, , 1–42 ()

Implementation Strategies for First-Class Continuations

WILLIAM D. CLINGER, ANNE H. HARTHEIMER, AND ERIC M. OST will@ccs.neu.edu
Lightship Software
Corresponding author: *William D. Clinger*
College of Computer Science
Northeastern University
360 Huntington Avenue
Boston, MA 02115, USA

Abstract. Scheme and Smalltalk continuations may have unlimited extent. This means that a purely stack-based implementation of continuations, as suffices for most languages, is inadequate. We review several implementation strategies for continuations and compare their performance using instruction counts for the normal case and continuation-intensive synthetic benchmarks for other scenarios, including coroutines and multitasking. All of the strategies constrain a compiler in some way, resulting in indirect costs that are hard to measure directly. We use related measurements on a set of benchmarks to calculate upper bounds for these indirect costs.

Keywords: continuations, stacks, heap allocation, coroutines, multitasking, Scheme, Smalltalk

This is a revised and greatly expanded version of a paper that was presented at the 1988 ACM Conference on Lisp and Functional Programming [13].

1. Introduction

A *continuation* is the abstract concept represented by the control stack, or dynamic chain of activation records, in a typical programming-language implementation. In languages such as Scheme and Smalltalk-80, continuations (known as *contexts* in Smalltalk-80) may become first-class objects with unlimited extent (lifetime), whereas continuations have only dynamic (nested) extent in most languages [23, 28, 29, 38]. In Scheme, first-class continuations allow multiple returns from a single procedure call. This implies that a conventional stack-based implementation of recursive procedure calls, in which continuation frames are allocated and deallocated in last-in, first-out manner by adjusting a stack pointer, is inadequate [6, 21].

Lightweight threads raise many of the same issues, and can be implemented very easily using first-class continuations [25]. Thus strategies for implementing first-class continuations may also be relevant for languages that do not support first-class continuations directly but do provide support for concurrent or pseudo-concurrent threads.

We use Scheme for our examples. In Scheme, the mechanism that allows continuations to outlive their more usual dynamic extent is the `call-with-current-continuation` procedure. One possible implementation of this procedure, in terms of low-level procedures `creg-get` and `creg-set!`, is shown below. This code assumes that the `creg-get` procedure converts the implicit continuation that was passed to `call-with-current-continuation` into some kind of Scheme object with unlimited extent, and that the `creg-set!` procedure takes such an object and installs it as the continuation for the currently executing procedure, overriding the previous implicit continuation. The operation performed by `creg-get` is called a *capture*. The operation performed by `creg-set!` is called a *throw*. The procedure that is passed to `f` is called an *escape procedure*, because a call to the escape procedure will allow control to bypass the implicit continuation.

```
(define (call-with-current-continuation f)
  (let ((k (creg-get)))
    (f (lambda (v)
         (creg-set! k)
         v))))
```

The simplest implementation strategy for first-class continuations is to allocate storage for each continuation frame (activation record) on a heap and to reclaim that storage through garbage collection or reference counting [23]. With this strategy, which we call the *gc strategy*, `creg-get` can just return the contents of a continuation register (which is often called the dynamic link, stack pointer, or frame pointer), and `creg-set!` can just store its argument into that register.

Several other implementation strategies for continuations with unlimited extent have been described [5, 8, 13, 18, 19, 26, 33, 35, 36, 44]. Most of these strategies improve upon the gc strategy by making ordinary procedure calls faster, but make captures and/or throws slower because the `creg-get` and `creg-set!` operations involve a transformation of representation and/or copying of data. Since procedure calls are more common than captures and throws, this tradeoff is worthwhile.

In this paper we compare these implementation strategies and evaluate their performance. Our analysis distinguishes between the relatively compiler-independent direct costs of a strategy and its more compiler-dependent indirect costs. This distinction is explained in Section 2. Section 3 describes three scenarios for the uses of continuations that are most common in real programs.

Section 4 reviews ten implementation strategies. Section 5 summarizes their costs for a normal procedure call and return. Their indirect costs are reviewed and bounded in Section 6. Our measurements show that the indirect cost of copying and sharing is very much smaller than was reported by Andrew Appel and Zhong Shao [2]. We discuss this discrepancy in Section 7. Section 8 gives two examples to show that all strategies incur some indirect cost from the mere existence of first-class continuations within a programming language.

Section 9 confirms some of the analytic results of Section 5 for two continuation-intensive benchmarks. Section 10 quantifies the continuation-intensive behavior that results from using first-class continuations to implement lightweight multitasking, and reviews the performance of several strategies for this important application of continuations.

Appel and Shao claimed that the stack-based strategies we describe are difficult to implement, citing seven specific problems [2]. Section 11 explains how those problems can be resolved by an implementation that uses the incremental stack/heap strategy, which we recommend.

2. Direct and indirect costs

To evaluate the performance of an implementation strategy for continuations, it is convenient to distinguish between the direct costs and the indirect costs of the strategy. We define the *direct cost* of a strategy as the number of machine instructions required to create a single continuation frame, link it to the continuation being extended, and to dispose of the frame. The direct cost so defined is relatively independent of the compiler.

The fraction of execution time that can be attributed to this direct cost varies greatly depending on both the program being executed and the compiler that was used to translate it into machine code. As an exceedingly rough rule of thumb, it is reasonable to assume that one continuation frame is created for every hundred machine instructions that are executed [2]. Under this assumption, each unit of direct cost will correspond to about 1% of the overall execution time.

We define the *indirect costs* of a strategy as any costs that are not part of its direct cost. All implementation strategies have indirect costs. Most of the indirect costs are very compiler-dependent or hardware-dependent, which makes them harder to estimate than the direct costs. Our discussion of compiler dependencies will rely upon certain technical concepts defined below.

Within the code for a procedure, there are syntactic positions at which the continuation is necessarily equivalent to the continuation that was passed to the procedure. These positions are known as *tail positions*, and can be defined by induction on the syntax of a programming language, as has been done for Scheme [16, 29]. A *tail call* is a procedure call that occurs in a tail position. For a tail call, it is syntactically obvious that the continuation that is passed to the procedure being called is semantically equivalent to the continuation that was passed to the procedure performing the call, which implies that the compiler does not have to create a new continuation frame for the tail call. A *non-tail call* is a procedure call that is not a tail call.

Depending on the compiler, a continuation frame may be created whenever:

1. a procedure is entered;
2. a non-leaf procedure is entered, where a leaf procedure is defined as a procedure whose body does not contain any procedure calls [46];
3. a non-leaf procedure is entered, where a leaf procedure is defined as a procedure whose body may contain tail calls, but does not contain any non-tail calls [9];
4. a procedure call is performed;
5. a non-tail call is performed [1];
6. a conditional branch is resolved along a path that ensures that a non-tail call will be executed [9];

7. a conditional branch in tail position is resolved along a path for which a non-tail call is possible, and deferring the creation of a frame until a later conditional branch is resolved would lose an opportunity to reuse a single frame for multiple non-tail calls [14].

Compilers that use rule 1, 2, 3, 6, or 7 often reuse a single continuation frame for multiple non-tail calls. Compilers that use rule 6 or 7 tend to create fewer continuation frames than compilers that use one of the first five rules.

Each of the implementation strategies that we will describe has exactly one of the following indirect costs:

- A strategy may make it difficult for the compiler to reuse a single continuation frame for multiple non-tail calls.
- A strategy may make it difficult for the compiler to allocate storage for mutable variables within a continuation frame.

From the calculations in Section 6, it appears that the indirect cost of not reusing continuation frames is larger than the cost of not allocating mutable variables within a frame, at least for languages like Scheme and Standard ML. Several other indirect costs are discussed in Sections 4 and 6.

3. Three common scenarios

This section describes three scenarios that abstract the most common behaviors of a program with respect to first-class continuations. These scenarios illustrate most of the important differences between the performance of different strategies for implementing first-class continuations. Furthermore most programs lie somewhere along the spectrum spanned by these scenarios.

3.1. No first-class continuations

Many Scheme programs do not use `call-with-current-continuation` at all. It would be nice if they didn't have to pay for the mere existence of `call-with-current-continuation` within the language.

We will say that an implementation strategy has *zero overhead* if, on programs that do not use first-class continuations at all, the strategy's direct cost is no greater than the direct cost of an ordinary stack-based implementation of a language that does not support first-class continuations.

To make this more concrete, we will take the Motorola PowerPC as representative of modern computer architectures [37]; Appendix 1 of this paper reviews the PowerPC instructions that are used below. Consider the following simple (therefore nonstandard) PowerPC assembly code for a non-tail call to a procedure `foo` that takes no arguments:

```
addi    cont,cont,-8 // creation of continuation frame
mflr    r0           // common instructions
```

```
stw     r0,4(cont)   // common instructions
bl      foo          // common instructions
lwz     r0,4(cont)   // common instructions
mtlr    r0           // common instructions
addi    cont,cont,8  // disposal of continuation frame
```

This code consists of one add-immediate instruction to create a continuation frame, another to dispose of that frame, and five instructions in between that save the link register (return address) within the newly created frame, branch and link to `foo`, and restore the link register. These five instructions are common to all of the PowerPC examples in this paper, and will henceforth be abbreviated to a comment. A zero-overhead strategy should require only two PowerPC instructions beyond the five common instructions, and neither of those two instructions should touch memory.

3.2. Non-local exits

Perhaps the most common use of escape procedures in Scheme is for non-local exits from a computation when an exceptional condition is encountered. An escape procedure that implements a non-local exit is seldom called more than once; indeed most such escape procedures are not called at all. For this *non-local-exit scenario* we will assume also that only a few escape procedures are created, because programs that create many escape procedures are likely to match the recapture scenario considered below.

For the non-local-exit scenario, we desire an implementation strategy that incurs no extra overhead even after a continuation has been captured. For real-time systems, we would also want to have some bound on the time required to perform a capture or throw.

3.3. The recapture scenario

Olivier Danvy suggested that captures tend to occur in clusters [18]. That is, the same continuation, once captured (by `call-with-current-continuation`), is likely to be captured again—either an enclosing continuation will be recaptured, or some subpart of it will be recaptured. In fact, recapturing a previously captured continuation frame can be more common than returning through a captured frame. We call this the *recapture scenario*.

For the recapture scenario, we desire an implementation strategy that does not require much additional time or space to recapture a previously captured continuation.

Not all programs that use `call-with-current-continuation` match the recapture scenario. In particular, many of the programs that use escape procedures only for non-local exits do not match the recapture scenario. The recapture scenario is nonetheless fairly common. For example, some programs create an escape procedure for each iteration of a loop, recapturing the loop's continuation each time.

When continuations are used to implement lightweight multitasking, as in Concurrent ML [39], it is possible for a program to match the recapture scenario even though it does not call `call-with-current-continuation` explicitly. As explained in Section 10, we found this to be true of many programs written using MacScheme+Toolsmith [41].

4. Implementation strategies

This section describes several implementation strategies. For most of the strategies we provide typical PowerPC code for a non-tail procedure call. We also state whether the strategy has zero overhead for such calls, compared to a conventional stack-based implementation of a language that does not support first-class continuations. We then describe the actions required to implement a capture (`creg-get`) and a throw (`creg-set!`) using the strategy.

Next we characterize the strategy's performance for each of our three scenarios: programs that don't use first-class continuations, programs that create only a few escape procedures and use them only for non-local exits, and the recapture scenario in which many continuation frames are captured repeatedly.

For each strategy, we mention at least one of its primary indirect costs. We also name an implementation that uses the strategy, including the earliest that we know.

In a few cases we cite an important variation of the strategy.

4.1. The garbage-collection strategy

The simplest strategy for ensuring that continuations have unlimited extent is to allocate them in the heap and to rely on garbage collection or reference counting to recover their storage [23, 27]. We call this the *gc strategy.*

Standard ML of New Jersey has demonstrated that the gc strategy is practical when used with a fast generational garbage collector [1]. The gc strategy is not a zero-overhead strategy, however. Here is typical PowerPC code for a non-tail call using the gc strategy:

```
stw     cont,4(avail)    // frame pointer
addi    r0,r0,12         // creation (init header)
stw     r0,-4(cont)      // creation
addi    cont,avail,4     // creation (allocate)
addi    avail,avail,12   // creation
cmpw    avail,limit      // creation (test for overflow)
bge     overflow         // creation
                         // five common instructions
lwz     cont,0(cont)     // frame pointer
```

The first and last instructions save and restore the continuation register (often called the dynamic link, stack pointer, or frame pointer). The second and third instructions initialize a header word with the total size of the block of storage being allocated, for the benefit of the garbage collector. The fourth and fifth instructions allocate a 12-byte block of storage, and place a pointer to that block into the continuation register. The compare-word and conditional-branch instructions test for heap overflow. This code uses 6 instructions to create the continuation frame, and another two instructions to save and to restore the frame pointer.

Appel and Shao have pointed out that the header word can be eliminated by using a garbage collector that can map return addresses to header words using an auxiliary table [2]. This saves two instructions.

The last three of the instructions that are labelled `creation` can sometimes be combined with another heap allocation that occurs within the same basic block. For the seven benchmarks considered by Appel and Shao, the percentage of non-tail calls for which this was *not* possible ranged from 46% to 86%, with an average of 69% [2]. If we assume that this average is representative of all programs, then the cost of those three instructions is effectively reduced to $3 \times .69 \doteq 2.1$ instructions.

The number of machine instructions required to create a continuation frame therefore ranges from 3.1 to 6. Two more instructions are required to save and to restore the frame pointer. The storage occupied by a frame is reclaimed through garbage collection, whose cost is considered in Section 5.

A capture (`creg-get`) is accomplished by a single instruction that copies the continuation register to another register. A throw (`creg-set!`) is accomplished by a single instruction that loads the continuation register with the new continuation.

The gc strategy is optimized for programs in which every continuation frame is captured. On synthetic benchmarks that capture all continuations, the gc strategy therefore performs better than the other strategies described below. Few real programs capture all continuations, however, so the gc strategy may not perform as well as a zero-overhead strategy even on programs that match the non-local-exit or recapture scenarios.

The most important indirect cost of the gc strategy is that the compiler must allocate a separate continuation frame for each non-tail call, unless the compiler can prove that the continuation will not be captured during the non-tail call. If it were captured, then reusing the frame would overwrite a part of the captured continuation.

The gc strategy also suffers more cache misses than the other strategies described in this paper.

The gc strategy has been used by many interpreters, including the first versions of Smalltalk and MacScheme, and was used for compiled code by Standard ML of New Jersey (SML/NJ), which supports first-class continuations as an extension to Standard ML [1]. Recent versions of SML/NJ have addressed the high costs of the gc strategy by allocating many continuation frames in callee-save registers instead of the heap [42].

4.2. The spaghetti strategy

The spaghetti stack used in Interlisp is a variation of the gc strategy [7]. The spaghetti stack is in effect a separate heap in which storage is reclaimed by reference counting rather than garbage collection. Though complex, the spaghetti stack was at one time more efficient than using a gc strategy with a nongenerational garbage collector, because the spaghetti stack's storage management is optimized to support procedure call, return, and a host of related operations. In the normal case, when all frames have dynamic extent, the spaghetti stack behaves as a conventional stack.

Unfortunately, the normal case also includes the overhead of testing the reference counts and branching conditional on the new counts. Here is PowerPC code for the normal case; this code assumes that the reference count is kept in the high-order bits of the header word:

```
stw     cont,4(avail)     // frame pointer
addi    r0,r0,12          // creation (init header)
stw     r0,-4(cont)       // creation
addi    cont,avail,4      // creation (allocate)
addi    avail,avail,12    // creation
cmpw    avail,limit       // creation (test for overflow)
bge     overflow          // creation
                          // five common instructions
lwz     r0,-4(cont)       // disposal
cmpwi   r0,12             // disposal
bgt     unusual_case      // disposal
addi    avail,avail,-12   // disposal
lwz     cont,0(cont)      // frame pointer
```

The allocation of a continuation frame cannot be combined with other heap allocation because the spaghetti stack is a separate heap. Furthermore the four instructions used here to dispose of a frame explicitly are likely to cost at least as much as using precise generational garbage collection to recover the frame's storage. When a fast garbage collector is available, the spaghetti strategy is probably slower than the gc strategy.

Captures and throws require updating the reference counts. It therefore appears that the gc strategy should always perform better than the spaghetti strategy.

Spaghetti stacks were designed to support dynamically scoped languages such as Interlisp. A macaroni stack is a variation of a spaghetti stack that is designed to support statically scoped languages [43].

4.3. The heap strategy

The lifetime of a continuation frame created for a procedure call normally ends when the called procedure returns. The only exception is for continuation frames that have been captured. This suggests the *heap strategy*, in which a one-bit reference count in each frame indicates whether the frame has been captured. Continuation frames are allocated in a garbage-collected heap, as in the gc strategy, but a free list of uncaptured frames is also used. When a frame is needed by a procedure call, it is taken from the free list unless the free list is empty. If the free list is empty, then the frame is allocated from the heap. When a frame is returned through, it is linked onto the free list if its reference count indicates that it has not been captured. Otherwise it is left for the garbage collector to reclaim.

The heap strategy is not a zero-overhead strategy. The following PowerPC code assumes that an empty free list can be detected via a memory protection exception during the execution of the first instruction. If exceptions must be avoided, then two more machine instructions would be required. This code also assumes that a separate word is used to link frames into the free list, and that the sign bit of the saved frame pointer is used as the one-bit reference count.

```
stw     cont,0(free)     // frame pointer
or      cont,free,free   // creation
lwz     free,link(free)  // creation
                         // five common instructions
stw     free,link(cont)  // disposal
or      free,cont,cont   // disposal
lwz     cont,0(cont)     // frame pointer
cmpwi   cont,0           // disposal
blt     unusual_case     // disposal
```

Capturing a continuation involves setting the reference count of every frame in the continuation to indicate that it has been captured. Throwing to a continuation involves nothing more than storing the continuation in the continuation register.

For programs that don't use first-class continuations, the performance of the heap strategy is roughly comparable to that of the gc strategy. The non-local-exit scenario degrades performance just a bit, because every return through a captured continuation causes a few extra instructions to be executed. The heap strategy performs well for the recapture scenario, because recapturing a continuation takes constant time.

The heap strategy is most practical if all continuation frames are the same size; otherwise multiple free lists may be required. This is an indirect cost of the heap strategy.

Like the gc strategy, the heap strategy makes it difficult to reuse a continuation frame for multiple non-tail calls. This is another indirect cost.

Danvy has described an incremental variation of the heap strategy that avoids the need to mark all continuation frames when a continuation is captured, at the cost of a few extra instructions for each call and return [18]. Eliot Moss has described another variation of the heap strategy that uses a single register to serve as both the continuation register and the free list pointer [36].

4.4. The stack strategy

A last-in, first-out stack discipline works well except for continuations that are captured. This suggests the *stack strategy*, in which the active continuation is represented as a contiguous stack in an area of storage we call the *stack cache*. Non-tail calls push continuation frames onto this stack cache, and returns pop frames from the stack cache, just as in an ordinary stack-based implementation.

When a continuation is captured, however, a copy of the entire stack cache is made and stored in the heap. When a continuation is thrown to, the stack cache is cleared and the continuation is copied back into the stack cache. A first-class continuation thus resides in the heap, but is cached in the stack cache whenever it is the active continuation.

The stack strategy is a zero-overhead strategy. It can use standard calling sequences, making procedure calls and returns just as fast as for languages that restrict continuations to have dynamic extent:

```
addi    cont,cont,-8     // creation
                         // five common instructions
addi    cont,cont,8      // disposal
```

Capturing, recapturing, and throwing to a continuation take time proportional to the size of the continuation.

The stack strategy performs well for the non-local-exit scenario, where the overhead consists of the time required to copy the few continuations that are captured. The stack strategy performs poorly for the recapture scenario, because it repeatedly copies the same continuation from the stack cache to the heap. This can increase the asymptotic storage space required by an implementation [1, 16], which must be counted as an indirect cost of the stack strategy.

The stack strategy prevents a compiler from allocating storage for mutable variables within a continuation frame, because an assignment to the variable that alters one frame cannot affect other copies of it (unless the implementation does something complicated as in T or SOAR [31, 45]). Mutable variables must generally be allocated in registers or in the heap. This is another indirect cost of the stack strategy.

The SOAR implementation of Smalltalk-80 avoided this indirect cost by deferring the copying of a captured frame until it was returned through, or until a throw was performed. Then and only then was it copied into the heap. Pointers to the frame were updated using information maintained by the generation scavenging garbage collector and its hardware-assisted write barrier [45].

The stack strategy does allow continuation frames to be reused for multiple non-tail calls.

The stack strategy appears to have been invented by Drew McDermott [34]. Variations of this strategy have been used by most implementations of Scheme and Smalltalk-80 [5, 19, 40, 44, 45].

4.5. The chunked-stack strategy

PC Scheme used a chunked-stack strategy: By maintaining a small bound on the size of the stack cache, and copying portions of the stack cache into the heap or back again as the stack-cache overflows and underflows, PC Scheme reduced the worst-case latency of captures and throws [5].

The chunked-stack strategy works well with generational garbage collection because limiting the size of the stack cache limits the size of the root set that the garbage collector must scan on each garbage collection. The portion of the continuation that resides in the heap will be scanned only when its generation is collected. This improves the efficiency of generational garbage collection [12]. All of the strategies described below share this advantage. Other strategies can use a "watermark" for this purpose [2].

On the other hand, the chunked-stack strategy requires a stack cache that is large enough to accomodate the depth of typical recursions. Otherwise the copying of continuations on stack-cache overflows and underflows will degrade performance. This indirect cost is shared by the stack/heap, incremental stack/heap strategies, and Hieb-Dybvig-Bruggeman strategies described below, but to a lesser degree.

We regard the chunked-stack strategy as a zero-overhead strategy, because the cost of stack-cache overflows and underflows is usually negligible. See Section 6.4.

4.6. The stack/heap strategy

The *stack/heap strategy* is similar to the stack strategy. All continuation frames are allocated in the stack cache. When a continuation is captured, however, the contents of the stack cache are moved into the heap and the stack cache is cleared. Likewise when a continuation is thrown to, the new active continuation is left in the heap and the stack cache is cleared; this can be done in constant time.

Since the current continuation may reside in either the stack cache or in the heap, each procedure return must test to see whether the frame should be popped off the stack cache. If the stack cache is separated from the heap by some limit address, and that limit is kept in a register, then the stack/heap strategy requires two extra PowerPC instructions beyond those of a zero-overhead strategy:

```
        addi    cont,cont,-8      // creation
                                  // five common instructions
        cmpw    cont,limit        // disposal
        blt     captured          // disposal
        addi    cont,cont,8       // disposal
captured:
```

The stack/heap strategy makes throwing very fast, and recapturing a previously captured continuation is very fast also. It therefore works well for the non-local-exit and recapture scenarios.

With the stack/heap strategy, there is never more than one copy of a continuation frame, so it is all right for the compiler to allocate storage for mutable variables within a frame. A concomitant disadvantage of the stack/heap strategy is that it prevents the compiler from reusing a single continuation frame for multiple non-tail calls.

The stack/heap strategy has been used by Tektronix Smalltalk [47], BrouHaHa [35], MacScheme [41], and Luis Mateu's implementation of coroutines [33]. The implementation of BrouHaHa is notable because continuation frames in the heap are represented quite differently from frames in the stack cache. BrouHaHa uses two distinct interpreters, depending on whether the topmost continuation frame is in the heap or in the stack cache.

4.7. The incremental stack/heap strategy

The *incremental stack/heap strategy* is a variation of the stack/heap strategy: When returning through a continuation frame that isn't in the stack cache, a trap occurs and copies the frame into the stack cache. The trap can be implemented by maintaining a permanent continuation frame at the bottom of the stack cache. This frame's return address points to system code that copies one or more frames from the heap into the stack cache, and immediately returns through the first of those continuation frames.

The incremental stack/heap strategy is a zero-overhead strategy, with the same calling sequence as the stack strategy.

In the non-local-exit scenario, the incremental stack/heap strategy must copy the stack cache into the heap when an escape procedure is created, must copy continuation frames back into the stack cache when returning to a continuation that has been captured by an escape procedure, and must perform a second indirect jump on each such return. These costs, which are incurred only for frames that have been captured, are about twice those for the stack and stack/heap strategies.

In the recapture scenario, capturing a previously captured continuation frame is more common than returning through a captured frame. Thus the incremental stack/heap strategy performs much better than the stack strategy, and approaches the performance of the stack/heap strategy.

Since the incremental stack/heap strategy copies frames from the heap into the stack cache, mutable variables cannot be kept within a continuation frame.

The incremental stack/heap strategy is used in Scheme 48 and in Larceny [14].

4.8. The Hieb-Dybvig-Bruggeman strategy

Chez Scheme uses a variation of the incremental stack/heap strategy due to Hieb, Dybvig, and Bruggeman [26]. This variation uses multiple *stack segments* that are allocated in the heap. The stack segment that contains the current continuation serves as the stack cache.

When the stack cache overflows, a new stack cache is allocated and linked to the old one. Stack-cache underflow is handled by an underflow frame, as in the incremental stack/heap strategy.

When a continuation is captured, the stack cache is split by allocating a small data structure that points to the current continuation frame within the stack cache. This data structure represents the captured continuation. The unused portion of the stack cache becomes the new stack cache, and an underflow frame is installed at its base.

A throw is handled as in the incremental stack/heap strategy: the current stack cache is cleared, and some number of continuation frames are copied into it. The underflow frame at the base of the stack cache is linked to the portion of the new continuation that was not copied.

The Hieb-Dybvig-Bruggeman strategy is a zero-overhead strategy. As with the stack strategy and the incremental stack/heap strategy, mutable variables generally cannot be allocated within a continuation frame, but continuation frames may be reused for multiple non-tail calls.

One of the main advantages of the Hieb-Dybvig-Bruggeman strategy over the incremental stack/heap strategy is that it performs about twice as well for captured frames in the non-local-exit scenario. For this scenario, the Hieb-Dybvig-Bruggeman strategy performs the same amount of copying as the stack/heap strategy, although the copying occurs at the time of return instead of the time of capture.

For the recapture scenario, the Hieb-Dybvig-Bruggeman strategy performs slightly better than the incremental stack/heap strategy because it avoids copying on the first capture.

4.9. One-shot continuations

The Hieb-Dybvig-Bruggeman strategy can be extended to provide more efficient support for escape procedures that cannot be called more than once [8]. Although these *one-shot continuations* are not first-class continuations, they suffice for the non-local-exit scenario and for multitasking.

Many programming languages, including C++ and Java, provide exception facilities and/or threads that rely on one-shot continuations. Strategies for implementing one-shot continuations are generally outside the scope of this paper. Our purpose in this section is to describe one of several techniques that allow one-shot continuations to coexist with first-class continuations.

One-shot continuations are captured by a `call1cc` procedure whose semantics are the same as the semantics of `call-with-current-continuation`, except that `call1cc` creates a one-shot escape procedure that cannot be called more than once.

A call to `call1cc` is implemented in much the same way as a stack-cache overflow with the Hieb-Dybvig-Bruggeman strategy. Instead of splitting the current stack cache as for a capture, a new stack cache is allocated and linked to the old one. The one-shot continuation is represented as a small data structure that points into the old stack cache.

A call to a one-shot escape procedure discards the current stack cache and reinstates the stack cache that is pointed to by the one-shot continuation. It also marks the one-shot continuation so that any subsequent attempt to call the one-shot escape procedure will signal an error.

Calls to `call-with-current-continuation` can capture part or all of a one-shot continuation, coercing it into a first-class continuation. This is implemented by a process much like performing a capture using the heap strategy, but marking stack segments instead of individual continuation frames. Otherwise captures are implemented as in the Hieb-Dybvig-Bruggeman strategy.

When coroutines or multitasking are implemented using one-shot continuations, the rapid discarding of stack caches is likely to cause excessive garbage collection, leading to performance that is worse than would be obtained from first-class continuations. This problem can be overcome by maintaining a free list of stack caches [8].

The performance of one-shot continuations for multitasking is discussed in Section 10.

One-shot continuations were implemented in Chez Scheme, in combination with the Hieb-Dybvig-Bruggeman strategy for first-class continuations [8].

4.10. Mateu's coroutines

Mateu implemented a variation of the gc strategy in which the two youngest generations of a generational garbage collector are devoted entirely to continuation frames, and reported that this improved the performance of the gc strategy for coroutining applications [33]. We can speculate on the reasons for this improvement:

- The youngest generation acts as a stack cache, which can be made small enough to fit within the hardware's primary data cache.
- Continuation frames have a higher mortality rate than other objects, so dedicating the youngest generations entirely to frames decreases the number of objects that must be copied into an older generation.
- The roots for a frames-only garbage collection consist of the register that holds the current continuation, and the data structures that represent the current set of threads; in Mateu's implementation these roots were linked together. A more general garbage collection would probably use a larger set of roots and a more cumbersome mechanism for the remembered set.

Mateu also implemented a variation in which frames that had not been captured were deleted explicitly, as in the stack/heap strategy, and found that this improved performance.

The performance of Mateu's implementation for coroutines in discussed in Section 10.

5. Summary of costs

Figure 1 summarizes the direct costs of several implementation strategies, and gives upper bounds for the known indirect costs of each strategy, as calculated in Section 6. The total cost of each strategy is then shown as the sum of its direct and indirect costs, in terms of the number of machine instructions per continuation frame created plus the number of cycles lost per continuation frame due to cache misses. Our Figure 1 can be compared with Appel and Shao's Figure 1, but the gc strategy of our Figure 1 corresponds to their "Heap" column, whereas our heap strategy corresponds to their "Quasi-Stack" column [2].

Most of the direct costs in Figure 1 are obtained by counting machine instructions. In several cases we report a range of costs. For the heap strategy, the cost of creating

Strategy:	gc	heap	stack	stack/ heap	incremental stack/heap, HDB
Creation	3.1	2.0 – 4.0	1.0	1.0 – 3.0	1.0 – 3.0
Frame pointers	2.0	2.0	0.0	0.0	0.0
Disposal (pop)	1.4	4.0	1.0	3.0	1.0
Direct Cost	**6.5**	**8.0** – 10.0	**2.0**	**4.0** – 6.0	**2.0** – 4.0
Indirect Costs	< 10.7	< 6.2	< 1.4	< 4.1	< 1.5
Total Cost	**6.5** – 17.2	**8.0** – 16.2	**2.0** – 3.4	**4.0** – 10.1	**2.0** – 5.5

Figure 1. The cost of procedure call and return for six implementation strategies, in instructions per continuation frame plus cycles lost due to cache misses. The indirect costs, from Figure 3, should be regarded as loose upper bounds. See Section 5.

a continuation frame is two instructions if an empty free list can be detected by a hardware exception, but one or two additional instructions may be required if hardware exceptions are not used. Similarly the cost of creating a stack frame depends upon whether stack-cache overflow is detected by hardware or by the code that creates the frame. Most systems detect stack overflow in hardware, but the stack/heap, incremental stack/heap, and Hieb-Dybvig-Bruggeman strategies are able to recover from a stack-cache overflow by copying frames into the heap. In systems that do not provide precise exceptions, this recovery may not be practical unless the stack-cache overflow is detected by software, which will require one or two additional instructions.

For the seven benchmarks used by Appel and Shao, the average cost to dispose of a continuation frame using garbage collection was 1.4 instructions [2]. The cost was less than 1.4 instructions for five of the benchmarks, slightly higher for one, and was 7.9 instructions for the outlier. This cost must be regarded as a little more uncertain than the other direct costs, and is very sensitive to details of the garbage collector in any case.

Figure 2 uses asymptotic notation to express

- the cost of capturing a continuation for the first time,
- the cost of recapturing it a second time,
- the cost of performing a throw, and
- the total marginal costs that are associated with performing all of the returns through all of the frames of a continuation that has been thrown to.

These costs are expressed in terms of the size N of the continuation that is contained within the stack cache, and the number of frames M that are contained within the continuation being thrown to. For the heap and stack/heap strategies the

Strategy:	gc	heap	stack	stack/ heap	incremental stack/heap	HDB
First capture	$\Theta(1)$	$\Theta(M)$	$\Theta(N)$	$\Theta(N)$	$\Theta(N)$	$\Theta(1)$
Recapture	$\Theta(1)$	$\Theta(1)$	$\Theta(N)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Throw	$\Theta(1)$	$\Theta(1)$	$\Theta(N)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Returns	$\Theta(1)$	$O(N)$	$\Theta(1)$	$O(N)$	$\Theta(N)$	$\Theta(N)$

Figure 2. The cost of captures, throws, and the total extra cost associated with procedure returns following a throw, for six implementation strategies. M is the number of frames that are contained within the continuation being thrown to, and N is the size of the continuation that is contained within the stack cache. See Section 5.

Strategy: Indirect cost:	gc	heap	stack	stack/ heap	incremental stack/heap, HDB
cache write misses	< 5.1	0	0	0	0
cache read misses	< 1.0	0	0	0	0
not reusing frames	< 4.6	< 5.4	0	< 3.2	0
heap variables	0	0	< 0.6	0	< 0.6
stack-cache overflow	0	0	0	< 0.1	< 0.1
copying and sharing	0	< 0.8	< 0.8	< 0.8	< 0.8
Indirect cost	< 10.7	< 6.2	< 1.4	< 4.1	< 1.5

Figure 3. Upper bounds for the indirect costs of five implementation strategies averaged over our ten benchmarks, in instructions per continuation frame. The true indirect costs are likely to be considerably less than the upper bounds shown here.

predominant cost of returning through a previously captured frame is the cost of allocating new heap storage, which is in $\Omega(M)$ and $O(N)$ but is not necessarily in $\Theta(M)$ or $\Theta(N)$. These costs are corroborated by our benchmark results in Section 9.

6. Indirect costs

All of the implementation strategies have indirect costs that are hard to estimate because they do not show up in the machine instructions that are used to perform a procedure call, and vary greatly depending upon the program being executed and the compiler that was used to compile it.

benchmark	lines of code	brief description
`smlboyer`	1003	Standard ML version of a Gabriel benchmark
`nboyer`	767	term rewriting and tautology checking
`conform`	616	subtype inference
`dynamic`	2343	Henglein's dynamic type inference
`graphs`	644	enumeration of directed graphs
`lattice`	219	enumeration of maps between lattices
`nbody`	1428	inverse-square law simulation
`nucleic2`	4748	determination of nucleic acids' spatial structure
`puzzle`	171	Pascal-like search; a Gabriel benchmark
`quicksort`	58	array quicksort of 30000 integers

Figure 4. Benchmarks used to bound the indirect costs.

We rely on Appel and Shao's estimates for the cost of cache misses associated with the gc strategy. For the other indirect costs we derive upper bounds from a set of instrumented benchmarks. These upper bounds are summarized in Figure 3.

Figure 4 lists the set of ten benchmarks that we used to bound the major indirect costs. The `smlboyer` benchmark was selected because Appel and Shao reported that it exhibited an unusually high indirect cost [2]. The `nboyer` benchmark was selected because it is essentially a scalable and less language-dependent version of the `smlboyer` benchmark, but its indirect costs were expected to be lower because `nboyer` is a first-order program. The other eight benchmarks were selected because various researchers have been using them to evaluate compilers and garbage collectors, and they represent a mix of functional and imperative programming styles.

The `nboyer` and `graphs` benchmarks take an integer parameter that determines the problem size. `nboyer0` solves the same problem that is solved by `smlboyer`, while `nboyer3` solves a problem that is large enough to give current machines more of a workout. Although the data we report for `nboyer0` give some insight into the differences between `nboyer` and `smlboyer`, we ignored the data for `nboyer0` when computing averages, and used the data for `nboyer3` instead.

Clinger modified the Twobit compiler used in Larceny v0.35 to generate code that collects dynamic counts for

- the number of continuation frames created,
- the number of words in those continuation frames,
- the number of non-tail calls,
- the number of procedure activations (at entry to a procedure),
- the number of local variables that are allocated on the heap instead of in registers or in the stack,

benchmark	frames created	frame words	non-tail calls	activations
`smlboyer`	654947	2548043	1965858	4081799
`nboyer0`	440863	1321706	624528	953856
`nboyer3`	18996440	56917766	30113464	42789559
`conform`	926350	2678079	2113284	4243491
`dynamic`	184415	577850	462452	681736
`graphs7`	95608529	403473195	64266948	249517273
`lattice`	69358517	278882250	110739112	152973602
`nbody`	3408040	18100938	6091223	17041183
`nucleic2`	227181	1033553	542594	863905
`puzzle`	19460	71932	25476	845883
`quicksort`	656884	4173904	782025	1224395

Figure 5. Data used to bound the indirect costs.

- the number of references to heap-allocated local variables,
- the number of assignments to heap-allocated local variables,
- the number of assignments to heap-allocated local variables that can take the fast path through a generational garbage collector's write barrier,
- the number of closures created for lambda expressions,
- and the number of data words in those closures, excluding the code pointer and static link. (A compiler switch forced Twobit to create flat closures, so the static link was always null.)

These statistics were collected for the code of each benchmark, but not for code that is part of the standard Scheme library; hence the number of procedure activations is less than the total number of procedure calls, which we did not measure. The data are shown in Figures 5 and 6.

Figure 5 shows that the average size of a continuation frame ranges from 2.9 words for the `conform` benchmark to 6.4 words for the `quicksort` benchmark, with an average over all ten benchmarks of 4.1 words per frame. This is slightly less than the 4.2 words per frame that were observed by Appel and Shao for a different set of benchmarks [2].

6.1. *Cost of cache misses*

The gc strategy is the only strategy for which cache misses will be common during normal procedure calls and returns. The cost of cache misses is difficult to compute, so we have relied on the analyses and simulations reported by Appel and Shao [2].

If the youngest generation of a generational garbage collector fits within the primary cache, then the writes that are performed by the gc strategy should almost always hit the cache. In many current implementations, however, the youngest generation is substantially larger than the primary cache, and the writes that are performed by the gc strategy almost always miss the cache. On some machines, write misses do not cost anything. On other machines, the cost of a write miss can be very high. For example, if the size of a cache line is 8 words, the average size of a continuation frame is 4.1 words (as calculated above), and the penalty for a write miss is 10 cycles, then the average cost per frame due to write misses is $10 \times (4.1/8) \doteq 5.1$ cycles.

The gc strategy also tends to incur more read misses than the other strategies. For a 16 kilobyte direct-mapped, write-allocate cache, Appel and Shao used simulations to estimate that these additional read misses cost the gc strategy about 1.0 machine cycles per frame when compared to the other strategies [2]. Appel and Shao also analyzed three very regular procedure calling behaviors: tail recursion, deep recursion, and the tree recursion that is performed while solving the Towers of Hanoi puzzle. For these behaviors, the gc strategy had essentially the same number of cache read misses as the other strategies.

When continuations are used to implement multitasking or coroutines, context switches are likely to cause more cache read misses with the gc strategy than with other strategies. The primary caches of current machines can hold the active portion of the continuations for perhaps 100 threads (see Figure 8), so procedure returns that follow a context switch are likely to hit the cache. With the gc strategy, however, an inactive thread's continuation will be flushed from the cache to make way for frames that are allocated on the heap by other threads. This effect appears to be visible with the `cofib` benchmark described in Section 10.2.

6.2. Cost of not reusing frames

The gc, spaghetti, heap, and stack/heap strategies do not copy continuation frames that have been captured. Although this has some advantages for continuation-intensive programs, it also means that the compiler cannot modify a continuation frame in order to reuse it for multiple non-tail calls. This is an indirect cost of those four strategies, and may well be the largest indirect cost of those strategies, but the size of this cost is very compiler-dependent.

Many compilers for CISC architectures use push instructions to allocate a separate continuation frame for every non-tail call. With a compiler that wouldn't reuse frames anyway, the indirect cost of an implementation strategy that prevents the compiler from reusing frames is zero.

On RISC machines, which may not even have push instructions, it is common for compilers to allocate a single frame at entry to a procedure, and to reuse that frame for all non-tail calls that occur within the procedure. Although changing such a compiler to allocate a separate frame for every non-tail call would have a significant engineering cost, and might have performance costs due to increased code size and overhead for procedure parameters that do not fit into registers, it is quite possible that the overall performance cost would be negative: Allocating a separate frame for every non-tail call might improve performance, not make it worse.

The reason for this is that many of the dynamic procedure calls are tail calls. For example, the many `do` loops that appear within the Scheme version of the `puzzle` benchmark are syntactic sugar for tail recursion [29]. For our ten benchmarks, there are about as many tail calls as non-tail calls. Since half of the calls are non-tail calls, a compiler that allocates a frame for each non-tail call will allocate only half as many frames as a compiler that allocates a frame on entry to each procedure.

As an optimization, many compilers for RISC machines do not allocate a frame when entering a leaf procedure. This helps, but not enough: In typical Scheme code, less than one third of all procedure activations involve a leaf procedure, even when procedures that perform only tail calls are classified as leaf procedures [9].

For many compilers, therefore, the indirect cost of not reusing frames is zero or close to zero, and might even be negative.

For other compilers this indirect cost might be quite large, even for properly tail-recursive languages like Scheme. Chez Scheme uses an algorithm that eliminates more than half of the continuation frames that would be allocated by a compiler that allocates a frame on entry to every leaf procedure [9]. Twobit uses a different algorithm for the same purpose. For our ten benchmarks, as compiled by Twobit, there are about 1.54 non-tail calls per continuation frame. An implementation strategy that prevents Twobit from reusing frames would therefore increase the number of frames that are created by about 54%.

The indirect cost of not reusing frames includes not only the direct costs for these extra frames, but also the cost of initializing a frame with values that would already have been present within a reused frame; we have not measured this, but estimate that it averages about two instructions for each frame that could have been eliminated through reuse. For the heap strategy, whose direct costs are 8 instructions per frame, the indirect cost of preventing Twobit from reusing frames therefore appears to be about $0.54 \times (8 + 2) \doteq 5.4$ instructions.

In practice, the indirect cost of not reusing frames is unlikely to be this large. Any compiler for which this indirect cost is greater than zero would have to be modified to prevent it from reusing frames, and that modification would probably involve changing the compiler to use fewer caller-save registers and to rely more on callee-save registers [42]. These changes mitigate but do not eliminate the cost of not reusing frames, and add some costs of their own.

If the compiler is constrained to be safe for space complexity in the sense described by Appel, then the compiler must ensure that a reused frame contains no stale values that might increase the asymptotic space complexity of the program [1, 16]. The simplest way to remove a stale value is to overwrite it with a useful value, or with a useless zero if there are no more useful values that need to be saved within the frame. The cost of zeroing stale data when reusing a frame effectively reduces the indirect cost of not reusing frames. We have not measured this, but it seems clear that zeroing stale data within a frame is usually cheaper than disposing of the frame, creating a new one, and initializing the new frame.

benchmark	heap variables				closures	
	allocated	referenced	assigned		created	words
			fast	slow		
`smlboyer`	49231	84574	49230	1	49231	98462
`nboyer0`	0	174731	211128	4	0	0
`nboyer3`	0	7389410	8866287	198	0	0
`conform`	21446	46933	45928	3	37980	5596
`dynamic`	0	0	0	0	1043	1803
`graphs7`	0	0	0	0	59900055	329771002
`lattice`	7	21525292	120499	75	2295329	4745990
`nbody`	0	0	0	0	33571	162838
`nucleic2`	0	0	0	0	9000	27000
`puzzle`	2019	4083	2082	0	0	0
`quicksort`	0	0	0	0	1	0

Figure 6. More data used to bound the indirect costs. These data include the number of local variables that are allocated on the heap, the number of references to them, and the number of assignments to them that take the fast and the slow path through the write barrier. For the **nboyer** benchmark, the heap-allocated local variables were created prior to the timed portion of the benchmark.

In summary, the indirect cost associated with implementation strategies that prevent the compiler from reusing continuation frames for multiple non-tail calls could approach the direct cost, but is probably less in practice. With compilers that already create a separate frame for each non-tail call, or create a frame on entry to every procedure or leaf procedure, the indirect cost of not reusing frames is zero or perhaps even negative.

6.3. Cost of not allocating mutable variables in frames

The stack, chunked-stack, incremental stack/heap, and Hieb-Dybvig-Bruggeman strategies prevent a compiler from allocating mutable variables within a continuation frame. This appears to be the largest indirect cost associated with those strategies.

Many compilers for higher-order languages such as Scheme and Standard ML allocate all mutable variables on the heap, for reasons that have nothing to do with continuations. In Standard ML, for example, a mutable variable is itself a first-class object, and Scheme compilers often treat mutable variables as first-class objects because this simplifies important optimizations such as lambda lifting and closure conversion. Twobit does this, so an upper bound for the indirect cost of not allocating mutable variables in a continuation frame can be calculated from the number of heap variables that are allocated and the number of references and assignments to them. These data are shown in Figure 6. We assume that it takes 6 instructions to allocate heap storage for a variable, and 2 instructions to reclaim that storage through garbage collection; that each reference to a heap-allocated variable requires one more instruction than a reference to a variable that is allocated in a continuation frame; and that each assignment to a heap-allocated variable requires one extra instruction, plus the cost associated with the write barrier of a generational garbage collector. We assume that the write barrier costs 30 instructions in the worst case, but costs only 3 instructions when the variable resides within the youngest generation or is already a part of the garbage collector's remembered set. For our benchmarks, over 99.9% of the assignments take the 3-instruction path through the write barrier. The indirect cost of not allocating mutable variables in a frame is at most 2.3 instructions per frame for the `nboyer3` benchmark, 1.5 instructions per frame for the `puzzle` benchmark, 1.0 instructions per frame for the `smlboyer` benchmark, and less than one instruction per frame for the other seven benchmarks. The average over all ten benchmarks is 0.55 instructions per frame.

This calculation should be considered an upper bound because some of these variables occur free within a lambda expression for which a closure must be created, which means they would have to be allocated on the heap in any case. Some others, including almost all of the mutable local variables that are allocated on the heap by the `smlboyer` and `lattice` benchmarks, are artifacts of a `letrec` macro that expands into code that contains assignments [29]. Most such assignments are eliminated by Twobit's first-order closure analysis, but assignments that are introduced by the definition of a higher-order procedure are not eliminated.

Most of the side effects that occur in imperative languages can be removed by translating the program into a mostly-functional intermediate form such as static single assignment (SSA) form [3, 17]. A scalar variable that appears within an SSA form can be allocated in a register or continuation frame if its lifetime permits, regardless of the implementation strategy used for continuations. If a variable is not scalar, or if its lifetime extends beyond that of a continuation frame, then the compiler would probably have had to allocate the variable on the heap anyway.

When the compiler translates the phi nodes of an SSA form into machine language, it may have to insert instructions that copy values between registers and/or stack temporaries. A few of these instructions might be part of the indirect cost of not being able to allocate mutable variables within a continuation frame.

The compiler that we used eliminates a few assignments to scalar variables, but does not use SSA form for this purpose. For a more imperative language such as Smalltalk or Java, this optimization is likely to be more important than it is in Scheme or Standard ML.

In summary, the indirect cost that is associated with implementation strategies that prevent the compiler from allocating mutable variables within a continuation frame is less than one instruction per frame for languages like Scheme and Standard ML, and can probably be made small for other garbage-collected languages. For compilers that already allocate mutable variables on the heap for other reasons, this indirect cost is zero.

6.4. Cost of stack-cache overflow and underflow

The chunked-stack, stack/heap, incremental stack/heap, and Hieb-Dybvig-Bruggeman strategies suffer an indirect cost from stack-cache overflows and/or underflows. Stack-cache overflows are usually caused by a deep recursion, and result in frames being copied into the heap (except with the Hieb-Dybvig-Bruggeman strategy). Stack-cache underflows must copy frames back into the stack cache (except with the stack/heap strategy, which avoids copying by performing explicit tests on every return).

The average cost of copying a frame into the heap and then back into the stack cache is a little over 20 instructions per frame, which might degrade the performance of a deep recursion by as much as 20%. This compares favorably with the 180% degradation caused by overflow and underflow of an UltraSPARC's on-chip register windows during deep recursion.

Deep recursions account for a very small percentage of the continuation frames that are created by most programs, and recursions that are deeper than a few thousand non-tail calls are extremely rare. The frequency of stack-cache overflows and underflows can therefore be controlled by using a sufficiently large stack cache. In Scheme, a stack cache of 64 kilobytes will accomodate about three thousand continuation frames. This is enough to eliminate almost all stack-cache overflows and underflows, and helps to bound the size of the root set that a generational garbage collector must scan on every collection. With compilers that are not properly tail-recursive, or allocate arrays and other structured data within continuation frames, or allocate unreasonably large frames, a larger stack cache might be necessary.

Larceny v0.35 flushes the entire stack cache into the heap at every garbage collection, so its overhead for stack-cache overflows and underflows is unusually large. Even so, the cost of overflows and underflows averaged less than 0.04 instructions per frame for our benchmarks.

For programs that do not capture continuations, the indirect cost of stack-cache overflow and underflow is usually negligible. For the most continuation-intensive programs imaginable, Section 9 shows that stack-cache overflow and underflow account for less than 20% of the overall execution time when the stack/heap or incremental stack/heap strategies are used, but can double execution time when the chunked-stack strategy is used.

6.5. Cost of copying and sharing

In a higher-order language such as Scheme or Standard ML, the evaluation of a lambda expression generally involves allocating a data structure known as a *closure* that points to the code for the newly created procedure as well as to the values of its free variables.

Since these closures have unlimited extent, they cannot contain pointers into a stack cache. Any stack-allocated values that must be retained by a closure must be copied into the closure, or perhaps copied into a heap-allocated structure that can be shared by several closures. Appel and Shao called this the *cost of copying and sharing*.

We believe this cost is orthogonal to the strategy used to implement continuations. Shao and Appel observed that a compiler that uses the gc strategy can allocate a continuation frame that contains the values needed by a closure but no dynamic link to the rest of the continuation; that dynamic link can be kept in a callee-save register [42]. A closure can then point to this frame without also retaining the rest of the continuation. A compiler that uses a stack-based strategy can use exactly the same optimization, of course, regarding the heap-allocated structure as an environment frame instead of a continuation frame.

Hence the indirect cost of copying and sharing, as defined by Appel and Shao, is really the cost of not performing this optimization. This indirect cost is difficult to measure directly, but is clearly bounded by the total cost of initializing all of the closures that are created during execution of a program, and the cost of initializing closures is easy to measure.

We assume that it takes one load and one store instruction to initialize each data word of a closure. For nine of our ten benchmarks, the total cost of initializing flat closures is less than 0.4 instructions per continuation frame. For the `graphs7` benchmark, which makes very heavy use of higher-order procedures, the cost of initializing closures is 6.9 instructions per frame. The indirect cost of copying and sharing is less than this, and is probably far less, because many of these instructions store register variables (for which the load instruction is omitted), and many of these register variables have never been copied into a stack frame (so the store instruction should not count toward the cost of copying and sharing).

In summary, the indirect cost of copying and sharing is small for most programs, but might be significant for programs that create an unusually large number of closures. This cost appears to be orthogonal to the implementation strategy used for continuations, however, and should be associated instead with the implementation strategy used for environments and closures.

7. Appel and Shao's estimates for copying and sharing

Our upper bounds for the indirect cost of copying and sharing are considerably lower than the values reported by Appel and Shao [2]. In this section we explain why our measurements of this cost should take precedence. We also give three possible explanations for the difference between our upper bound and the cost measured by Appel and Shao.

For the Standard ML version of the `smlboyer` benchmark, Appel and Shao reported that the cost of copying and sharing was 5.75 instructions per frame. This was the highest cost reported for their seven benchmarks, which averaged 3.4 instructions per frame. For the Scheme version of the `smlboyer` benchmark, we measured an upper bound of 0.30 instructions per frame for this cost. For our set of ten benchmarks, which had only `smlboyer` in common with theirs, our upper bound for this cost averaged 0.8 instructions per frame, despite our inclusion of one extremely closure-intensive benchmark.

Our upper bounds for the indirect cost of copying and sharing were obtained by direct measurement of the number of words of storage that were allocated for all closures.

The values reported by Appel and Shao were computed indirectly. They constructed two implementations that allocated all frames on the heap using the gc strategy. One of these implementations used the optimization described in Section 6.5 to allow closures to point to certain continuation frames. The other implementation did not use this optimization, and used representations for continuation frames and closures that are more typical of stack-based compilers. For each benchmark they counted the number of instructions required by both implementations. They then assumed that the difference between the number of instructions executed represented the cost of copying and sharing.

Conversations with Appel and Shao have revealed three factors that would have inflated their measurement of the cost of copying and sharing for the `smlboyer` benchmark [4]. The first two factors would have inflated this measurement for all benchmarks.

The first factor is that the compiler that was used for the stack-like implementation did not reuse frames. This would have affected the reported cost of copying and sharing as follows. The cost of copying and sharing is incurred only when a lambda expression closes over a stack-allocated variable that, with the optimization described in Section 6.5, would have been allocated in a special heap-allocated record. With the optimization, that variable would not have been copied at all. Without the optimization, the cost of copying that variable into a closure legitimately counts toward the cost of copying and sharing, but the cost of copying that variable out of one frame into a register and then into another frame should count instead toward the cost of not reusing continuation frames. The method used by Appel and Shao could not distinguish these costs. It therefore appears likely that a large part of the cost they reported for copying and sharing is really part of the cost of not reusing continuation frames.

The second factor is that their implementation did not have any global or module-level environment, so creating a closure required copying all of the global and module-level variables that are used by the closure's code. This would have amplified the cost of copying and sharing. (It should be noted that using a global environment does not violate Appel and Shao's requirement that implementations be safe for space complexity, because the size of the global environment is fixed at the beginning of execution and the values of any global variables that become unreachable will henceforth remain fixed in size. The asymptotic space complexity is therefore unchanged by any failure to reclaim the storage occupied by global variables.)

The third factor is that Appel and Shao's implementation of exceptions created a closure for each exception handler. These closures are unnecessary, even if implementations are required to be safe for space complexity. The `smlboyer` benchmark creates only 49231 closures but establishes 129774 exception handlers, so unnecessary closures probably accounted for most of the cost of copying and sharing that Appel and Shao reported for this benchmark.

8. Observational equivalence

Programmers often assume that there is at most one return from a single dynamic procedure call, and some compilers assume this also, but first-class continuations imply that there can be more than one return from a single procedure call. First-class continuations therefore render certain optimizations unsafe. The fact that these optimizations cannot be performed must be counted as an indirect cost of all strategies for implementing first-class continuations. We have not attempted to measure this cost, but it appears to be small. In this section we give two examples.

Consider the following two-argument version of Scheme's `map` procedure:

```
(define (map2 f xs)              ; original, functional
  (if (null? xs)
      '()
      (let ((y (f (car xs))))
        (cons y (map2 f (cdr xs))))))
```

We will regard this code as the specification of `map2`, and will analyze two more definitions that a Scheme programmer might write in an attempt to create a more efficient but equivalent definition of `map2`.

If we ignore the continuation space that might be required by `f`, then the continuation space required by the original code is linear in the length of the list `xs`. If we convert this definition to iterative form as follows, then we obtain an observationally equivalent definition that requires only a fixed amount of continuation space.

```
(define (map2 f xs)              ; iterative, functional
  (do ((xs xs (cdr xs))
       (ys '() (cons (f (car xs)) ys)))
      ((null? xs)
       (reverse ys))))
```

Although this iterative version needs only a constant amount of storage for its continuations, it allocates twice as much list storage as was allocated by the original definition. To eliminate this extra allocation, a Scheme programmer might be tempted to rewrite the iterative version using `reverse!`, which is like `reverse` except that it reverses its argument by side-effecting the links of the list instead of allocating a new list:

```
(define (map2 f xs)              ; iterative, functional
  (do ((xs xs (cdr xs))
       (ys '() (cons (f (car xs)) ys)))
      ((null? xs)
       (reverse! ys))))
```

Unfortunately, this imperative version of `map2` is incorrect. With the original and iterative functional versions of `map2`, the expression shown below evaluates to `(0 1 4 9 100)`. With the iterative imperative version, it evaluates to `(16 9 100)`.

```
(let* ((k (lambda (x) x))
       (first-time? #t)
       (z (map2 (lambda (x)
                  (call-with-current-continuation
                   (lambda (return)
                     (set! k return)
                     (* x x))))
                '(0 1 2 3 4))))
  (if first-time?
      (begin (set! first-time? #f)
             (k 100))
      z))
```

This is not obvious, but it is explicable. With the iterative imperative version, the last continuation that is captured preserves an environment in which the R-value of `xs` is `(4)` and the R-value of `ys` is `(9 4 1 0)`. When `reverse!` is called, its side effects change the R-value of `ys` to `(9 16)`. (This is the most mysterious part of the explanation, but the mystery is attributable to side effects, not to continuations.) Passing 100 to the last captured continuation is equivalent to returning a second time from the call to `f` with argument 4, returning 100 instead of 16. The 100 is consed onto the R-value of `ys` and the resulting list `(100 9 16)` is passed to `reverse!`, which returns `(16 9 100)`.

This is surprising to most programmers, so we conclude that side effects interact badly with first-class continuations. By converting all of the code above to continuation-passing style (CPS) we can obtain the same surprising results without using first-class continuations at all, which shows that side effects interact badly with higher-order procedures in general. The CPS equivalents *appear* more complex, however, so programmers are not quite so surprised that the imperative CPS version behaves differently from the functional CPS versions. The real problem with first-class continuations is that they allow subtle and complex programs to be written more simply.

Whether first-class continuations are desirable is beyond the scope of this paper, but it is clear that first-class continuations break many of the observational equivalences that would hold in the absence of first-class continuations. This affects performance because the iterative imperative code for `map2` is likely to be a little faster than the iterative functional code, even in systems that use generational garbage collection. The fact that the imperative code cannot be used in place of the functional versions, because it is not equivalent to them, therefore counts as an indirect cost of first-class continuations.

In general, this cost arises whenever the existence of first-class continuations within a language prevents a programmer or a compiler from improving the performance of a program by introducing a side effect or by moving a side effect across a procedure call.

We will offer one more example of this indirect cost. Suppose `E1`, `E2`, ... are arbitrary expressions, and the variables `list` and `cons` are known to refer to Scheme's standard procedures for constructing a new list or pair. Then

Benchmark	gc strategy	stack strategy	stack/heap strategy	incremental stack/heap strategy
loop1		1.0		
loop2	5.9	12.1	5.9	6.9
tak		1.0		
ctak	10.9	24.2	11.2	11.5

Figure 7. Four strategies compared on two continuation-intensive benchmarks, `loop2` and `ctak`. Timings are relative to `loop1` and `tak`, which are related benchmarks that do not use continuations at all.

```
(list E1 E2 ... En)
```

is equivalent to

```
(let ((t1 E1)
      (t2 E2)
      ...
      (tn En))
  (cons t1 (cons t2 ... (cons tn '())...)))
```

in Scheme. Both of the above expressions would be equivalent to

```
(cons E1 (cons E2 ... (cons En '())...))
```

if not for the existence of first-class continuations in Scheme. In the first two expressions, all of the expressions `E1`, `E2`, ... are evaluated before any part of the list is allocated. In the last expression, some of the pairs that comprise the list may be allocated before `E1` has been evaluated. This can be observable if the list that results from these expressions is side effected, and `E1` is a procedure call that returns more than once. A Scheme compiler therefore cannot always translate the first or second expression into the third expression. The last expression can be evaluated from right to left using a fixed number of temporaries, whereas the first two expressions require n temporaries. This can affect performance when there aren't enough registers to keep all of the temporaries in registers. It therefore counts as an indirect cost of first-class continuations.

9. Continuation-intensive benchmarks

In 1988 we implemented the gc, stack, stack/heap, and incremental stack/heap strategies by modifying MacScheme+Toolsmith version 1.5 [41]. Non-tail-recursive procedure calls are slow in MacScheme because its calling conventions were designed for a byte code interpreter, where it is faster to have a byte code do potentially unnecessary work than it is to decode multiple byte codes. Since the operations associated with allocating and linking a continuation frame are so complex in MacScheme, they are performed by an out-of-line routine in native code as well as in byte code; likewise for deallocating and unlinking a continuation frame. This made it possible for us to test all four strategies using identical native code.

We were unable to test the heap strategy because MacScheme uses continuation frames of various sizes.

We tested our four strategies on two outrageously continuation-intensive benchmarks. The `loop2` benchmark corresponds to a non-local-exit scenario in which a tight loop repeatedly throws to the same continuation. The `ctak` benchmark is a continuation-intensive variation of the call-intensive `tak` benchmark. As modified for Scheme, the `ctak` benchmark captures a continuation on every procedure call and throws on every return, which creates a recapture scenario.

Source code for these benchmarks, together with their less exotic analogues `loop1` and `tak`, are shown in an appendix. These benchmarks were run on a Macintosh II with 5 megabytes of RAM using generic arithmetic and fully safe code; stack-cache overflow was detected by software, because the Macintosh II was unable to detect stack overflows in hardware.

Figure 7 shows the time required by `loop2` relative to the time required by `loop1`, and the time required by `ctak` relative to the time required by `tak`. These relative timings are accurate to the precision shown in Figure 7. Under carefully controlled conditions, our absolute timings were repeatable to within plus or minus one count of the hardware timer, which had a resolution of 1/60 second.

We also ran these benchmarks on comparable hardware using PC Scheme and T3. These implementations used the stack or chunked-stack strategies, but we found that they did not perform as well on continuation-intensive benchmarks as our experimental implementation of the stack strategy.

The stack strategy is easily the worst of the tested strategies on the continuation-intensive benchmarks `loop2` and `ctak`. The other three strategies are about twice as fast. The incremental stack/heap strategy is a little slower than the stack/heap strategy on the `loop2` benchmark because it has to copy a frame into the stack cache each time through the loop. The gc strategy has a slight edge on the `ctak` benchmark because it never has to copy any frames.

Ten years later we verified these conclusions by using a coroutine benchmark `cofib`, similar to `ctak`, to compare the performance of the gc, stack/heap, Hieb-Dybvig-Bruggeman, and incremental stack/heap strategies as implemented in Standard ML of New Jersey, MacScheme, Chez Scheme, and Larceny. These results are shown in Figure 9 and are analyzed in Section 10.2.

10. Multitasking

Lightweight threads can be implemented very easily using first-class continuations [25]. This has been done in many implementations, including Concurrent ML, Chez Scheme, and MacScheme+Toolsmith [8, 39, 41].

Benchmark	Average Size of Continuation (32-bit words)	Per Cent Recaptured	Adjusted Percentage (see text)
tak	189	69	23
ctak	166	76	37
takl	195	75	37
boyer	381	81	58
browse	105	76	25
destructive	69	78	3
traverse-init	68	82	23
traverse	205	90	74
deriv	96	70	2
dderiv	96	70	2
div-iter	61	80	0
div-rec	476	54	0
fft	80	91	65
puzzle	159	88	68
triangle	207	81	54
fprint	103	82	42
fread	202	83	58
tprint	106	81	41
compiler	455	92	83

Figure 8. Multitasking can create a recapture scenario. The "Per Cent Recaptured" column shows how much of the continuation remained unchanged from one task switch to the next in MacScheme+Toolsmith. The "Adjusted Percentage" column suggests what these percentages might have been with a less continuation-intensive implementation of multitasking.

In this section we explain how multitasking creates a recapture scenario, which can be exploited by the stack/heap, incremental stack/heap, and Hieb-Dybvig-Bruggeman strategies. We then consider the performance of first-class continuations for multitasking and coroutines.

10.1. Multitasking creates a recapture scenario

The MacScheme compiler implicitly inserts code to decrement a countdown timer at every backward branch, at every procedure call, and at every return. When the timer reaches zero, this code generates a software interrupt by calling a subroutine within MacScheme's runtime system. This routine polls the operating system to learn whether any enabled events are pending. If so, then the event is packaged as a Scheme data structure that can be passed to the interrupt handler. Otherwise the interrupt becomes a task-switch interrupt.

The default value of the countdown timer generated a task switch interrupt for every 5000 procedure calls (counting tail calls). On the Macintosh II this generated at least ten task switches per second. To improve responsiveness, the interrupt rate was increased to one interrupt for every 500 calls for a short time following each user interface event.

Since the continuation tends to change little during short periods of time, most programs written using MacScheme+Toolsmith matched the recapture scenario fairly well. Figure 8 quantifies the extent to which the MacScheme compiler and eighteen of the Gabriel benchmarks matched the recapture scenario when multitasking was enabled. Only the compiler and the `ctak` benchmark called `call-with-current-continuation` at all.

One of the peculiarities of MacScheme+Toolsmith is that each timer interrupt captures a continuation. Since the task scheduler will capture exactly the same continuation, the average fraction of continuation structure that is being recaptured due to multitasking can never be less than half. We therefore adjusted the data to show what would happen if captures were performed only by the task scheduler. We also subtracted 37 words of continuation structure created by the read/eval/print loop and other system code. The last column of Figure 8 shows the adjusted percentages. These numbers show that the extent to which a program matches the recapture scenario can be sensitive to the details of an implementation. The smaller benchmarks are more sensitive than the larger ones.

Higher switch rates would create even more of a recapture scenario.

10.2. Performance of multitasking

Mateu's implementation of the stack/heap strategy on a Sun/670MP achieved more than 150,000 coroutine switches per second for the `samefringe` benchmark, with roughly one task switch for every two procedure calls, but excessive garbage collection limited its usefulness. Mateu's implementation of the stack/heap strategy with special support for coroutines reduced the overhead of garbage collection and

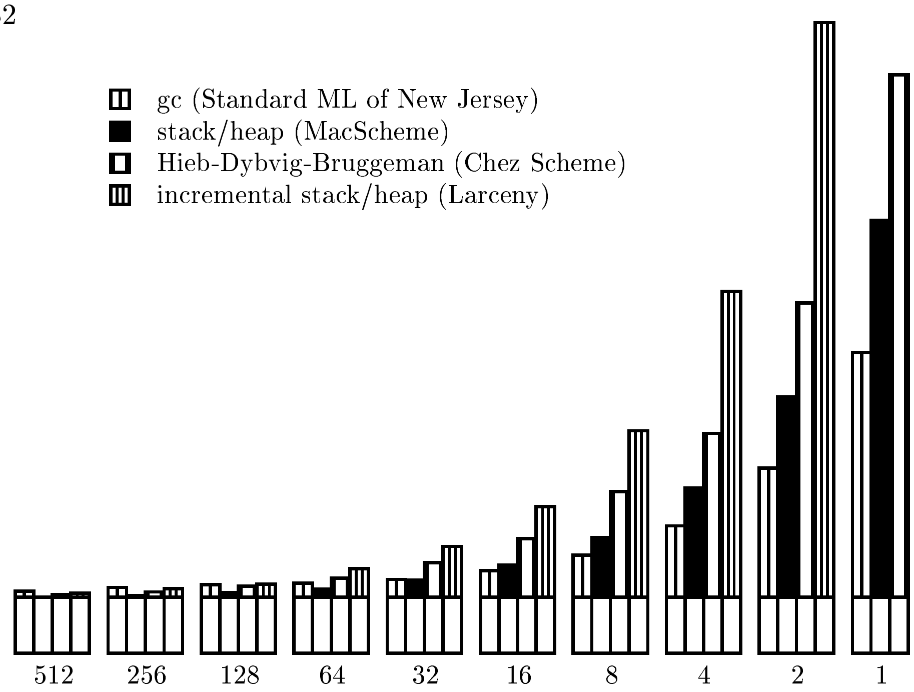

Figure 9. CPU times for the 100-thread `cofib` benchmark relative to the time required to execute the threads sequentially (represented by the white rectangle at the bottom of each bar), as the frequency of context switches increases from one context switch for every 512 procedure calls to one context switch for every call. A bar has been omitted from the right of this figure because it would be almost twice as high as the bar to its left.

was able to achieve 430,000 coroutine switches per second for the same benchmark [33].

Bruggeman, Waddell, and Dybvig have reported results from a synthetic multitasking benchmark that creates 10, 100, or 1000 threads, each of which computes the 20th Fibonacci number using the doubly recursive (exponential) algorithm [8]. They ran this benchmark on a DEC Alpha 3000/600 to compare the performance of their implementation of one-shot continuations with the Hieb-Dybvig-Bruggeman strategy.

At low switch rates, with at least 128 procedure calls between task switches (about 5000 task switches per second on their machine), there was little difference between the performance of one-shot continuations and the Hieb-Dybvig-Bruggeman strategy. At high switch rates, including the extremely high rate of one task switch for every procedure call, one-shot continuations were up to 15% faster than the Hieb-Dybvig-Bruggeman strategy. These results suggest that only the most demanding uses of multitasking or coroutines would benefit from one-shot continuations.

The stack/heap strategy used in MacScheme+Toolsmith performed well when there was one task switch for every 5000 procedure calls, but operating system and user interface overhead limited performance at higher interrupt rates. To separate the overhead of continuations from unrelated operating system and task scheduler

	procedure calls per context switch						
	512	256	128	64	32	16	8
gc (SML/NJ)	1.12	1.11	1.15	1.18	1.24	1.41	1.70
stack/heap (MacScheme)	1.00	1.03	1.08	1.15	1.31	1.60	2.12
HDB (Chez Scheme)	1.05	1.10	1.21	1.35	1.65	2.10	2.98
incremental stack/heap (Larceny)	1.08	1.16	1.24	1.53	1.95	2.70	4.13

Figure 10. CPU times for the 100-thread `cofib` benchmark relative to the time required to execute the threads sequentially, as the frequency of context switches increases from one context switch for every 512 procedure calls to one context switch for every 8 calls.

overhead, we modified the Fibonacci benchmark to use explicit coroutining instead of multitasking; this also made the benchmark more portable. We refer to our modified benchmark as `cofib`.

We used `cofib` to benchmark four implementations that use four different strategies for continuations:

- the gc strategy as implemented in Standard ML of New Jersey v110.0.3, running on a SPARC Ultra 1;
- the stack/heap strategy as implemented in MacScheme v4.2, running on a Macintosh PowerBook 170 with 4 megabytes of RAM; this machine could not run the 1000-thread version of the benchmark without paging, and does not have a data cache;
- the Hieb-Dybvig-Bruggeman strategy as implemented in Chez Scheme v5.0a, running on a SPARCserver and also on the SPARC Ultra 1; and
- the incremental stack/heap strategy as implemented in Larceny v0.34, running on the SPARC Ultra 1.

For Chez Scheme, the relative overhead of coroutining was consistent across the two machines we benchmarked; we report timings for the SPARC Ultra 1.

Figures 9 and 10 show CPU times for the 100-thread `cofib` benchmark relative to the time required to execute the threads sequentially. The timings shown for Standard ML of New Jersey are the arithmetic mean of 12 runs. For each frequency of context switching, including sequential execution, the sample deviation was less than 3% of the mean.

At a low switch rate, with 512 procedure calls between task switches (about 8000 task switches per second on the SPARC Ultra 1), the gc strategy has more overhead than the other strategies. This can be explained by its cache read misses, as discussed at the end of Section 6.1.

At high switch rates the cost of copying continuation frames becomes apparent. The gc strategy does not copy any frames. The stack/heap strategy makes exactly one copy of every frame that is live at a task switch, the Hieb-Dybvig-Bruggeman strategy makes at least one, and the incremental stack/heap strategy makes at least two.

Figure 9 suggests that, for any fixed overhead that is large enough to expose the cost of copying continuation frames, the gc strategy can perform about 1.5 times as many task switches per second as the stack/heap strategy, the stack/heap strategy can perform about twice as many as the Hieb-Dybvig-Bruggeman strategy, and the Hieb-Dybvig-Bruggeman strategy can perform twice as many as the incremental stack/heap strategy. If the largest acceptable overhead is 100%, then current hardware can perform about 300,000 task switches per second using the gc strategy, 200,000 task switches per second using the stack/heap strategy, 100,000 task switches per second using the Hieb-Dybvig-Bruggeman strategy, and 50,000 task switches per second using the incremental stack/heap strategy.

11. Difficulty of implementation

We recommend the incremental stack/heap and Hieb-Dybvig-Bruggeman strategies for implementing first-class continuations in almost-functional languages like Scheme and Standard ML. For more imperative languages, the stack/heap strategy may deserve consideration.

Appel and Shao characterized these strategies as “complicated to implement”, citing seven specific complications. In this section we review these issues and how they are resolved in Larceny version 0.35 [14].

First-class continuations: Larceny uses the incremental stack/heap strategy described in Section 4.7.

Frame descriptors: Larceny stores a frame descriptor in every frame, which adds two instructions to the cost of creating a frame. This is unnecessary, and is likely to change in a future version.

Detection of stack-cache overflow: The incremental stack/heap strategy allows a single stack cache to be used by multiple threads. This makes it easier to detect stack-cache overflow in hardware. Nonetheless Larceny relies on software to detect stack-cache overflow. This adds two instructions to the cost of creating a frame.

Generational garbage collection: Larceny's stack cache decreases the size of the root set that its generational garbage collector must scan on every collection, so there is no need to maintain a separate “watermark” for this purpose.

Multitasking: The incremental stack/heap strategy uses a single stack cache to support multiple threads. This is adequate for applications with ten thousand task switches per second. Higher rates of task switching can be accomodated by the Hieb-Dybvig-Bruggeman or stack/heap strategies. See Section 10.2.

Proper tail recursion: Almost-functional languages such as Scheme and Standard ML depend upon proper tail recursion, which conflicts with stack allocation of variables [11, 16]. This conflict can be resolved by using the complicated technique described by Chris Hanson [24], or by abandoning stack allocation for non-local variables.

The Twobit compiler used in Larceny does not use stack allocation for non-local variables. Lambda lifting converts almost all non-local variables into local variables, and the few non-local variables that remain after lambda lifting are allocated on the heap [14].

Space complexity: For the most part, issues of space complexity are orthogonal to the strategy that is used to implement continuations. That strategy affects space complexity only if it creates multiple copies of frames or allows continuation frames to be reused.

The stack strategy can increase the asymptotic space complexity of programs that recapture continuations. The closure that is created to represent an escape procedure occupies some storage in any case, so the chunked-stack, incremental stack/heap, and Hieb-Dybvig-Bruggeman strategies are safe for asymptotic space complexity provided there is a bound, such as the fixed size of a stack cache, on the total size of the continuation frames that are copied on a stack-cache underflow. In Larceny 0.35, stack-cache underflows copy a single frame.

If frames are reused, then the compiler may have to emit code to clear any slots of a frame that have not been overwritten and are no longer live when the frame is reused for a subsequent non-tail call. Alternatively, the compiler can add a descriptor to each frame that tells the garbage collector which of the frame's slots are live. As noted by Appel and Shao, this descriptor does not imply any runtime overhead, because the runtime system can maintain a mapping from return addresses to frame descriptors [2].

12. Conclusion

Many strategies can be used to implement first-class continuations. On most programs the zero-overhead strategies perform better than the gc strategy, but all of the strategies have indirect costs. The incremental stack/heap strategy is a zero-overhead strategy that performs well and is not hard to implement. The Hieb-Dybvig-Bruggeman and stack/heap strategies are also attractive, and perform better for multitasking.

Acknowledgments

Techniques invented for Algol 60 use a single stack to represent both environments and continuations. Sometime during the 1982–1983 academic year Jonathan Rees pointed out that we could forget about environments by assuming that all variables are in registers or in heap-allocated storage. This insight led us to invent the stack/heap and incremental stack/heap strategies. The stack/heap strategy was invented independently at about the same time by the implementors of Tektronix Smalltalk [47].

The comments and experience of Norman Adams, Lars Hansen, Richard Kelsey, Jonathan Rees, Allen Wirfs-Brock, and several anonymous referees were very helpful to us.

Our revision of this paper was supported by NSF grant CCR-9629801.

References

1. Appel, Andrew W. *Compiling with Continuations.* Cambridge University Press, 1992.
2. Appel, Andrew W., and Shao, Zhong. An empirical and analytic study of stack vs. heap cost for languages with closures. *Journal of Functional Programming* 6, 1 (1996), 47–74.
3. Appel, Andrew W. Modern Compiler Implementation in Java. Cambridge University Press, 1998.
4. Appel, Andrew W, and Shao, Zhong. Personal communications by electronic mail in October 1996 and September 1998, and by a telephone conference on 9 September 1998.
5. Bartley, David H., and Jensen, John C. The implementation of PC Scheme. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming* (August 1986), 86–93.
6. Berry, Daniel M. Block structure: retention or deletion? (Extended Abstract). In *Conference Record of the Third Annual ACM Symposium on Theory of Computing*, May 1971, 86–100.
7. Bobrow, D. G., and Wegbreit, Ben. A model and stack implementation of multiple environments. CACM 16, 10 (Oct. 1973) 591–603.
8. Bruggeman, C., Waddell, O., and Dybvig, R. K. Representing control in the presence of one-shot continuations. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1996, *SIGPLAN Notices* 31, 5 (May 1996), 99–107.
9. Burger, Robert G., Waddell, Oscar, and Dybvig, R. Kent. Register allocation using lazy saves, eager restores, and greedy shuffling. In *Proceedings of the 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1995, 130–138.
10. Caudill, Patrick J., and Wirfs-Brock, Allen. A third generation Smalltalk-80 implementation. In *Conference Proceedings of OOPSLA '86*, *SIGPLAN Notices* 21, 11 (November 1986), 119–130.
11. Chase, David R. Safety considerations for storage allocation optimizations. In *Proceedings of the 1988 ACM Conference on Programming Language Design and Implementation*, 1–10.
12. Cheng, Perry, Harper, Robert, and Lee, Peter. Generational stack collection and profile-driven pretenuring. *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1998, 162–173.
13. Clinger, William D., Hartheimer, Anne H., and Ost, Eric. Implementation strategies for continuations. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, 124–131.
14. Clinger, William D., and Hansen, Lars Thomas. Lambda, the ultimate label, or a simple optimizing compiler for Scheme. In *Proc. 1994 ACM Conference on Lisp and Functional Programming*, 1994, 128–139.
15. Clinger, William D., and Hansen, Lars T. Generational garbage collection and the radioactive decay model. *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1997, 97–108.
16. Clinger, William D. Proper tail recursion and space efficiency. *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1998, 174–185.
17. Cytron, Ron, Ferrante, Jeanne, Rosen, Barry N., Wegman, Mark N., and Zadeck, F. Kenneth. Efficiently computing static single assignment form and the control dependence graph. ACM TOPLAS 13, 4 (October 1991), 451–490.
18. Danvy, Olivier. Memory allocation and higher-order functions. In *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, June 1987, 241–252.
19. Deutsch, L. Peter, and Schiffman, Allan M. Efficient implementation of the Smalltalk-80 system. In *Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages*, January 1984, 297–302.

20. Feeley, Marc. Gambit-C version 3.0. An implementation of Scheme available via `http://www.iro.umontreal.ca/~gambit`, 6 May 1998.
21. Fischer, Michael J. Lambda-calculus schemata. In *Journal of Lisp and Symbolic Computation* 6, 3/4 (December 1993), 259–288.
22. Gabriel, Richard P. *Performance and Evaluation of Lisp Systems.* The MIT Press, 1985.
23. Goldberg, Adele, and Robson, David. *Smalltalk-80: the Language and its Implementation.* Addison-Wesley, 1983.
24. Hanson, Chris. Efficient stack allocation for tail-recursive languages. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, 106–118.
25. Haynes, Christopher T., and Friedman, Daniel P. Engines build process abstractions. *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming* (August 1984), 18–24.
26. Hieb, Robert, Dybvig, R. Kent, and Bruggeman, Carl. Representing control in the presence of first-class continuations. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, *ACM SIGPLAN Notices* 25, 6 (June 1990), 66-77.
27. Holloway, Jack, Steele, Guy L, Sussman, Gerald Jay, and Bell, Alan. The SCHEME-79 chip. MIT AI Laboratory, AI Memo 559 (January 1980).
28. IEEE Standard 1178-1990. *IEEE Standard for the Scheme Programming Language.* IEEE, New York, 1991.
29. Kelsey, Richard, Clinger, William, and Rees, Jonathan. Revised^5 report on the algorithmic language Scheme. In *Higher-Order and Symbolic Computation* 11, 3 (1998), 7–105.
30. Kranz, David A., Kelsey, Richard, Rees, Jonathan A., Hudak, Paul, Philbin, James, and Adams, Norman I. Orbit: An optimizing compiler for Scheme. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction. SIGPLAN Notices 21*, 7 (July 1986), 219–223.
31. Kranz, David Andrew. *ORBIT: An Optimizing Compiler for Scheme.* PhD thesis, Yale University, May 1988.
32. Lightship Software. *MacScheme Manual and Software.* The Scientific Press, 1990.
33. Mateu, Luis. An efficient implementation for coroutines. In Bekkers, Y., and Cohen, J. [editors]. *Memory Management* (Proceedings of the International Workshop on Memory Management IWMM 92), Springer-Verlag, 1992, 230–247.
34. McDermott, D. An efficient environment allocation scheme in an interpreter for a lexically-scoped Lisp. In *Conference Record of the 1980 Lisp Conference* (August 1980), 154–162.
35. Miranda, Eliot. BrouHaHa—a portable Smalltalk interpreter. In *Conference Proceedings of OOPSLA '87*, *SIGPLAN Notices* 22, 12 (December 1987), 354–365.
36. Moss, J. Eliot B. Managing stack frames in Smalltalk. In *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, June 1987, 229–240.
37. *PowerPC 601 RISC Microprocessor User's Manual.* Motorola, 1993.
38. Rees, Jonathan, and Clinger, William [editors]. Revised^3 report on the algorithmic language Scheme. In *SIGPLAN Notices* 21, 12 (December 1986), 37–79.
39. Reppy, J. H. CML: A higher-order concurrent language. In *Proceedings of the 1991 ACM SIGPLAN Conference on Programming Language Design and Implementation*, *SIGPLAN Notices* 26, 6 (1991), 294–305.
40. Samples, A. Dain, Ungar, David, and Hilfinger, Paul. SOAR: Smalltalk without bytecodes In *Conference Proceedings of OOPSLA '86*, *SIGPLAN Notices* 21, 11 (November 1986), 107–118.
41. Semantic Microsystems. *MacScheme+Toolsmith.* August 1987.
42. Shao, Zhong, and Appel, Andrew W. Space-efficient closure representations. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, 150–161.
43. Steele Jr, Guy Lewis. Macaroni is better than spaghetti. In *Proceedings of the Symposium on Artificial Intelligence and Programming Languages* (August 1977), 60–66.
44. Suzuki, Norihisa, and Terada, Minoru. Creating efficient systems for object-oriented languages. In *Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages*, January 1984, 290–296.
45. Ungar, David M. *The Design and Evaluation of a High Performance Smalltalk System.* The MIT Press, 1987.

46. Weaver, David L., and Germond, Tom. *The SPARC Architecture Manual*, Version 9. SPARC International and PTR Prentice Hall, 1994.
47. Wirfs-Brock, Allen. Personal communication, April 1988. Tektronix Smalltalk was described by Caudill and Wirfs-brock, but not in enough detail for us to realize that Tektronix Smalltalk uses the stack/heap strategy rather than the stack strategy [10].

Appendix 1: PowerPC assembly language

The PowerPC is a load/store architecture with 32 general purpose registers, 32 floating point registers, and a small number of special purpose registers such as the link register, which is used to hold a return address. Memory is byte-addressable. A word of memory consists of four 8-bit bytes. The `lwz` (Load Word and Zero) instruction loads a word from memory into a general purpose register; the `stw` (Store Word) instruction stores the contents of a general register into memory. The first operand of the `lwz` and `stw` instructions is the general register being loaded or stored. The effective address is formed by adding the second operand (a displacement, in bytes) to the contents of the general register specified by the third operand. Thus

```
lwz     r3,0(r1)            // Load Word and Zero
stw     r3,4(r1)            // Store Word
```

copies the word of memory whose address is in register `r1` to the following word of memory.

Most integer instructions operate on the contents of two general registers and place their result in a destination register. The destination register is the first operand, so

```
or      r3,r4,r4            // Or (inclusive)
```

copies the contents of register `r4` to register `r3`. An immediate instruction takes an integer as its last operand, so

```
addi    r1,r1,-8            // Add Immediate
```

decrements register `r1` by 8.

The `cmpw` (Compare Word) and `cmpwi` (Compare Word Immediate) instructions compare two integers. When only two operands are specified for these instructions, the implicit destination is a condition register, which can be used to control a conditional branch instruction such as `bge` (Branch if Greater or Equal) or `blt` (Branch if Less Than). Thus

```
cmpw    r3,r3               // Compare Word
bge     L1790               // Branch if Greater or Equal
```

always branches to `L1790`, but

```
cmpwi   r4,0                // Compare Word Immediate
blt     L25                 // Branch if Less Than
```

branches to L25 if and only if the value in `r4` is negative.

The `bl` (Branch and Link) instruction is an unconditional branch that places the address of the following instruction into the link register. The `blr` (Branch to Link Register) instruction branches to the address that is contained within the link register. Thus the following code implements an infinite loop:

```
      bl     next                // Branch and Link
next: blr                        // Branch to Link Register
```

The `mflr` (Move From Link Register) instruction copies the contents of the link register into a general register, and the `mtlr` (Move To Link Register) instruction copies the contents of a general register into the link register. These instructions are used to save and to restore a return address, as illustrated by the example in Section 3.1.

Appendix 2: Source code for benchmarks

These benchmarks use a `run-benchmark` procedure as in the benchmarks that come with Gambit-C [20].

```
;;; LOOP1 -- A perverse way to write a loop.

(define (loop1 n)
  (let ((n n)
        (k 0))
    (define (loop ignored)
      (if (zero? n)
          'done
          (begin (set! n (- n 1))
                 (loop #t))))
    (loop #t)))

(run-benchmark "Loop1" 1 (lambda () (loop1 1000000))
                         (lambda (result) (eq? result 'done)))

;;; LOOP2 -- An extremely perverse way to write a loop.

(define (loop2 n)
  (let ((n n)
        (k 0))
    (define (loop ignored)
      (call-with-current-continuation
       (lambda (cont)
         (set! k cont)))
      (if (zero? n)
          'done
          (begin (set! n (- n 1))
                 (k #t))))
    (loop #t)))
```

```
(run-benchmark "Loop2" 1 (lambda () (loop2 1000000))
                         (lambda (result) (eq? result 'done)))

;;; TAK -- A vanilla version of the TAKeuchi function

(define (tak x y z)
  (if (not (< y x))
      z
      (tak (tak (- x 1) y z)
           (tak (- y 1) z x)
           (tak (- z 1) x y))))

(run-benchmark "TAK" 1 (lambda () (tak 18 12 6))
                       (lambda (result) (= result 7)))

;;; CTAK -- A version of the TAK procedure that uses
;;; continuations.

(define (ctak x y z)
  (call-with-current-continuation
   (lambda (k)
     (ctak-aux k x y z))))

(define (ctak-aux k x y z)
  (cond ((not (< y x))  ;xy
         (k z))
        (else (call-with-current-continuation
               (lambda (k)
                 (ctak-aux
                  k
                  (call-with-current-continuation
                   (lambda (k)
                     (ctak-aux k
                               (- x 1)
                               y
                               z)))
                  (call-with-current-continuation
                   (lambda (k)
                     (ctak-aux k
                               (- y 1)
                               z
                               x)))
                  (call-with-current-continuation
                   (lambda (k)
                     (ctak-aux k
                               (- z 1)
                               x
                               y))))))))))
```

```
(run-benchmark "CTAK" 1 (lambda () (ctak 18 12 6))
                        (lambda (result) (= result 7)))

(* Coroutine benchmark for Standard ML of New Jersey. *)

fun main (nthreads, n, ncalls) =
  let open Array
      open SMLofNJ.Cont
  in callcc (fn (terminate) =>
      let val tasks = tabulate (nthreads,
                                fn (i) =>
                                 fn (ignored) =>
                                  throw terminate 0)
          val count = ref nthreads
          val counter = ref ncalls
          fun make_task (i) =
            let
              val j = let val j = i + 1
                      in if j >= nthreads then 0 else j
                      end
              fun task_switch (k) =
               ((*print "\nTask switch,";*)
                update (tasks, i, fn (x) => throw k x);
                counter := ncalls;
                sub (tasks, j) (0))
              fun task_switch0 (f) =
               (update (tasks, i, f);
                counter := ncalls;
                sub (tasks, j) (0))
              fun fib (n) =
                if !counter = 0
                  then (callcc task_switch;
                        fib (n))
                  else (counter := !counter - 1;
                        if n < 2
                          then n
                          else fib (n - 1) + fib (n - 2))
              fun done (ignored) =
               (print "\nThis shouldn't have happened.";
                throw terminate 0)
            in fn (ignored) =>
                  let val result = fib (n)
                  in (count := !count -1;
                      if !count = 0
                        then throw terminate result
                        else task_switch0 done)
```

```
                  end
            end
          fun init i =
            if i < nthreads
              then (update (tasks, i, make_task (i));
                    init (i + 1))
              else ()
      in (init 0;
          sub (tasks, 0) (0))
      end)
  end

fun cofib_benchmark (nthreads) =
  let fun benchmark (name, ncalls) =
        run_benchmark (name, 1, fn () => main (nthreads, 20, ncalls),
                                fn (x) => (x = 6765))
  in (benchmark ("Sequential", 1000000);
      benchmark ("512 calls/switch", 512);
      benchmark ("256 calls/switch", 256);
      benchmark ("128 calls/switch", 128);
      benchmark ("64 calls/switch", 64);
      benchmark ("32 calls/switch", 32);
      benchmark ("16 calls/switch", 16);
      benchmark ("8 calls/switch", 8);
      benchmark ("4 calls/switch", 4);
      benchmark ("2 calls/switch", 2);
      benchmark ("1 call/switch", 1))
  end
```