Weighing Continuations for Concurrency

Kavon Farvardin
Abstract

There have been a number of efforts to broaden the use of lightweight continuations for concurrency in programming languages, however, the implementation trade-offs of various designs under sequential and concurrent workloads are not well understood. Prior empirical evaluations used cross-language and cross-compiler analyses, leaving much of the folklore surrounding their performance without evidence.

We present the implementation of a single compiler and runtime system that supports a broad range of strategies for continuations. Using this compiler, we conduct an empirical analysis of both the performance and challenges involved in implementing different forms of continuations for high-performance concurrency.
Overview

1.1 Introduction

Parallel languages can support lightweight threading through the use of reified continuations, which are an abstraction that represents the rest of the program (i.e., the control stack) at a given point of its execution. Continuations naturally encapsulate a paused thread in the implementation of user-level threading libraries (Fluet et al. 2008). While the application of continuations to concurrent systems is certainly not a new idea (Wand 1980; Reppy 1999), the aspects of their implementation are fundamental to its scalability.

Unfortunately, determining which implementation strategy to use for continuations is an elusive topic. The control stack is the foundation upon which recursive languages are implemented, so their design is constrained by many other aspects of the language implementation, e.g., language features and the runtime system. In addition, the trade-offs with respect to performance and implementation complexity of various strategies are not well understood.

Much of the current understanding of performance trade-offs are based on cross-language and cross-compiler comparisons (Clinger et al. 1999), simulations and theoretical analysis (Appel and Shao 1996), or direct measurements performed nearly 30 years ago (Clinger et al. 1988). In the absence of recent, well-normalized measurements and their implementation specifics, the path of least resistance when choosing a strategy is to trust folklore.

For example, MULTILTON and GUILE avoided the use of the segmented stack strategy because reports from RUST and Go developers suggested poor performance due to segment bouncing (Sivaramakrishnan et al. 2014; Wingo 2014; Randall 2013; Anderson 2013). Interestingly, the bouncing was solved by Bruggeman et al. (1996) in their original design of segmented stacks. While their solution seems impossible to implement in, say, RUST, it is possible in higher-level languages like ML or SCHEME. Sometimes the folklore is misleading! Thus, the feasibility of implementing crucial aspects of a design is an important consideration when weighing one strategy against another.

callcc is passé  Prior comparison of continuation strategies, led by Clinger et al. (1999), have focused on multi-shot continuations obtained via callcc, which has a higher degree of expressive power than needed for many applications. Multi-shot continuations are control stacks that are captured such that they can be resumed multiple times. Thus, when implementing callcc, making a copy of the control stack is typically necessary, adding overhead.

Bruggeman et al. (1996) introduced one-shot continuations, which trade some power for efficiency in that copying is no longer necessary. Realistic programs only resume a continuation at most once, making one-shot continuations a better fit for lightweight concurrency (Fisher and Reppy 2002; Li et al. 2007a). However, the performance of various strategies for one-shot continuations has not been investigated.

1.2 Contributions

By extending the compiler for a concurrent, functional language to support multiple strategies for one-shot continuations, we evaluate the trade-offs of each strategy under a number of metrics. We are interested not only in performance under various workloads, but in qualities such as support for deep recursion and implementation complexity. To our knowledge, this is the first head-to-head comparison of one-shot continuation strategies.
Chapter 2

Design Space

In this chapter, we provide background on various design aspects associated with the allocation and management of continuations.

2.1 Fundamental Aspects

2.1.1 Memory Structure

Whenever a non-tail call is encountered in the program, the current continuation is implicitly captured and saved into an additional frame, or activation record. The process of appending another activation record onto the current continuation is what we refer to as linking a frame. The structure of lightweight, one-shot continuations fall under a spectrum based on how pointers are used to link frames together.

On one end, the traditional contiguous stack requires no inter-frame pointers if the frame sizes are statically known, which is common in functional languages. This reduces the effort required to establish a new frame and locate the caller’s frame to simply adjusting the stack pointer by a fixed amount.

Variants of segmented stacks (Bruggeman et al. 1996) allocate smaller segments of memory at a time, linking each segment together with pointers. Frames within a segment are addressed as in a contiguous stack, and the inter-segment pointers are only accessed when a segment underflows or overflows. Thus, as the segment size shrinks, segment pointer reads and writes become more frequent. If we were to allocate one segment per frame, we are effectively at the other end of the spectrum, which is to allocate each frame separately in the heap and link them together with pointers.

When using CPS-based code generation, the closure conversion strategy plays an important role in determining the structure of the control stack. A flat closure (Cardelli 1983) is an array consisting of a function pointer and the free variables required by that function, which in CPS includes the current continuation. Immutable linked frames are equivalent to using flat closures to represent the continuation of a non-tail call. Significant efficiency gains for these linked frames are attainable using the more sophisticated closure strategy by Shao and Appel (2000).

2.1.2 Overflow Detection

Functional programs often make use of deep recursion, which may trigger the dreaded stack overflow. The two primary methods of detecting stack overflow when trying to extend the current continuation are memory faults and explicit limit checks.

Memory Faults   Placing a memory protected guard page at the end of the memory region where the control stack is allocated is a classic approach to cheaply detecting overflow, as no additional instructions are required per function call. The trade-off is that stack overflow becomes extremely difficult to recover from, as the garbage collector may need to relocate the stack to a larger area. Every instruction that accesses the stack is required to be a safe point in the program, where garbage collection may occur. Tracking the locations of live roots at all of these points is possible (Stichnoth et al. 1999) but a huge pain. For example, a memory fault may occur while spilling a register during an object initialization, leaving that heap object in a uniquely indeterminate state.

Thus, many language implementations using this approach simply crash the program rather than try to recover. For programs making use of deep recursion, the programmer is left with the option of setting a
large default stack size in hopes of avoiding overflow. Given a huge number of stacks when using concurrency, this could produce noticeable internal fragmentation.

**Limit Checks** If we emit explicit checks for overflow of the memory region when trying to allocate a stack frame, overflow recovery is relatively easy. This is because we are restricting safe points to one per function entry, rather than per stack access, and the locations of live values are given by the function’s calling convention. Internal fragmentation is avoided by growing and shrinking stack areas on demand. Of course, the trade-off is a compare-and-branch instruction sequence for each function call, but the branch’s direction is predicable.

If stack frames are allocated directly in the heap, stack overflow is no different than heap overflow, and thus separate stack and heap limit checks are no longer necessary.

**2.1.3 Reducing Frame Allocation**

Leaf functions are those that may only contain tail calls, and thus their stack frame is never passed to any other function. During the call of a leaf function, if the caller guarantees that there is free space beyond the stack pointer, sometimes called a red zone, the callee would not need to explicitly establish a stack frame, if any stack space is used for spills. Instead, spill slots are assigned past the stack pointer, and accessed without changing its register.

Callee-saved registers (CSRs) are an optimistic register convention that allows the caller of a function to pass the burden of saving those values to a frame onto the callee. The hope is that the callee and its descendents do not make use of those registers, so they will remain in register until the callee returns. If any of the descendents do need those registers, there is no longer a benefit as they will still be saved to memory, typically in the descendant function’s prologue.

Chow (1988) describes a technique to avoid saving CSRs to memory unless if the path taken within the function actually uses those registers, i.e., it wraps only the parts of a function that needs those registers with a spill-restore sequence, not the entire function. Overall, the only downside to using CSRs is that the source of callee-saved values are dynamic during execution, which complicates garbage-collected runtime systems (Section 2.1.5).

**2.1.4 Reusing Frames**

One of the arguments in favor of reusing frames is to reduce cache misses: between multiple non-tail calls within the same function, space allocated for the continuation of a previous call is reused for the next call. The additional benefit of this approach is a reduction in the number of instructions that copy values that are live but unused across multiple non-tail calls from one frame to the next.

Additionally, keeping frames contiguously allocated and out of the heap is commonly held as a benefit specifically for stacks in increasing cache locality and reducing the garbage collection burden. However, Stefanovic and Moss (1994) found that the lifetimes of immutable, heap-allocated frames were extremely short, which suggests that with sufficient memory and a copy-collected nursery, the load on the garbage collector may not be so large (Appel 1987). Hertz and Berger (2005) also found that the regular compaction offered by such a nursery is beneficial to cache locality verses other schemes for heap allocation, but it is unclear whether this benefit can match the efficiency of stack allocation.

**2.1.5 Finding Roots**

Accurately identifying roots in a frame is essential in a garbage collected runtime system. The use of uniformly sized values plus a bit tagging scheme is sufficient to distinguish pointers from non-pointers, so long as register spills do not alias pointers. Otherwise, a standard solution is to include an additional layout descriptor tag in each frame. This tag can be avoided if a mapping from return addresses to layouts are generated by the compiler (Cheng et al. 1998). An older approach to identifying roots is to use two stacks, one for pointers and the other for non-pointers, however, adjusting and checking two different stack pointers can add overhead (Peyton Jones and Salkild 1989).
**Space Complexity**  Frame reuse in a garbage-collected runtime introduces the challenge of identifying only the roots in the frame that are live after the call, since we may have dead pointers left in an unused frame slot from a prior non-tail call. We do not want the garbage collector to misidentify a dead pointer as a live root, because we would no longer preserve the space complexity of the program (Appel 1992).

If the bit tagging scheme is used, it would be necessary to overwrite these stale pointers once they have become dead. With a layout descriptor, the number of writes to remain safe-for-space is reduced to changing the descriptor before each call. When a return-address table is used, the layouts must simply describe only the currently live locations.

**Callee-saved Registers**  The use of callee-saved registers adds another layer of complexity in terms of identifying roots. The difficulty is that the type of value in those registers, *i.e.*, whether the value is a pointer, is dependent upon the function’s caller. Cheng et al. (1998) found that the presence of callee-saved registers requires additional information output by the compiler, combined with a two pass approach when scanning the stack to compute the root set. A simple compromise would be to only allow callee-saved registers to contain non-pointer values. Look at Appel’s CSR approach and see how it avoids this problem.

### 2.1.6 Generational Stack Collection

Generational garbage collection has proven to be an efficient means of implementing functional languages, given the high turnover rate of heap allocations. Cheng et al. (1998) found that, in a generational runtime system, stacks also benefit from a generational approach. They found that much of the garbage collector’s time was spent rescanning deep control stacks, where most of the frame’s roots have already been promoted.

There are a number of ways to implement generational stack collection (Anderson 2010), with the universal goal of detecting whether a given stack frame has been modified since the last collection cycle. In Section 3.6 we describe our approach, which places a special marker in each stack frame, incurring only one additional instruction per function call. If stacks are also kept in the generational heap, a write barrier to detect stack frame updates may be necessary.

### 2.1.7 CPU and ABI Support

Historically, instruction sets such as the x86 have contained dedicated instructions to assist with the allocation of activation records. Some CPUs have dedicated hardware in the instruction pipeline to reduce the cost of adjusting the stack pointer and increase branch prediction rates for calls and returns (Gochman et al. 2003). Pettersson et al. (2002) found that without the use of dedicated call/return instructions on x86-64 CPUs, overall performance dropped by 9.2%.

With the use of dedicated instructions for continuations, which requires the use of a special stack pointer register, the operating system’s application binary interface (ABI) constrains their design. Foreign functions typically expect a guard page to detect stack overflow (Section 2.1.2), thus handing over a stack pointer that is in the middle of a heap-allocated stack is dangerous without placing such a page in the heap. If stacks in the heap are also relocated, the collector must use system calls to initialize and destroy guard pages. An alternative is to switch to a page-protected stack for each foreign-function call, incurring a few instructions per call. If dedicated instructions are not used, the stack pointer register can be reserved to hold a stack for foreign-function calls, while another register is used for program’s control stack.

### 2.1.8 Capturing and Throwing

The primary design constraint for first-class continuations is the lifetime of the continuation, which is tied to the number of times it may be used. For example, the standard callcc function captures the current continuation, which is the point of return for the call, and passes it to its function argument. The continuation captured is multi-shot, thus it may be returned back to the call-site of the callcc and used later, *i.e.*, the continuation has unlimited extent. The act of returning from callcc is an implicit invocation of the continuation that was captured, so if the continuation were returned, one use would have already occurred. Thus, if frames are reused (Section 2.1.4), a copy of the stack at the point of capture must be preserved in
order to allow multiple invocations. The use of immutable stack frames removes the need for copying to implement \texttt{callcc} (Appel 1992).

In contrast, the lifetime of a one-shot continuation captured with \texttt{call1cc} is bounded by the point of capture, as returning from that point invalidates the captured continuation. Thus, no copying operation is required upon capture even when frames are reused. The act of capturing a one-shot continuation, sometimes referred to as an \textit{escape continuation}\textsuperscript{1}, is as cheap as C’s \texttt{setjmp}; effectively, a pointer into the stack at the capture point is all that is saved.

Inevitably, since all continuation throws we will consider abort the current continuation, throwing to an escape continuation is as cheap as performing a \texttt{longjmp}, which essentially just changes the stack pointer.

\textbf{Threading} \hspace{1em} On its own, \texttt{call1cc} is not always enough to implement a threading system. For example, in a typical contiguous stack with mutable frames, two continuations captured via \texttt{call1cc} would be adjacent to each other in memory, as one is the prefix of the other. Dispatching the prefixed continuation will trample the frames of the subsequent one, so in this case only one paused thread can be represented with an escape continuation.

To solve this, we add the operation \texttt{newstack} which allocates a continuation that will begin executing the provided function once thrown to. A call to \texttt{newstack} does not capture the current continuation. It provides a new delimited continuation whose prompt yields control to the scheduler, \textit{i.e.}, when the provided function returns, its value is thrown away and we look for more work.

One additional hiccup occurs in the presence of parallelism when accessing the scheduling queue. We pause a thread using \texttt{call1cc}, and then place it on the queue to dispatch other work. However, frames allocated while enqueuing that continuation and obtaining another one to dispatch may be adjacent to the captured continuation. Thus, there is the risk of another process trampling those frames if it dispatches the enqueued continuation before the process who captured the thread has a chance to dispatch. To solve this, Fisher and Reppy (2002) add a lock field the captured continuation until those frames are no longer needed, whereas Li et al. (2007b) model this exchange as higher-level transaction on the scheduling queue.

\textbf{2.2 Prior Analysis}

Appel and Shao (1996) compared the cost of using heap allocated frames against the use of a stack. Stack-allocated frames were found to be more complicated to implement than heap-allocated frames, while offering similar performance. Their arguments were supported by simulations measuring cache effects and instruction counts. One of the flaws in their simulation of stack behavior with their modified compiler was due to the lack of sharing among frames of multiple non-tail calls (see Clinger et al. 1999, sec. 7).

Clinger et al. (1988) extended the MacScheme+Toolsmith compiler with support for various implementations of multi-shot continuation designs. Each design was tested by modifying out-of-line routines that are invoked to create and retire each frame; the compiler was otherwise left unchanged. They found that the traditional stack performs well only when \texttt{callcc} is not used, and that heap-allocated frames perform very poorly when the heap size is small. However, the results of this experiment no longer reflect modern hardware, and multi-shot continuations are known to be overly taxing when stacks are used for concurrency because of the additional copying (Bruggeman et al. 1996).

Bruggeman et al. (1996) introduced the use of the cheaper one-shot continuations with their chunked-stack design, and compared it against heap-allocated frames via CPS using a synthetic workload. Their performance evaluation varied the number of function calls per context switch, and reported that heap-allocated frames are only a win when the switching rate is more frequent than once every four function calls. We know of no other quantitative evaluations of one-shot continuations for concurrency.

\textsuperscript{1}The terminology for this particular continuation is a bit muddled. Ramsey and Peyton Jones (2000); Fisher and Reppy (2002) cite differences with Bruggeman et al. (1996)’s definition of one-shot continuation, using the term escape continuation instead. We do not see a distinction and use both terms interchangeably.
Chapter 3

Experimental Framework

“A manticore is a fabulous creature with a lion’s body, a man’s face, and a sting in his tail.”

(Davies (1977))

In this chapter, we provide details of the extensions made to the Manticore system in order to evaluate various strategies for implementing one-shot continuations.

3.1 Overview

Manticore is an optimizing compiler system based on continuations that implements PARALLEL ML (Fluet et al. 2007). In particular, the compiler uses a continuation-passing style (CPS) transformation to aid optimization and code generation, producing code that allocates immutable continuations à la SML/NJ (Appel 1992). Manticore uses a form of Appel and Shao (1996)’s heap strategy for continuations (Section 3.2) paired with an appropriate garbage collector (Section 3.6).

One of the key problems with Appel and Shao (1996)’s experimental framework was in their simulation of a stack in SML/NJ for analysis. We take a different approach in that we extended the Manticore compiler and runtime system with real support for various stack-allocated continuation strategies.

To produce high-quality stack-based code, we make use of a direct-style (DS) transformation (Section 3.3) and then use a modified version of LLVM (Lattner 2002) to stack-allocate continuations. The continuation strategies now supported by Manticore are:

- Immutable Heap Frames (Section 3.2)
- Contiguous Stack (Section 3.4)
- Segmented Stack (Section 3.5)

The details of how these strategies are implemented in Manticore is the focus of this chapter.

3.2 Immutable Heap Frames via CPS

Talk briefly about closure-passing style, and how (Appel 1987) discussed the merits of the approach. Describe how we use flat closures, and also discuss the join point optimization, which should nicely lead us into the next section.

3.3 Direct-style Conversion

Manticore uses a higher-order CPS intermediate representation (IR) in its middle-end for optimization and to simplify the implementation of callcc in its back-end (Section 3.2). The lifetimes of each return continuation allocated are bounded by the lexical scope of the function, and thus are amenable to stack allocation (Adams et al. 1986). However, this stack optimization alone would not produce the type of code typically output by stack-oriented compilers. It would be necessary to also merge the layouts of the return
continuations of each non-tail call made from a given function, to enable efficient memory reuse. In addition, it is desirable to use architecture specific stack manipulation instructions (Gochman et al. 2003), but generating such instructions using LLVM is not possible while in CPS as every call is in tail position. Thus, we extended Manticore to undo CPS conversion after optimization to take advantage of LLVM for efficient, stack-allocated continuations.

While conversion to CPS is widely known, comparatively less has been said about converting CPS programs back to direct-style. In this section, we describe a practical application of direct-style conversion in a compiler. Our approach cherry-picks ideas from Reppy (2002); Kelsey (1995); Danvy and Lawall (1992). Direct-style conversion is performed during closure conversion, which takes the higher-order CPS IR and generates a first-order program, and relies on additional analysis information and a smaller transformation beforehand.

3.3.1 Preliminaries

Before delving into the conversion, we briefly discuss intermediate representations in Manticore.

blurb about how BOM is the primary direct-style IR in Manticore, and it supports continuation binding. The point is that not all continuations in CPS come from the conversion, but some may appear from the translation of higher-level constructs in the language.

Figure 3.1 shows the relevant parts of the higher-order CPS representation used in Manticore. During CPS conversion, we separate the additional continuation parameters that are added to each function and include their kind. This provides enough information to establish a classification of continuations after optimizations have been applied to the program in CPS (Section 3.3.2).

Figure X is a simplified CFG IR with added support for non-tail calls

3.3.2 Classifying Continuations

Before direct-style conversion occurs, we run an analysis pass over the CPS program to assign a classification to continuation bindings (i.e., Cont expressions) and their uses. We define the context of a non-function expression to be the innermost function in which it resides. In terms of Figure 3.1, the context is a lambda belonging to a Fun expression. Similarly, the current continuation is the designated return continuation bound as a parameter of the context. A continuation is an escaping value if it is saved to memory, or passed as a
non-continuation argument in an `Apply` or `Throw` expression. The overall goal of the analysis is to identify all second-class continuations in the program, which satisfy the following restriction.

**Definition 3.3.1.** (Second-class Restriction)
1. All uses of the continuation must occur in the context in which it is bound.
2. The continuation must not be an escaping value.

Any continuations which fail to satisfy the second-class restriction are considered first-class continuations. We further delineate the kind of a second-class continuation by their use in certain expressions. A second-class continuation is a return continuation if it appears as such an argument in at least one `Apply` expression. Otherwise, the continuation is considered a join continuation.

We mark each `Apply` expression depending on whether the function call is a tail call or not. While every function application appears in tail position in CPS, these calls are not necessarily tail calls, as they may grow the control stack. In particular, only an `Apply` which is passed the current continuation as its return continuation argument is a tail call; otherwise, the `Apply` is a non-tail call that returns to the given second-class continuation.

Similarly, we mark each `Throw` with its kind by inspecting the continuation thrown to. A local return to the caller occurs only if the throw is to the current continuation. Any other second-class throw, to either a return or join continuation, is simply a “jump-with-arguments” that does not change the current continuation. A throw to a first-class continuation is handled in the standard way, i.e., it will abandon the current continuation.

Is there even a need to distinguish between return and join conts? the real distinction is whether it’s parameter bound vs second-class vs first-class when throwing, and whether it is second-class in an `Apply`. I think even in the code it’s mapped to the same or-pattern.

**Taming CPS Optimizations** Because we are performing direct-style conversion after CPS optimizations have been applied to the program, our classification scheme must be impervious to such transformations.

```plaintext
1 fun outer (_ / outerRet) = let
2   fun f (_ / fRet) = throw fRet ()
3   fun g (_ / gRet) =
4     if ...
5     then apply f (_ / gRet)
6     else throw gRet ()
7     cont retK () =
8       ...
9     throw outerRet ()
10    in
11   apply g (_ / retK)
12 end

1 fun outer (_ / outerRet) = let
2   fun f (_ / _) = ... throw outerRet ()
3   fun g (_ / gRet) =
4     if ...
5     then apply f (_ / _)
6     else throw gRet ()
7     cont retK () =
8     ...
9     throw outerRet ()
10    in
11   apply g (_ / retK)
12 end
```

(a) Initial CPS program. (b) After inlining `retK` into the body of `f`.

Figure 3.2: Inlining through a non-tail call can introduce both local and non-local continuation throws.

The primary method by which our classification scheme goes awry is when inlining breaks our assumptions about throws to the current continuation (Figure 3.2). For example, when control-flow analysis determines that the throw to `fRet` in Figure 3.2a is to `retK`, the body of `retK` can be inlined at that site. After inlining has occurred (Figure 3.2b), outerRet now has two different kinds of throws according to our classification: line 9 is a local return, and line 2 is a non-local return.

Observe that `retK` contains a use of its current continuation, and if we follow `retK`’s def-use chain to the throw site on line 2, that chain passes through a non-tail call; namely, the apply of `g` on line 11. That function application would push another frame onto the stack, hiding the frame represented by outerRet unless we explicitly capture and pass it as an escape continuation. Instead of making the inliner smarter about this case when DS conversion is enabled, we simply disable the inlining of continuations in CPS, as DS conversion is only a means to an end. For safety, we also check to ensure that a non-local throw to a return-bound parameter does not appear after optimizations were applied.
3.3.3 Dealing with First-class Continuations

After classification has identified all second-class continuations, we must transform the program so that first-class continuations are captured appropriately (Figure 3.3).

3.3.4 Closure Conversion

Discuss free variables of the return cont, and how we map second-class conts to blocks instead of functions.

3.4 Contiguous Stack

Contiguous stacks are similar to the classic approach used in production compilers for C, where each function allocates a single frame for its entire lifetime. Stack overflow is detected by protecting the last page of the stack from memory access (Figure 3.4), which makes recovery in the event of overflow from deep recursion impractical (Section 2.1.2), in exchange for omitting limit tests in each function’s prologue.

3.4.1 Frame Management

In each function’s prologue, the stack pointer is adjusted to make space for stack saved values such that the bottom of the frame is 16-byte aligned. Alignment is necessary to be compatible with C foreign-function calls. Frame-pointer elimination is used throughout since the caller’s frame is adjacent in memory to the callee’s, and frame sizes are statically known.

Thus, only the stack pointer is bumped down in the prologue to establish an empty frame, and correspondingly it is bumped up before returning (Figure 3.5). The prologue also initializes a slot in each
; some_function:
; prologue
subq $SpillSz, %rsp
pushq $0 ; watermark
...
; epilogue
addq $(SpillSz+8), %rsp
retq

Figure 3.5: Prologue/epilogue for a contiguous stack.

; initialize return continuation for func1
movq %rax, 24(%rsp) ; slot 3
movq %rcx, 16(%rsp) ; slot 2
movq %r14, 8(%rsp) ; slot 1
callq _func1
movq 8(%rsp), %rdi ; use of val
; initialize return for func2
movq %rax, 8(%rsp) ; reuse slot
callq _func2
; reload live vals for use
movq 24(%rsp), %rax
movq 16(%rsp), %rcx

Figure 3.6: Example of frame sharing across multiple non-tail calls.

frame with a watermark value to support generational stack collection (Section 3.6). Callees do not preserve any registers because we have not implemented the corresponding callee-saved-register optimization for immutable heap frames (Section 3.2).

**Frame Sharing** The main benefit of using mutable continuations is the ability to reuse space allocated for a continuation when the continuation cannot be invoked more than once. Figure 3.6 shows an example of the code output by our framework to share a stack frame across multiple non-tail calls. Values that are live but unused for multiple calls are not reloaded until they are needed, in contrast to immutable frames where they must be copied to the next frame. The downside of frame sharing is that it complicates garbage collection (Sections 3.4.3 and 3.6).

### 3.4.2 Continuation Capture

Describe Figure 3.4 and also describe in prose what the Callec ASM is doing, referring back to the fact that calls to that code are inserted by the wrap-captures transform, and the role of the landing pad frame that determines how the captured frames making up the one-shot continuation should be treated: a regular return or a longjmp.
3.4.3 Space Complexity

Our solution for this problem is standard: we output stack layout information during compilation with LLVM, and then build a hash table for the garbage collector. The table is keyed on return addresses, associating with that address the size of the frame and the locations of live heap pointers in the frame at that point in the function.

Also all tail calls use constant space, and we guarantee tail calls are preserved in our direct-style transformation.

3.5 Segmented Stack

Fundamentally, a segmented stack is a contiguous stack broken into smaller segments that are linked together, with each segment providing space for multiple frames allocated contiguously, as in Section 3.4.1. Stack overflow is a recoverable event that allocates and links another stack segment (Section 3.5.1). This design allows for arbitrarily deep recursion at the expense of stack limit tests in the function prologue. Our segmented stack is implemented as described by Bruggeman et al. (1996), with a minor variation (Section 3.5.2).

3.5.1 Handling Overflow

Every function that utilizes a stack frame ensures that there is sufficient space in the current segment before allocating their frame. This check adds a few additional instructions to the function’s prologue (Figure 3.8). As an added optimization, there are 128 bytes of stack spillover, or slop, available past the limit of every segment (Figure 3.7). The slop area enables us to omit the add instruction in the stack limit instruction sequence, if the function’s stack frame is less than the slop size.

If there is insufficient space available in the current segment, growstack invokes the segment overflow handler. Handling overflow with segmented stacks almost identical to capturing an escape continuation, except that the continuation is effectively allocated in the segment descriptors rather than the heap.
The first step in the process of handling overflow is to obtain a fresh stack segment, either from the free list or newly allocated, and link it back to the filled segment. The fresh segment has an underflow handler frame installed at the bottom, which performs an continuation throw to the previous segment using the descriptors.

Then, we move a handful of the most recently allocated frames to the new segment (Figure 3.7b), as described by Bruggeman et al. (1996), to solve the bouncing problem cited by others (Anderson 2013; Randall 2013). Segment bouncing is more likely to occur without this trick, as the function that caused the overflow may immediately return and perhaps be called again, repeatedly, invoking the overflow/underflow handlers each time. Currently, we bound the amount of data moved to most four frames or less than half of the segment's size, whichever comes first. The segment is efficiently parsed, i.e., without the use of a lookup table, to find frames to move because every frame also includes its size (Figure 3.8).

Moving these frames is a challenge if pointers from the heap into the stack are allowed, because we would need to update them. One way to find these pointers without scanning the entire heap is to maintain a remembered set in each segment descriptor, adding to it whenever a pointer into the stack is created. In Manticore, this is unnecessary as there are no pointers into the stack, except in the case of continuation capture (Section 3.5.3). When this occurs, we simply seal off the captured segment without moving frames into the fresh segment.

### 3.5.2 Segment Allocation

Rather than allocating each segment in our normal heap, we allocate segments in their own non-moving, mark-sweep managed heap region (Section 3.6). The reason for this is two-fold.

The first reason is due the particular setup of our generational garbage collector. Bruggeman et al. (1996) strongly suggests the use of a cache of free stack segments to improve performance, but if the segments were kept in our heap, which is regularly compacted, the cache would also be regularly emptied. Emptying the cache places extra strain on the collector, as a burst of thread allocations following a collection could quickly fill the nursery with empty segments, triggering another collection to promote them.

The second reason is due to the difficulty of implementing a C foreign-function call. While a Manticore function would regularly check to ensure a segment does not overflow, a C function relies on a guard page. One solution is to change every C call so that it calls a special assembly routine that takes the C function...
pointer as an extra argument, and switches between a page-protected stack and the current segment before and after the call. While this should have only a minor overhead, this solution would be annoying to implement in practice, as the extra argument complicates calling conventions.

We instead take advantage of the plentiful virtual memory in modern systems. The page at the end of each segment is memory protected, and we also add a few kilobytes of foreign-function stack space past the segment’s limit (Figure 3.7a). This way, C calls can be safely performed at any point in a segment.

### 3.5.3 Continuation Capture

mostly the same as in contig stacks, except we seal the current segment.

### 3.6 Garbage Collection

In Manticore’s runtime system, a VProc, or virtual processor, corresponds to a single POSIX thread (Rainey 2007). Each VProc has its own private local heap and scheduling queues, which consist of lightweight Manticore threads, and access to a global heap shared with other VProcs. Local heaps are collected using Appel (1989)’s semi-generational collector, while the global heap uses a stop-the-world copy-collector (Auhagen et al. 2011). In place of a write barrier, transitive object promotion is used to maintain the property that there are no pointers from an older generation into a younger one. In total, there are three generations in which heap objects may reside, the youngest of which is where objects are initially allocated.

#### 3.6.1 High Watermarks

With immutable, heap-allocated frames, there is no extra effort needed to add generational stack scanning (Section 2.1.6) to a runtime system that already employs generational garbage collection. This is because the contents of each frame, and thus any live roots, will never change after being promoted to a later generation.

For continuation strategies that use mutable frames, we employ the use of a **watermark** in each stack frame to avoid excessive stack scanning. A watermark is an indicator shared between the mutator and garbage collector that represents the state of the frame and its predecessors. In our runtime system, there are three values a watermark can take on, one for each generation from youngest to oldest: nursery, major heap, global heap. Whenever a function is called, the mutator places a nursery watermark in its frame to indicate that the frame may contain pointers into the nursery (Figures 3.5 and 3.8).

During the collection of a generation, these watermarks are overwritten by the collector with the indicator corresponding to the generation that the frame’s pointers were forwarded to. A generation’s collector will stop scanning a stack once it sees a watermark that is older than the current generation. This point represents the **high watermark** of the current stack, i.e., the furthest point back where pointers in the current generation may reside, as all pointers behind it have already been forwarded. However, the collector must still check the first older frame encountered. This is because the mutator only pushes a nursery watermark when the frame is first setup. So, a function that completed multiple non-tail calls between collections, but did not yet return, will have an older watermark in its frame with roots that might be in a younger generation.

#### 3.6.2 Stack Cache

Stack segments and contiguous stacks are kept in a separate region of the heap that uses a non-moving, mark-sweep collection strategy. (Section 3.5.2). Each VProc maintains two lists: one consisting of all active allocations, and another for freed allocations for use as a cache whenever new stack allocations are needed.

By the nature of a generational heap, not all active stack allocations will be encountered during a collection cycle, which complicates the integration of a mark-sweep region. For example, continuations may be reachable from a channel object that is located in an older part of the heap, which is not traced on every cycle. To solve this, each stack allocation has an age field in its descriptor that is used to track the oldest generation of the heap in which a pointer to that allocation has ever existed. This field is used to determine
whether it is safe to free an unmarked stack after tracing only part of the heap. A collector for a given generation will only free an unmarked stack if its age is younger or equal to the current generation.
Analysis

plots go here
Chapter 5

Conclusions

words go here
Bibliography


