bound. It is not clear that doing so would be worth the effort; static analysis of the source code for *Chez Scheme* indicates that 99% of all frames are smaller than 30 words, and we suspect that the dynamic behavior is skewed toward even smaller frames. We are investigating the use of similar mechanisms in the implementation of concurrent continuations [11].

Acknowledgement: We wish to thank Olivier Danvy for providing comments on an earlier draft of this abstract.

References

[1] Andrew W. Appel, "Garbage collection can be faster than stack allocation," *Information Processing Letters 25*, 1987, 275–279.

[2] David H. Bartley and John C. Jensen, "The Implementation of PC Scheme," *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, August 1986, 86–93.

[3] William D. Clinger, Anne H. Hartheimer, and Eric M. Ost, "Implementation Strategies for Continuations," *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, July 1988, 124–131.

[4] Olivier Danvy, "Memory Allocation and Higher-Order Functions," *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, June 1987, 241–252.

[5] Edsger W. Dijkstra, "Recursive Programming," in *Programming Systems and Languages*, Saul Rosen (ed.), McGraw-Hill, NY, 1967.

[6] R. Kent Dybvig, *Three Implementation Models for Scheme,* University of North Carolina at Chapel Hill Department of Computer Science Technical Report #87-011 (PhD Dissertation), April 1987.

[7] R. Kent Dybvig and Robert Hieb, "Engines from Continuations," Indiana University Computer Science Department Technical Report No. 254, July 1988.

[8] Daniel P. Friedman, Christopher T. Haynes and Mitchell Wand, "Obtaining Coroutines with Continuations," *Computer Languages 11*, 3/4, 1986, 143–153.

[9] Adele Goldberg and David Robson, *Smalltalk 80: the Language and its Implementation*, Addison-Wesley, 1983.

[10] Christopher T. Haynes and Daniel P. Friedman, "Abstracting Timed Preemption with Engines," *Journal of Computer Languages 12*, 2, 1987, 109–121.

[11] Robert Hieb and R. Kent Dybvig, "Continuations and Concurrency," Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), March 1990 (to appear).

[12] Drew McDermott, "An Efficient Environment Allocation Scheme in an Interpreter for a Lexically-Scoped Lisp," *Conference Record of the 1980 Lisp Conference*, August 1980, 154–162.

[13] David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams, "Orbit: An optimizing compiler for Scheme," *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, published as *SIGPLAN Notices 21*, 7, July 1986, 219–233.

[14] Jonathan A. Rees and William Clinger, eds., "The Revised^3 Report on the Algorithmic Language Scheme," *SIGPLAN Notices 21*, 12, December 1986.

[15] Guy L. Steele Jr., *Common LISP: The Language*, Digital Press, 1984.

[16] Gerald J. Sussman and Guy L. Steele Jr., "Scheme: an Interpreter for Extended Lambda Calculus," Massachusetts Institute of Technology Artificial Intelligence Memo 349, 1975.