Representing Control in the Presence of One-Shot Continuations*

Carl Bruggeman Oscar Waddell R. Kent Dybvig

Indiana University Computer Science Department
Lindley Hall 215
Bloomington, Indiana 47405
{bruggema,owaddell,dyb}@cs.indiana.edu

Abstract

Traditional first-class continuation mechanisms allow a captured continuation to be invoked multiple times. Many continuations, however, are invoked only once. This paper introduces *one-shot* continuations, shows how they interact with traditional multi-shot continuations, and describes a stack-based implementation of control that handles both one-shot and multi-shot continuations. The implementation eliminates the copying overhead for one-shot continuations that is inherent in multi-shot continuations.

1 Introduction

Scheme [5] and some implementations of ML [17] provide continuations as first-class data objects. Continuations can be used to implement, at the source level, a number of interesting control features, such as loops, nonlocal exits, nonblind backtracking [22], nondeterministic computations [10, 14], and coroutines [7]. Source-level implementations of thread systems [9, 15, 21], especially in the area of graphical user interfaces (GUIs) [12, 13, 20, 23], are an important and rapidly growing application area for first-class continuations.

Continuations represent the remainder of a computation from a given point in the computation. When a continuation is invoked, control returns to the point in the program at which the continuation was captured. Traditional continuation mechanisms allow a continuation to be invoked multiple times. In our experience, however, most continuations are, in fact, invoked only once. In particular, this is true for continuations used to implement threads. This observation motivated us to develop *one-shot* continuations, continuations that can be invoked only once, and to investigate whether the copying costs associated with traditional *multi-shot* continuations could be avoided for one-shot continuations.

In this paper, we introduce one-shot continuations and explain how they interact with traditional multi-shot continuations. We describe an implementation of one-shot continuations that eliminates the copying overhead associated with multi-shot continuations. We present performance measurements that demonstrate that one-shot continuations are indeed more efficient than multi-shot continuations for certain applications, such as thread systems.

The remainder of this paper is organized as follows. Sections 2 and 3 describe one-shot continuations and the changes required to adapt our stack-based Scheme implementation to support them. Section 4 discusses the performance characteristics of our implementation. Finally, Section 5 summarizes the paper and compares our approach to related approaches.

2 One-shot continuations

Continuations in Scheme are procedural 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. *call/cc* must be passed a procedure p of one argument. *call/cc* obtains the current continuation and passes it to p. The continuation itself is represented by a procedure k. Each time k is applied to a value, it returns the value to the continuation of the *call/cc* application. This value is, in essence, the value of the application of *call/cc*. If p returns without invoking k, the value returned by the procedure is the value of the application of *call/cc*.

If control has not otherwise passed out of the call to *call/cc*, invoking the continuation merely results in a nonlocal exit with the specified value. If control has already passed out of the call to *call/cc*, the continuation can still be invoked, but the result is to restart the computation at a point from which the system has already returned.

The continuation of a procedure call is essentially the control stack of procedure activation records. If continuations were used only for nonlocal exits, as for C's `setjmp/longjmp`, then the essence of a continuation object would be a pointer into the control stack. Because continuations can outlive the context of their capture, however, continuation objects must have indefinite extent and a pointer into the stack is not sufficient. If this simple representation were used and control passed out of the context where the continuation was created, the stack might be overwritten by other procedure activation records, and the information required upon return to the continuation would be lost.

One-shot continuations are obtained with *call/1cc* and differ from multi-shot continuations only in that it is an error to invoke a one-shot continuation more than once. Note

*This material is based on work supported in part by the National Science Foundation under grant number CDA-9312614.

To appear in the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation.

that a continuation can be invoked either implicitly, by returning from the procedure passed to *call/cc* or *call/1cc*, or explicitly, by invoking the continuation procedure obtained from the *call/cc* or *call/1cc*.

One-shot continuations can be used in most contexts where multi-shot continuations are currently used, *e.g.*, to implement non-local exits, non-blind backtracking [22], and coroutines [7]. One-shot continuations can also be used to implement thread systems in user code.

One-shot continuations cannot be used to implement nondeterminism, as in Prolog [6], in which a continuation is invoked multiple times to yield additional values [10, 14]. In these sorts of applications, multi-shot continuations must still be used.

If a language supports both multi-shot and one-shot continuations, it is necessary handle cases in which programs use both varieties of continuation. For example, a Prolog interpreter might use multi-shot continuations to support nondeterminism while employing a thread system based on one-shot continuations at a lower level. One-shot continuations must be *promoted* to multi-shot status when they are captured as part of a multi-shot continuation. This allows programmers to build abstractions based on one-shot continuations that can be composed with abstractions based on multi-shot continuations in a consistent manner.

3 Implementation

A detailed description of our implementation of multi-shot continuations is described elsewhere [16]. In this section, we review the essential details and discuss the changes necessary to implement one-shot continuations.

3.1 Segmented stack model

In our model, the control stack is represented as a linked list of *stack segments*. 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 segment, including a pointer to the base of the segment, a pointer to the next stack record, the size of the segment, and the return address associated with the topmost frame of the continuation. (See Figure 1.)

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[1]. 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 the same stack is not used for 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.

[1] Our compiler actually passes the return address and the first few arguments in registers, where feasible [4]. Although this complicates the implementation only slightly, we assume a more straightforward model here to simplify our presentation.

No explicit links are formed between frames on the stack. Some 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 3.2). 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. If the return address 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 [16].

3.2 Continuation operations

A large initial stack segment and an associated current stack record are created in the heap at the beginning of a program run. Each time a multi-shot continuation is captured, the occupied portion of the current stack segment is sealed and the current stack record is converted into a continuation object. This involves setting the size field to the size of the occupied portion, *i.e.*, the relative position of the frame pointer within the segment, and storing the current return address in the return address field. (See Figure 2.) The return address in the current frame is replaced by the address of an underflow handler that implicitly invokes the captured continuation. A new stack record is allocated to serve as the current stack record. Its base is the address of the first 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 multi-shot continuation, therefore, does not entail copying the stack, but it does shorten the current stack segment, which eventually results in stack overflow and the allocation of a new stack segment. The initial stack segment is made large to reduce the frequency of stack overflows for programs that create many continuations and for deeply recursive programs.

Capturing a one-shot continuation is similar to capturing a multi-shot continuation except that the entire current segment is encapsulated in the continuation and a fresh stack segment is allocated to replace the current stack segment (see Figure 2). Two size fields are required to record both the total size of the segment and the current size. The cur-

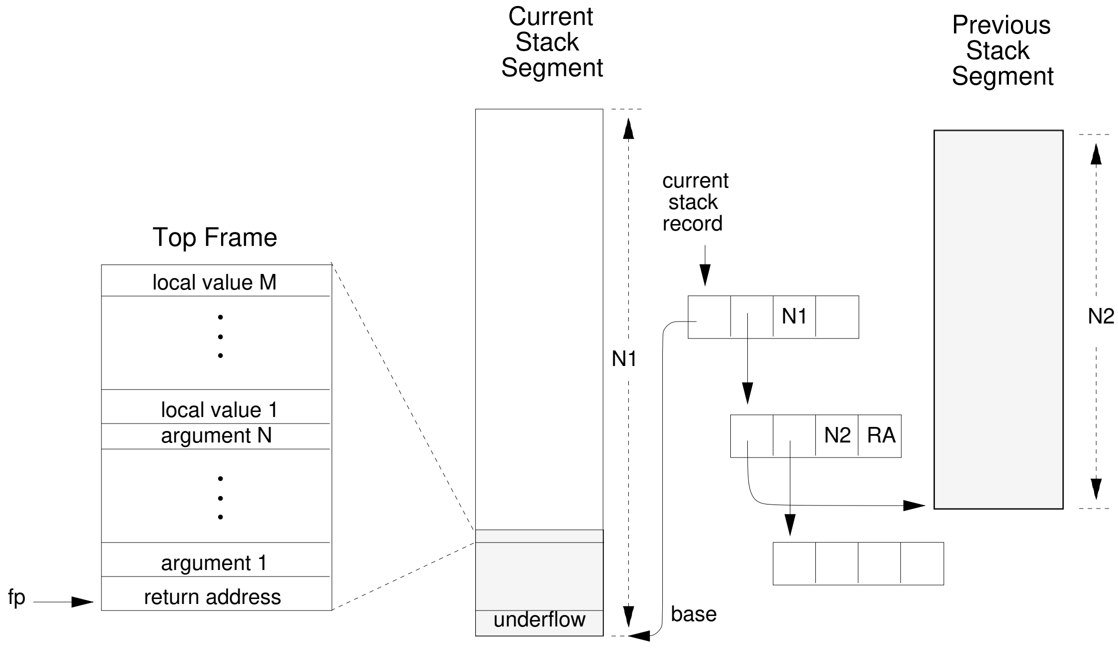

Figure 1. The segmented stack model is a simple generalization of the traditional stack model. A logical stack is represented as a list of stack segments that are linked together using stack records. A stack record contains a pointer to the base of a stack segment, a pointer to the stack record for the next segment in the stack, the size of the segment, and the return address that is displaced by the underflow handler address. Although the picture shows a frame containing a return address and arguments, the first several arguments and the return address may actually be passed in registers.

rent size is the size of the occupied portion of the stack, *i.e.*, the relative position of the frame pointer within the stack segment.

Since the total size of a multi-shot continuation is precisely the size of the occupied portion, the two size fields are always equal in multi-shot continuations. Our system uses this fact to distinguish between one-shot and multi-shot continuations.

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.

Invoking continuations is more complex. For multi-shot continuations, 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 (see Figure 3). If the current stack segment is not large enough, a new one is allocated. Since the size of a saved stack segment can be large, the cost of continuation invocation would be bounded only by this large amount, if the whole segment were copied at once. This is 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 the 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. See [16] for additional details on splitting.

For one-shot continuations, there is no need to copy the saved stack segment since the continuation will be invoked only once. Thus, the current stack segment is discarded and control is simply returned to the saved stack segment. The base, link, and size fields of the continuation are used to reinitialize the current stack record and to reset the frame pointer. (See Figure 4.) Since the contents of the stack segment are not copied, there is no need to split the segment, regardless of its size. To allow subsequent attempts to invoke the continuation to be detected and prevented, it is marked "shot" by setting the size and current size to -1.

A typical application involving one-shot continuations obtains the current continuation using *call/1cc*, saves the continuation, and invokes a previously saved one-shot continuation. In this scenario, a new stack segment is allocated by *call/1cc* and almost immediately discarded when the saved one-shot continuation is invoked. This rapid allocation and release of stack segments can overtax the storage management system. A solution to this problem is to use some type of stack segment cache, which can be represented as a simple internally linked free list of stack segments. When a one-shot continuation is invoked, the current stack segment is added to the cache, and when *call/1cc* requires a new segment, the stack cache is checked before a new segment is allocated. The stacks in this cache can be discarded by the storage manager during garbage collection. Without a stack segment cache, we found that many programs written in terms of *call/1cc* were unacceptably slow, much slower than the equivalent programs written in terms of *call/cc*.

Stack overflow can be treated as an implicit *call/cc* [16], although since overflow occurs when the current stack segment has insufficient space, a new current segment must be allocated. Improper overflow handling can result in *bouncing*, in which a program makes a call that overflows the stack, underflows immediately by returning from the call,

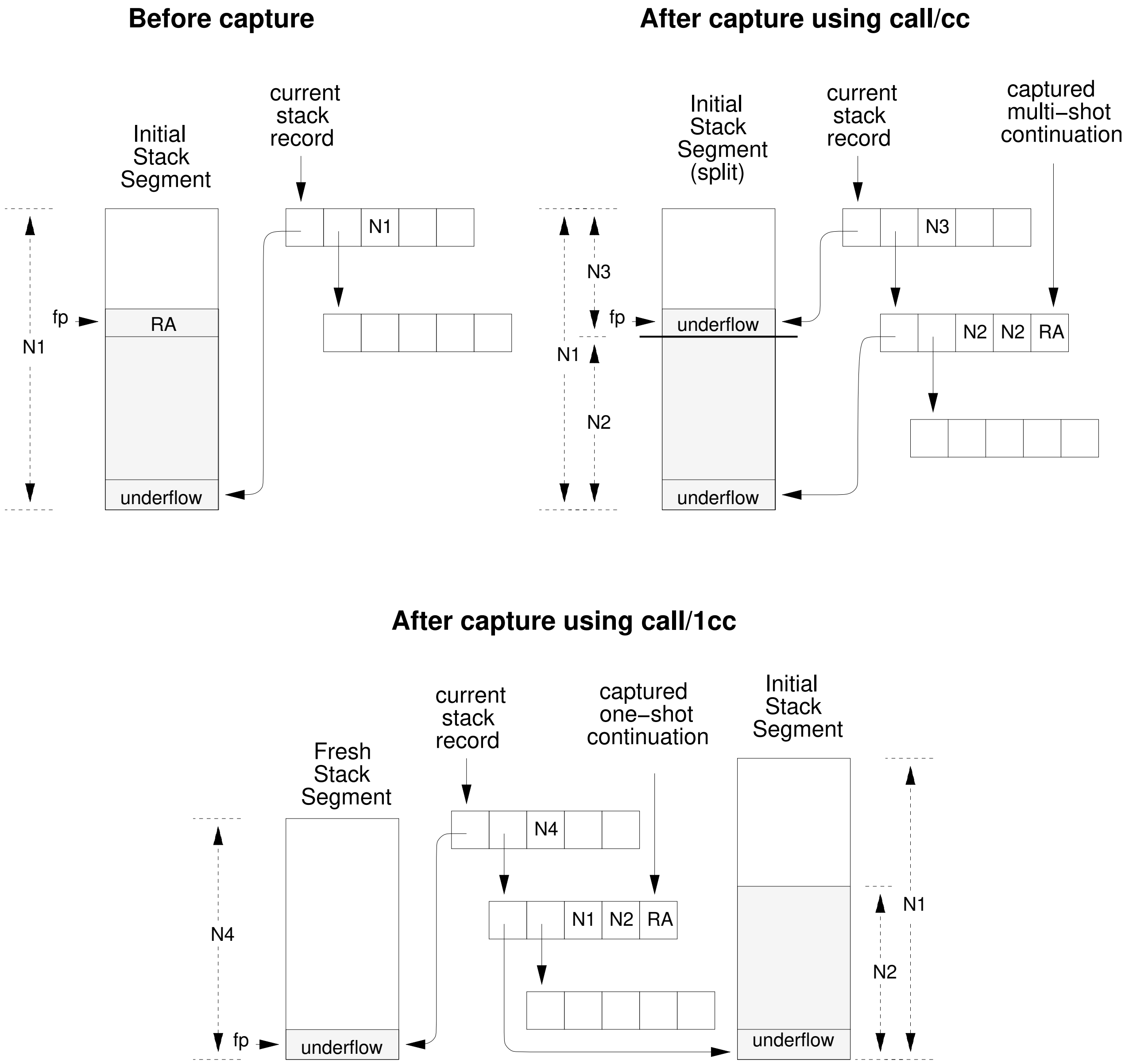

Figure 2. Stack records in the new system have an additional field that specifies how much of the stack segment is currently in use. For multi-shot continuations the current size is always equal to the size of the segment. For one-shot continuations they always differ. *call/cc* creates a multi-shot continuation by sealing off the stack segment at the current frame and using the remainder of the stack as the new stack segment. *call/1cc* encapsulates the entire current stack segment in the captured continuation and allocates a fresh stack segment. Both operations also allocate and initialize a new current stack record.

immediately makes another call that overflows the stack, and so on. Treating overflow as an implicit *call/cc* avoids the bouncing problem since the entire newly allocated stack must be refilled before another overflow can occur.

Stack overflow can be treated as an implicit *call/1cc* instead. Doing so naively, however, can cause bouncing, since an immediate underflow switches back to the saved (full) stack, at which point a call is guaranteed to cause a stack overflow. This problem can be reduced by copying up several frames on overflow from the current stack segment into the newly allocated stack segment. The overflow continuation thus includes the portion of the stack segment that is not copied. A similar mechanism is used in the spineless tagless G-machine [19] to solve essentially the same problem [18]. We found that, without the hysteresis provided by this mechanism, there was a noticeable performance degradation in certain programs. With this mechanism in place, deeply recursive programs run faster than with overflow treated as an implicit *call/cc*, due to the decrease in copying.

3.3 Promotion

As discussed in Section 2, it is necessary to promote one-shot continuations in the continuation of a call to *call/cc* to multi-shot continuations. Promotion of a one-shot continuation is trivial given our representations of one-shot and

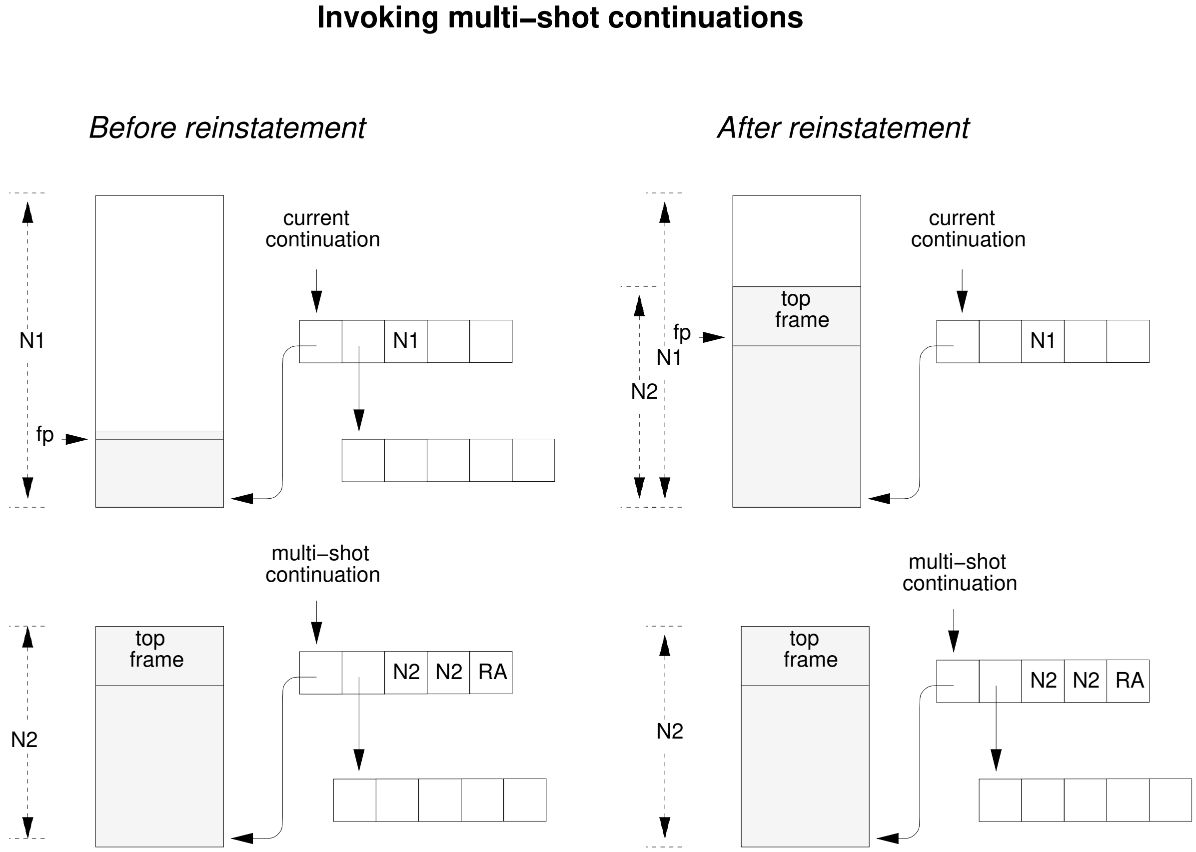

Figure 3. When a multi-shot continuation is invoked, the contents of the saved stack segment is copied into the current stack segment. If the size of the saved stack segment is greater than the copy-bound, the segment is first split into two segments [16]. If the current stack segment is not large enough to hold the contents of the saved stack segment, a new stack segment is allocated.

multi-shot continuations: promotion simply sets the size of a one-shot continuation equal to its current size. Since the current continuation at any point may include a chain of one-shot continuations, it is necessary to iterate down the chain until a multi-shot continuation is found, resetting all one-shot continuations along the way. It is not necessary to iterate beyond a multi-shot continuation in the chain because the operation that created the multi-shot continuation would have reset all one-shot continuations below it in the chain. Although the linear traversal involved in the promotion of one-shot continuations captured by *call/cc* means that there is no hard bound on the speed of *call/cc* operations, there is no quadratic time complexity problem because a one-shot continuation can be promoted only once. One solution that would allow *call/cc* to run in bounded time (which we have not implemented) is to share a boxed flag among all one-shot continuations in a chain. All of the one-shot continuations could then be promoted simultaneously by simply setting the shared flag.

Even if the system did not promote one-shot continuations created explicitly by the program, it would still be obligated to promote one-shot continuations created implicitly as the result of a stack overflow.

3.4 Stack segment fragmentation

Internal fragmentation can result from the inclusion of unoccupied memory in the stack segments of one-shot continuations. With a default stack size of 16KB, 100 threads created using *call/1cc* occupy 1.6MB of memory. Unless each of the threads is deeply recursive, most of this storage is wasted. Multi-shot continuations, in contrast, do not cause fragmentation because saved segments contain no unused storage.

One way to decrease fragmentation is to use a small default stack size. This would penalize deeply recursive programs and programs that create many multi-shot continuations, however, because they would overflow more often. Another solution, which we are currently using, is to limit the amount of unused memory encapsulated in the stack record of a one-shot continuation by sealing the current stack segment at a fixed displacement above above the occupied portion of the stack. We then use the remaining portion of the stack segment as the new current stack segment rather than allocating a fresh stack (possibly from the stack cache).

4 Performance

We have added one-shot continuations to the implementation of *Chez Scheme* while maintaining support for *call/cc*, *dynamic-wind* [8], and multiple return values [3].

To determine the benefit of one-shot continuations for programs in which *call/cc* can be replaced by *call/1cc*, we modified the call-intensive *tak* program [11] so that each call captures and invokes a continuation, either with *call/cc* or with *call/1cc*. The version using *call/1cc* is 13% faster than the version using *call/cc* and allocates 23% less memory.

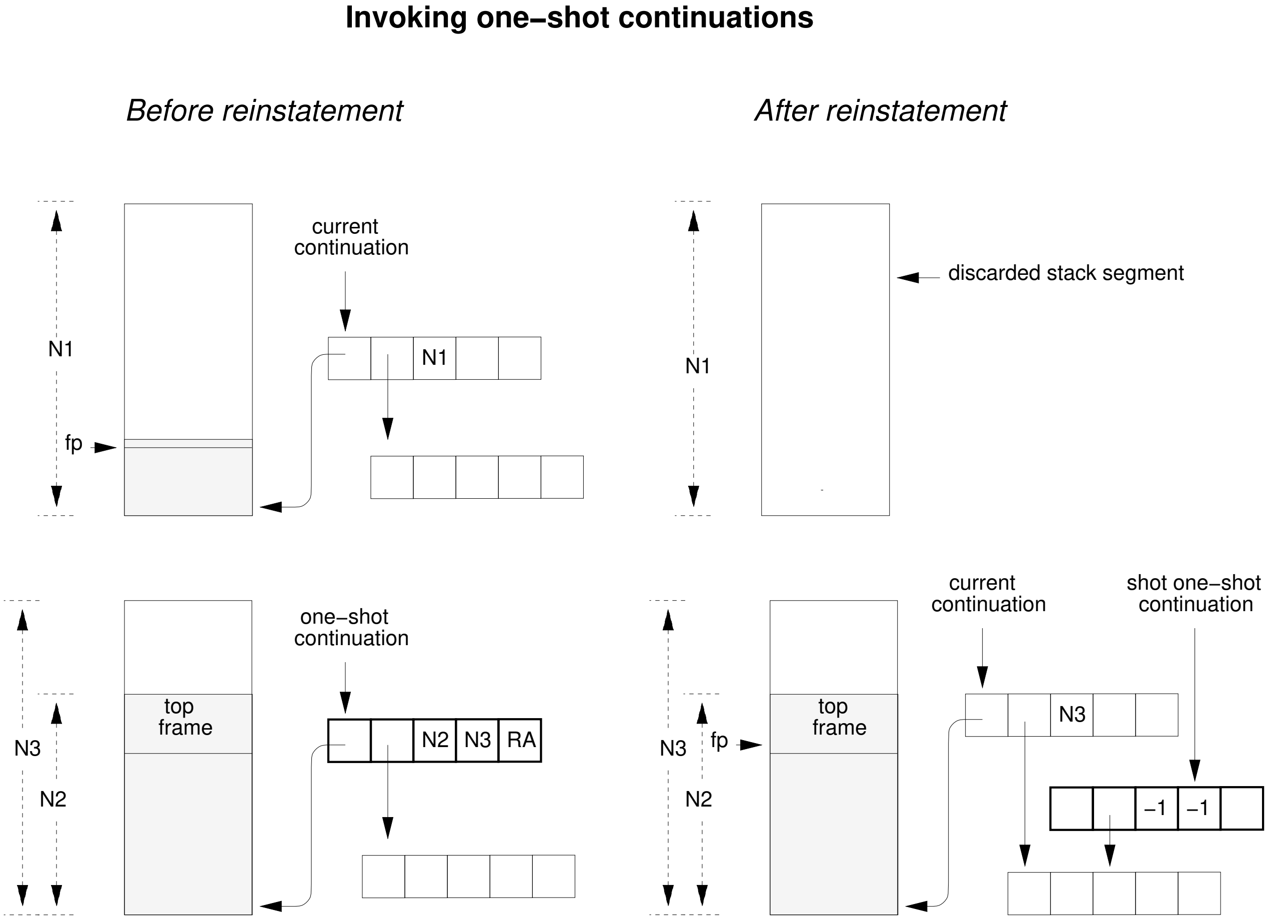

Figure 4. When a one-shot continuation is invoked, the current stack segment is discarded, and the contents of the stack record for the one-shot continuation is used to update the current stack record. The size and current size of the one-shot continuation are then set to -1 to indicate that the continuation has been shot.

We also compared the performance of three versions of a thread system, one implemented using *call/cc*, one using *call/1cc*, and one using continuation-passing style (CPS). The continuation-passing style version simulates a heap-based representation of control, although it does not include the additional overhead for supporting *dynamic-wind*. Figure 5 compares run times for different context-switch frequencies for 10, 100, and 1000 active threads. The figure shows that *call/1cc* threads are consistently faster than *call/cc* threads, although the advantage is minimal for low context switch frequencies, as is to be expected. The figure also shows that, although the CPS version is faster than either of the other versions for extremely rapid context switches (more often than once every four procedure calls), it loses its advantage quickly as the number of procedure calls between context switches increases.

To determine the benefit derived from using one-shot continuations rather than multi-shot continuations for handling overflows, we compared the performance of a program that repeatedly recurs deeply (one million calls) while doing very little work between calls. In this extreme case overflow handling using one-shot continuations is 300% faster and allocates much less. In fact, after the first recursion, the one-shot version always finds fresh stack segments in the stack cache and so allocates very little additional memory. For real programs the difference is typically much less dramatic.

5 Conclusions

In this paper, we have introduced one-shot continuations, shown how they interact with traditional multi-shot continuations, and described a stack-based implementation of control that handles both multi-shot and one-shot continuations, including the promotion of one-shot continuations to multi-shot continuations when captured by a multi-shot continuation. We have described how the copying overhead incurred by multi-shot continuations can be eliminated for one-shot continuations.

Our performance analysis shows that two important classes of applications benefit from the use of one-shot continuations: deeply recursive programs and continuation-intensive applications such as thread systems with rapid context switching. In other cases, the copying overhead associated with multi-shot continuations appears to be insignificant. For example, one-shot continuations are only a few percent faster than multi-shot continuations in our thread benchmarks when context switches occur less frequently than once every 128 function calls.

Others have proposed using a heap-based representation

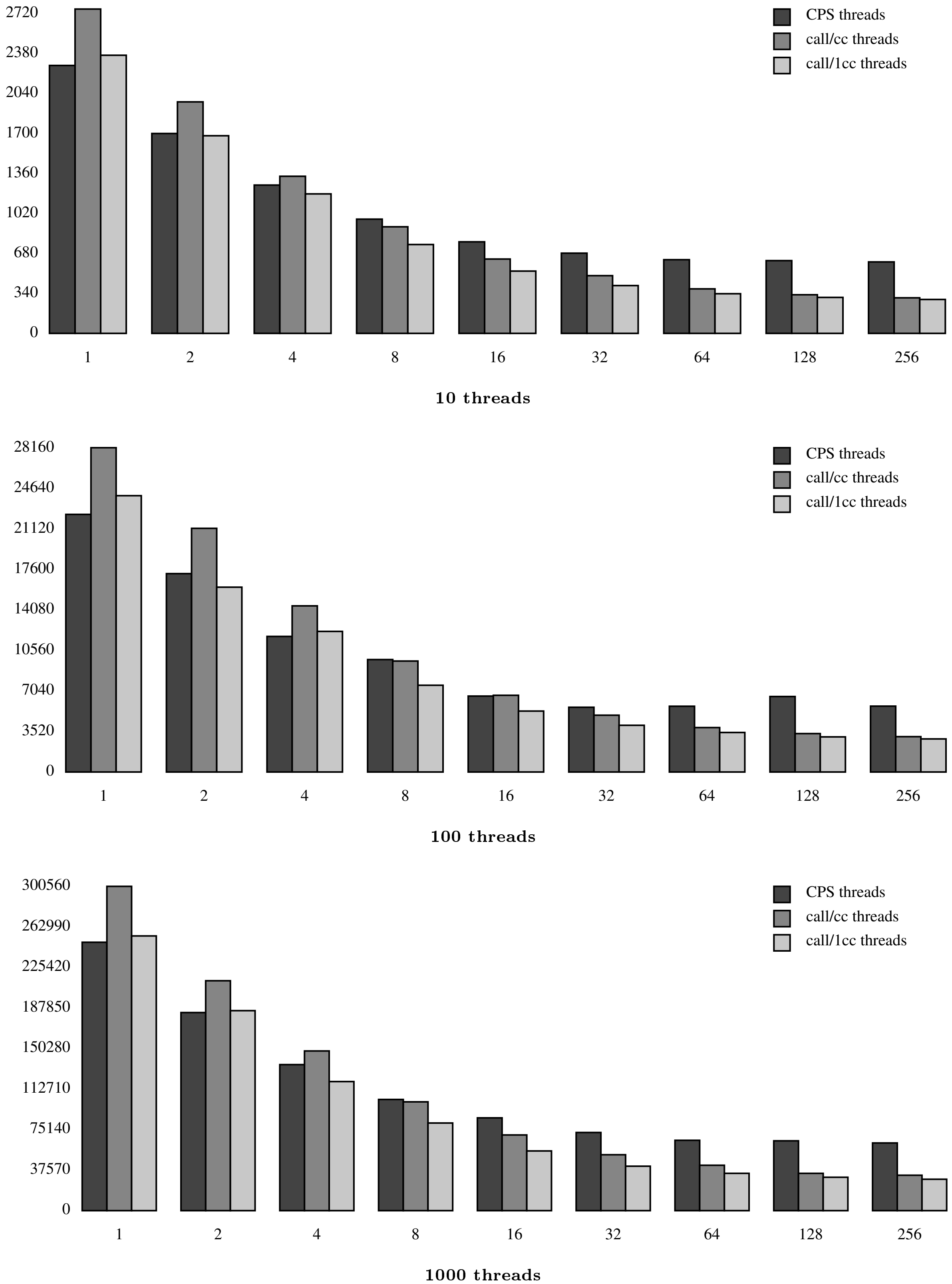

Figure 5. These graphs show the relative performance of CPS, *call/cc*, and *call/1cc* versions of a thread system. Each run involved 10, 100, or 1000 active threads each computing the 20th Fibonacci number using the simple doubly recursive algorithm. Context switch frequency is shown varying from once every procedure call through once every 512 procedure calls. The performance was measured on a 96MB DEC Alpha 3000/600 running OSF/1. Times are shown in milliseconds.

of control, in which control stacks are represented as linked lists of frames rather than as true stacks. This approach is used by Appel and MacQueen [1] in a compiler for ML [17]. Appel and Shao [2] have compared their heap-based approach to a simulated stack-based approach and found them to have approximately the same per-frame overhead (an average of 7.5 and 7.4 instructions per frame, respectively) when potential negative cache effects associated with the heap-based approach are factored out. They attribute 3.4 instructions of the 7.4 instruction per frame overhead for stack-based implementations to closure creation costs. We have analyzed a large set of benchmark programs and have found, however, that the overhead in our system is actually much lower, on the order of 0.1 instructions per frame. In particular, they report a closure creation cost overhead of 5.75 instructions per frame for the Boyer benchmark, whereas our implementation allocates no closures at all. One possible explanation for this discrepancy is that Appel and Shao's simulated stack model uses a CPS-based compiler with stack-allocated continuations that inhibits the sharing among frames that derives automatically from a direct-style compiler employing a true stack-based representation of control. For example, a variable live across several calls must be copied into each continuation frame in their model; the same variable in our stack-based implementation can remain in the same stack location across all calls without incurring any overhead.

In spite of the performance advantages of a stack-based approach for most programs, it is tempting to conclude that a heap-based approach is a better choice for thread systems implemented using continuations because of the copying overhead incurred by multi-shot continuations and the relatively more complex implementation of stack-based continuations. We have shown, however, that a simple heap-based implementation is superior only if context switches occur more frequently than once every eight procedure calls (about once every four for *call/1cc*). While various compiler optimizations can be introduced to make the heap model more competitive, the complexity of these optimizations more than compensates for the difference in representation complexity without fully eliminating the performance differential.

With stack overflow treated as an implicit call to *call/1cc*, deeply recursive programs that do not use multi-shot continuations do not incur any copying overhead on stack underflow. Since this can result in significant savings, a stack-based implementation should use one-shot continuations internally to handle stack overflow, even if the implementation does not otherwise support *call/1cc*. The same mechanism is also applicable in the context of thread packages for languages such as C and Fortran that do not support first-class continuations. In this context, the segmented stack permits the use of large numbers of threads while supporting arbitrary recursion, since it is possible to allocate threads with relatively small stacks that grow on demand. This fact was observed by Peyton-Jones and Salkild in the context of the Spineless Tagless G-machine [19].

References

[1] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In *Proceedings of the Third International Symposium on Programming Language Implementation and Logic Programming*, pages 1–13, August 1991.

[2] Andrew W. Appel and Zhong Shao. An empirical and analytical study of stack vs. heap cost for languages with closures. Technical Report CS-TR-450-94, Princeton University, March 1994. Revised version to appear in Journal of Functional Programming.

[3] J. Michael Ashley and R. Kent Dybvig. An efficient implementation of multiple return values in scheme. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pages 140–149, July 1994.

[4] Robert G. Burger, Oscar Waddell, and R. Kent Dybvig. Register allocation using lazy saves, eager restores, and greedy shuffling. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 130–138, June 1995.

[5] William Clinger, Jonathan A. Rees, et al. The revised^4 report on the algorithmic language Scheme. *LISP Pointers*, 4(3), 1991.

[6] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog*. Springer-Verlag, second edition, 1984.

[7] Christopher T. Haynes Daniel P. Friedman and Mitchell Wand. Obtaining coroutines with continuations. *Computer Languages*, 11(3/4):143–153, 1986.

[8] R. Kent Dybvig. *The Scheme Programming Language*. Prentice Hall, second edition, 1996.

[9] R. Kent Dybvig and Robert Hieb. Engines from continuations. *Computer Languages*, 14(2):109–123, 1989.

[10] Mathias Felleisen. Transliterating Prolog into Scheme. Technical Report 182, Indiana University, October 1985.

[11] Richard P. Gabriel. *Performance and Evaluation of LISP Systems*. MIT Press, Cambridge, MA, 1985.

[12] Emden R. Gansner and John H. Reppy. A multithreaded higher-order user interface toolkit. In *User Interface Software*. John Wiley & Sons Ltd, 1993.

[13] Paul Haahr. Montage: Breaking windowing into small pieces. In *USENIX Summer Conference*, pages 289–297, Anaheim, June 1990.

[14] Christopher T. Haynes. Logic continuations. *LISP Pointers*, pages 157–176, 1987.

[15] Christopher T. Haynes and Daniel P. Friedman. Abstracting timed preemption with engines. *Computer Languages*, 12(2):109–121, 1987.

[16] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 66–77, June 1990.

[17] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, 1990.

[18] Simon L. Peyton-Jones. *private communication*, December 1991.

[19] Simon L. Peyton-Jones and Jon Salkild. The spineless tagless G-machine. In *Proceedings of the Fourth Conference on Functional Programming and Computer Architecture*, pages 184–201, September 1989.

[20] R. Pike. A concurrent window system. *Computing Systems*, 2(2):133–153, 89.

[21] John H. Reppy. CML: A higher-order concurrent language. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 293–305, Toronto, June 1991.

[22] Gerald J. Sussman and Guy L. Steele Jr. Scheme: An interpreter for extended lambda calculus. AI Memo 349, Massachusetts Institute of Technology Artificial Intelligence Lab, 1975.

[23] Oscar Waddell. *The Scheme Widget Library User's Manual*. Indiana University, Bloomington, Indiana, 1995.