# 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.

# Contents

Chapter 1

# Overview

## 1.1 Introduction

Functional languages rely heavily on recursion and efficient function calls, making the implementation strategy used for such operations a major factor in program performance. A *call stack* is a data structure used as the backbone of languages that support functions,[1] which normally need to allocate memory for each invocation (Dijkstra 1960). In addition, a captured call stack is often used to represent a paused thread in languages that support concurrency. Thus, having extremely lightweight mechanisms to allocate new call stacks and switch between them is essential for the ease-of-use and scalablity of a concurrent language.

Unfortunately, it is not clear which implementation strategy for a call stack is best suited for a functional, concurrent language. 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.

But, sometimes the folklore is misleading! For example, MULTIMLTON and GUILE avoided the use of the segmented stack strategy in part because of reports from RUST and GO developers suggesting 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 for their SCHEME compiler. A subtle aspect of their solution is its effectiveness *only* for runtime systems that do not allow pointers into the stack, which is the norm for languages like ML and SCHEME. Thus, when weighing one strategy against another, a deep understanding of the design space is crucial. The purpose of this work is to help set the record straight by providing an empirical evaluation of different strategies, with well-normalized data collected from the same compiler.

## 1.2 Background

When entering a function, the top of the *call stack* represents the continuation of that invocation, *i.e.*, the context in which the function's result is needed. Concretely, the top of the

[1] Specifically, reentrant functions.

stack consists of a *frame* containing data, such as local variables and a code address, that belong to the function's caller. A function makes a *tail* call if the callee will return directly to the same context that the function itself would return to; otherwise it is a *non-tail* call. To put it another way: if the very last thing a function does is perform a function call, then that call is a tail call. Whenever a non-tail call occurs, a frame is pushed onto the call stack. Once the callee has finished, it uses the topmost frame to resume its caller with its result.

Given any point in the program, its *continuation* is the abstract notion of the "next step of computation" to be performed. A continuation represents this notion by capturing all of the values needed to continue execution, such as the call stack and other live values. For example, once one machine instruction has completed, the continuation of that instruction consists of the machine registers containing live values, the reachable memory, and the next instruction to be executed.

Thus, a new *return continuation* must be captured every time a non-tail call is encountered. This typically involves appending a new frame onto the call stack to store values needed in the call's continuation.[2] Return continuations are captured and used (thrown) very frequently in functional programs, which express loops with recursion. These continuations have a restricted, last-captured-first-thrown lifetime that enables many optimizations (Chapter 2).

### 1.2.1 Continuation-passing Style

It is difficult to precisely discuss function calls and other control-flow mechanisms without introducing a standard notation for them. We can rewrite a program in *continuation-passing style* (CPS) to make *all* control flow explicit using continuations. Consider the following factorial function written in CPS, where we will use STANDARD ML (SML) functions represent continuations.

```
fun factorial (n, k) =
  if n = 0
    then k 1
    else let
           fun retK x = k (n * x)
         in
           factorial (n-1, retK)
         end
```

The additional parameter `k` represents the return continuation, which is used by an invocation of `factorial` to return a result to its caller. When `n` is zero, the result of the then-expression is not just `1`, we *explicitly* return it by tail-calling (throwing) the return continuation `k` with the argument `1`. Informally, this is the regularization we are after by rewriting in CPS: *all* control-flow is made explicit through either function calls or continuation throws that are in tail position.

Before CPS conversion, when `n` is not zero, the else-expression would be `n * factorial (n-1)`. This recursive call is not in tail position, as its result is then multiplied by `n`. Thus, we rewrite it in CPS by introducing (capturing) a new return continuation, `retK`, that is passed in the recursive call, leaving the call in tail position. This new continuation captures

[2] Machine registers preserved by the callee may also be used to capture part of these continuations.

$$
\begin{array}{lcl}
\langle exp \rangle & ::= & \texttt{let}\ (x_1, ..., x_n) = \langle prim \rangle\ \texttt{in}\ \langle exp \rangle \\
& | & \texttt{fun}\ f\ (x_1, ..., x_n / k) = \langle exp \rangle\ \texttt{in}\ \langle exp \rangle \\
& | & \texttt{cont}\ k\ (x_1, ..., x_n) = \langle exp \rangle\ \texttt{in}\ \langle exp \rangle \\
& | & \texttt{if}\ x\ \texttt{then}\ \langle exp \rangle\ \texttt{else}\ \langle exp \rangle \\
& | & \texttt{apply}\ f\ (x_1, ..., x_n / k) \\
& | & \texttt{throw}\ k\ (x_1, ..., x_n) \\
\\
\langle prim \rangle & ::= & \text{primitive operations and values}
\end{array}
$$

Figure 1.1: A continuation-passing style intermediate representation.

the computation to be performed on the recursive call's result, which eventually returns to the caller via k.

**A CPS Intermediate Representation** One of the problems with our CPS factorial example is the lack of distinction between functions and continuations. Figure 1.1 shows a CPS intermediate representation (IR) that can better serve as a vehicle for performing optimizations and analysis within a compiler. Below, we have rewritten our factorial example to use the CPS IR instead.

```
fun factorial (n / k) =
  let isZero = n == 0 in
  if isZero
    then throw k (1)
    else cont retK (x) =
           let res = n * x in
           throw k (res)
         in
           let arg = n - 1 in
           apply factorial (arg / retK)
in
  ...
```

Now, we are in CPS and the distinction between continuations and functions is preserved. The slash used in the parameter list of `fun` expressions separates continuation parameters introduced by CPS conversion, from uses of *reified continuations* in the original program, which would appear as regular function arguments. Reified continuations are continuations exposed to the programmer as concrete values, which provides a powerful way for programmers to express advanced control-flow.

### 1.2.2 Types of Continuations

Reified continuations are typically classified based on their lifetime and allowed number of invocations, as these factors affect their implementation. If a continuation can be invoked at most once, it is known as a *one-shot* continuation. Otherwise, a *multi-shot* continuation can be invoked arbitrarily many times.

```
type 'a cont

val callec   : ('a cont -> 'a) -> 'a
val throw    : 'a cont -> 'a -> 'b
val newstack : ('a -> 'b) -> 'a cont
```

Figure 1.2: An interface for reified escape continuations.

```
val nums = ...
val ans =
    callec (fn bind => let
      fun prod [] = 1
        | prod (hd::tl) =
          if hd = 0
          then throw bind 0
          else hd * (prod tl)
    in
      prod nums
    end)
val next = ...
```

Figure 1.3: Using a continuation to exit recursion.

The function `callcc`, found in SCHEME and SML/NJ, produces a *first-class* continuation that has no restrictions: it is multi-shot and has an unlimited lifetime. All other continuations are *second-class* because they are restricted in at least one of these dimensions.

For example, the continuations produced by `call1cc` as described by Bruggeman et al. (1996) are one-shot, but have an arbitrary lifetime.[3] An *escape* continuation is a one-shot continuation whose lifetime is limited to its lexical scope (Fisher and Reppy 2002; Ramsey and Peyton Jones 2000). Escape continuations are simpler to implement due to their restrictions, yet still powerful enough to support concurrency.

### 1.2.3 Programming with Continuations

Programmers can use escape continuations as the singular building block for programs featuring advanced control-flow mechanisms. Figure 1.2 shows a simple interface in SML for writing programs that use such continuations. When called, the function `callec` captures the continuation of its call (as an escape continuation) and invokes the function argument it was given, passing the continuation to it. We use `throw` to invoke a continuation, which abandons the context in which `throw` is used and establishes the continuation's context. Thus, `throw` will never return a value, so it is safe to say that it can return anything.

In the example shown in Figure 1.3, `callec` is used to capture a continuation, `bind`, that binds `ans` to the value thrown to it, and then continues on with evaluating the right-hand side of `next`. We use the continuation `bind` to exit the computation of the product early, if one of the integers in the list `nums` is zero. Otherwise, when the initial call of `prod` completes, the result is returned normally by `callec`. While we could have encoded this type of control-flow using the exceptions built into SML, continuations can fully implement exceptions and much more.

[3] While the return from `call1cc` is itself a use of the captured continuation, this is not enough to restrict its lifetime: nested usage of `call1cc` allows for the return to be skipped.

```
type thread = unit cont
type queue = thread list ref

(* queue actions *)
val globalQueue : queue
val addThread : thread -> unit
val tryRemove : unit -> thread option

val newThread : (unit -> unit) -> thread

(* scheduler actions *)
val exit : unit -> 'a
val yield : unit -> unit
```

Figure 1.4: A userspace threading library interface.

**Concurrency** The ability to create and control threads of computation provides a basis for writing programs with explicit concurrency. Continuations can naturally represent a paused thread, which enables programmers to implement threading as a userspace library (Ramsey 1990; Reppy 1991). While this application of continuations to concurrent systems is certainly not a new idea (Wand 1980), we will illustrate it with an example.

Figure 1.4 shows an example threading library interface that can be implemented using just the functions in Figure 1.3. All functions in this example operate on the same work queue, `globalQueue`. The `yield` function (Figure 1.5) enables cooperative scheduling among threads in the `globalQueue`. When invoked, `yield` checks to see if other work is available, and if so, it captures the continuation of its caller, places that paused thread on the work queue, and then dispatches the other thread using `throw`.

```
fun yield () =
  case tryRemove ()
    of NONE => ()
     | SOME thd =>
         callec (fn self =>
           addThread self;
           throw thd ()
         )
```

Figure 1.5: Cooperative scheduling with continuations.

```
fun newThread f =
  newstack(fn () => let
      val () = f ()
    in
      exit ()
    end)
```

Figure 1.6: Creating a new thread with `newstack`.

Any continuation produced with `callec` is a prefix of some existing continuation. To spawn new paused threads in the function `newThread` (Figure 1.6), we use `newstack` to allocate a brand new continuation that will execute the given function in the new context. In `newThread`, we wrap the function `f` so that, when it completes, the thread performs an `exit`, which discards its continuation and dispatches the next thread in the queue.

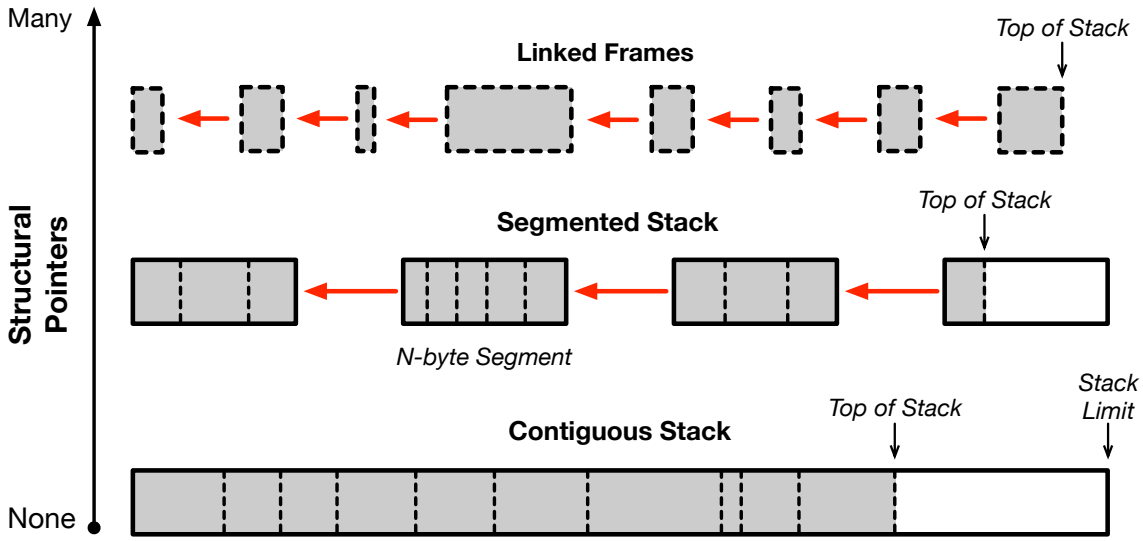


Figure 1.7: High-level spectrum of call stack implementations, as a function of the number of pointers used to maintain the structure. Dashed lines separate stack frames.

### 1.2.4 Manticore

Manticore is an optimizing compiler system that implements PARALLEL ML (PML) (Fluet et al. 2008b), and serves as our common ground in exploring strategies for implementing continuations. PML's sequential language is a subset of SMLwithout mutation, and features parallelism in the form of explicit and implicit threading. Explicit threading is available through the ability to spawn lightweight threads that communicate over typed, synchronous channels *à la* CONCURRENT ML (CML) (Reppy 1991). All threading features are implemented by Manticore using continuations (Chapter 3).

## 1.3 Implementation Strategies

When capturing a continuation for a non-tail call, a frame is typically pushed, or linked, onto the top of the call stack. The memory structure of call stacks is characterized by how often pointers are used to link frames together (Figure 1.7).

At one end of the spectrum, the traditional contiguous stack requires no inter-frame pointers if the frame sizes are statically known[4], 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 allocate smaller segments of memory at a time, linking each *segment* together with pointers (Bruggeman et al. 1996). Frames within a segment are accessed as in a contiguous stack, and the inter-segment pointers are only used when a segment underflows or overflows. Thus, as the segment size shrinks, segment pointer

[4] Also known as *frame pointer elimination*.

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.

**Trade-offs** A major concern for compiler designers is the efficiency of the call stack for sequential programs. Reusing memory allocated for stack frames is often seen as a benefit for performance, as it would increase cache locality and reduce garbage collector burden. But, reusing stack frames increases runtime system complexity, and makes first-class continuations expensive to use. In a parallel runtime system, a locking mechanism is also required when exchanging continuations on a scheduling queue, such as during a `yield` (Figure 1.5). Thus, immutable stack frames have desirable characteristics, but the performance trade-off is not well understood.

Another concern that effects users of a compiler is a limit on recursion depth. Contiguous stack implementations often reserve a fixed-size stack area which has no way to expand in the event of overflow. This design choice removes the burden of checking for stack overflow and updating pointers into the stack during a relocation. When stacks are allocated in the heap, recursion is no longer limited in depth by the size of the reserved stack space, but can use all memory available to the process.

### 1.3.1 Prior Work

Appel and Shao (1996) compared the cost of using immutable heap-allocated frames (Section 3.4) against a typical 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 produced using a modified compiler that measured cache effects and instruction counts.

In Appel and Shao (1996)'s cost model, all instructions were assumed to have the same basic cost of 1 cycle. They also simulated a direct-mapped cache with a 10-cycle read-miss penalty, no write miss penalty, and an infinite instruction cache. The main weakness in their simulation of stack behavior was inflated instruction counts for frame initialization in all benchmarks due to lack of frame reuse for non-tail calls (see Clinger et al. 1999, Section 7). In addition, while their cost model may have been sufficient for processors at the time, the performance of modern superscalars is more difficult to estimate.

Clinger et al. (1988) extended the MacScheme+Toolsmith compiler with support for various implementations of first-class continuation designs. Each design was tested by modifying the 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 first-class continuations are not used, and that immutable heap-allocated frames perform very poorly when the heap size is small. But first-class continuations are known to be overly taxing when stacks are used for concurrency because of the additional copying that is required (Bruggeman et al. 1996).

Bruggeman et al. (1996) put forth the use of cheaper one-shot continuations, with a segmented-stack design, and compared them against immutable heap-allocated frames using only a synthetic workload. Their performance evaluation varied the number of function calls per context switch, and reported that the heap-allocated frames are only a win

when the switching rate is more frequent than once every four function calls.

Fisher and Reppy (2002); Ramsey and Peyton Jones (2000) suggested the use of escape continuations for extremely cheap context switches when continuations are used for concurrency. But, neither of those works include an evaluation, and we are not aware of any other comparison of strategies for implementing escape continuations.

**Allocation Characteristics** A number of analyses have focused on the efficiency of frame allocation and reuse for call stacks. Explicitly managing the allocation of frames in a contiguous "stack region" (*e.g.*, reusing recently popped frames first) is commonly seen as a technique that benefits performance in two ways: (1) improved cache locality and (2) reduced garbage collector burden because frames are kept out of the heap.

Gonçalves and Appel (1995) show that most frame reads are performed very soon after allocating frames, and thus would still be in the cache regardless of whether the frame was allocated in the heap or not. 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) found that the regular compaction offered by such a nursery is also beneficial to cache locality verses other schemes for heap allocation, but it is unclear whether this benefit can match the efficiency of stack allocation.

## 1.4 This Work

The process of capturing a reified continuation depends on the continuation's lifetime and number of uses (Section 1.2). If stack frames are reused, capturing a first-class continuation requires making a copy of the call stack, either at the time of capture or when it is thrown (Clinger et al. 1999). This makes first-class continuations nonviable for heavy-duty applications such as concurrency. Nearly all of the prior work comparing implementations of reified continuations have focused on first-class continuations.

Capturing an escape continuation is much cheaper than a first-class continuation in the presence of frame reuse, since copying is not necessary, and escape continuations can be used to implement concurrency (Section 1.2.3). Yet, what remains is the question of *which* implementation of escape continuations should be used. There are a number of options (Section 1.3), each of which have side effects, good and bad, on the rest of the compiler: performance, complexity, and flexibility.

The goal of this work is to provide an empirical analysis of implementations of escape continuations in order to make the trade-offs clear for compiler writers. To do this, we extended the Manticore compiler and runtime system with support for multiple implementations of such continuations (Chapter 3). To our knowledge, this is the first head-to-head comparison of implementations for lightweight escape continuations. In addition, we provide a high-level review of the essential implementation optimizations and issues regarding continuations (Chapter 2).

Overall, we are interested not only in the performance of continuations under sequential and concurrent workloads, but in qualities such as support for deep recursion and implementation complexity (Chapter 5).

Chapter 2

# Issues and Optimizations

It is difficult to compare implementation techniques for continuations without understanding the essential challenges they solve. In this chapter, we highlight the most common challenges faced by implementations of continuations, with an added focus on functional languages that rely on garbage collection.

## 2.1 Overflow Detection

Functional programs often make use of deep recursion, which may trigger a *stack overflow* if the area in which stack frames are allocated has been exhausted. The two primary methods of detecting overflow when trying to allocate a frame are memory faults (via page protection) and explicit limit checks.

**Memory Faults** Oftentimes, a special region of memory is reserved exclusively for allocating stack frames. Placing an access protected *guard page* at the end of this memory region is a classic approach to detecting overflow without explicitly checking for it. The downside is that *any* instruction that accesses a stack frame can trigger the memory fault interrupt that is interpreted as a stack overflow. For example, this interrupt could occur while spilling a register to the frame during a heap object's initialization, which could leave the heap in an indeterminate state.

Thus, heap allocations must be viewed as an atomic operation in order to be safe in the presence of interrupts. Shivers et al. (1999) describe a number of approaches for implementing heap allocations as lightweight transactions that can be, for example, run until completion or restarted if an interrupt occurs. The registers and stack frame locations that contain live heap pointers must also be decipherable when handling the interrupt (Section 2.4).

Instead of trying to recover from a fault-triggered overflow, many language implementations simply crash the program. 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.

**Limit Checks** Since we can determine the amount of stack space required for each function, one check for available space at function entry is sufficient to detect overflow. These stack-limit checks serve as safe points, allowing the garbage collector to avoid internal fragmentation by growing and shrinking stack areas on demand.

Of course, limit checks come at the cost of executing a compare-and-branch instruction sequence for each function call. If stack frames are allocated directly in the heap, these stack-limit checks can be combined with limit checks for other heap allocations in the function.

## 2.2 Reducing Frame Allocation

A *capture-free function* is a function that contains no calls other than tail calls. An important subset of capture-free functions are *leaf functions*, which are functions that contain no function calls. Most function calls in a program are to leaf functions (Appel 1998), so they are crucial to optimize.

Capture-free functions can often omit frame allocation because they do not capture the continuation of a call; they only pass or use the continuation of their caller. The presence of register spills or passing arguments in memory can prevent this optimization. But, if all callers of a capture-free function guarantee that there is free space associated with its frame,[1] known as a *red zone*, the callee can use the caller's frame for register spills. Thus, a red zone allows a capture-free function *with* register spills to avoid allocating a frame.

**Register Conventions** *Callee-saved registers* (CSRs) are an optimistic register convention for function calls. The caller of a function can leave live values in registers, allowing the callee to save them to its frame only when those registers are needed. The hope is that the callee and its descendants, particularly leaf functions, do not make use of those registers, so they will remain in register until the callee returns. If any of the descendants *do* need those registers, there is no longer a benefit as they will still be saved to a stack frame.

When CSRs are used anywhere within a function, one approach to preserving the registers is to spill and restore them at the entry and exits of a function. Chow (1988) describes a more precise technique that saves and restores CSRs within a smaller subset of the function's control-flow paths. Overall, the main downside to using CSRs is that the source of callee-saved values are dynamic during execution, which complicates garbage-collected runtime systems (Section 2.4).

## 2.3 Optimizing Frame Usage

Since each non-tail call within a function is a continuation capture, a frame is needed for each call. We could allocate a fresh, immutable frame for each of these captures. But once a call returns, if the call's continuation was not reified, the space occupied by the frame for that call is exclusively owned by the caller. This means that the function can reclaim, or *reuse*, the frame's space for the next call's frame.[2]

A compiler can optimize the initialization of a frame by taking into account the values already saved in the previously allocated frame. This optimization is called *frame sharing*.

[1] For example, a contiguous stack can reserve free space beyond the stack pointer for the callee's use (Matz et al. 2012).

[2] Sobel and Friedman (1998) explore a fun alternative for reusing frames, however, performance gains seem unlikely.

For example, if a value is live but unused across multiple non-tail calls, the value can be allocated to a slot in the first frame, and left intact if the frame is reused. Without frame sharing, the value would be copied out of the frame and into the next, using arbitrary slots. Frame sharing with reuse can be treated as a form of the register allocation problem, *e.g.*, values with disjoint lifetimes can be assigned to the same slot to minimize frame size.

Even if frames are not reused, it is possible to benefit from frame sharing. Consider the case of several values simultaneously live across multiple non-tail calls. A pointer to an object containing these live values can be copied to the freshly allocated frames for those calls, reducing the copying overhead imposed by not reusing frames. When using function closures to represent frames, this sharing optimization can be achieved, while preserving space complexity (Section 2.4), by using *safely linked closures* (Shao and Appel 2000).

## 2.4 Finding Roots

Accurately identifying live heap pointers, or *roots*, in the stack frame and machine registers is essential in a garbage collected runtime system. One approach for identifying roots is to segregate them by using two stacks and two sets of machine registers: one for pointers and the other for non-pointers. But, adjusting and performing limit tests for two different stacks adds mutator overhead (Peyton Jones and Salkild 1989). Instead, uniformly sized frame slots, plus either a tag bit per value or a parsable layout word per frame, is sufficient to distinguish pointers from non-pointers in the frame.

A straightforward approach for identifying roots is to generate a map containing location data at call sites,[3] using the stack frame's return address as the key (Diwan et al. 1992). This allows the garbage collector to lookup a rich description of the frame slots and other information while parsing a call stack. The map can be implemented using a hash table, or spatially by placing location data just before the instruction pointed to by the return address (GHC Team 2006, see Info Tables). It is unclear which combination of the above approaches is most efficient.

**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 use callee-saved registers for non-pointer values.

**Space Complexity** Frame sharing with reuse (Section 2.3) complicates the precise communication of the live roots in a frame to the garbage collector. We must prevent the garbage collector from treating a leftover, dead pointer as a live root, because it would cause a leak that can change the space complexity of the program (Appel 1992).

[3] It is also possible to generate location data for *every* instruction using clever data compression techniques (Stichnoth et al. 1999).

Consider, for example, the bit-tagging scheme for stack frames described earlier. To prevent space leaks, it would be necessary to overwrite each pointer in the frame as they become dead. In comparison, when using a layout word or a return-address map, at most one write to the frame is required per call.

## 2.5 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 3.2.4 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.

## 2.6 CPU and ABI Support

Historically, instruction sets such as the x86 have contained dedicated instructions to assist with the allocation of stack frames. Some CPUs use hardware in the instruction pipeline to reduce the cost of adjusting the stack pointer register and increase branch prediction rates for call and return (Gochman et al. 2003). Pettersson et al. (2002) tested the effectiveness of this type of branch prediction on the x86-32 by modifying function calls so they are emitted as a `push` followed by a `jump` instead of using the `call` instruction, which would activate the return-branch predictor. They found that without the use of `call`, overall performance of their benchmark suite dropped by 9.2%, with some call-heavy programs losing 20-30%.

The combination of dedicated stack instructions and the operating system's *application binary interface* (ABI) constrains the implementation of continuations. Many foreign function ABIs expect a guard page to detect stack overflow (Section 2.1), thus passing a stack pointer that is in the middle of a heap-allocated stack to a foreign function is unsafe without the addition of a guard page in the heap. If stacks in the heap are also relocated, the garbage collector must use system calls to enable and disable the guard pages. An alternative is to use *stack switching* for each foreign-function call, incurring a few instructions per call to swap a different stack into the stack pointer register.

## 2.7 Concurrency

Due to the lifetime restrictions of a captured escape continuation (Section 1.2), it is not possible to implement `newThread` (Figure 1.4) with such captures. Instead, to create a disjoint execution context, we take `newThread` as a primitive operation that is implemented by the

runtime system as follows. First, a call stack is allocated, consisting of one frame that will execute the `exit` routine. Then, pushed onto this stack is a frame that will apply the given function `f` to unit. Now, the top of the stack is a primed-and-ready paused thread that executes `f`.

A hiccup occurs with frame reuse the presence of parallelism. Suppose we would like to yield the current thread to other work waiting in a scheduling queue that is shared among multiple processors. After processor $P$ pauses the current thread by capturing an escape continuation, it places the thread on the queue and then removes a thread to dispatch it. This processor's exchange on the scheduling queue is at risk of a race, as it is not finished using the call stack of the current thread until it has dispatched another thread. Thus, if some processor $Q$ dispatches the thread from $P$ too early, it can end up overwriting frames still needed by $P$. To solve this, Fisher and Reppy (2002) use a lock field on captured continuations to protect those frames.

# Chapter 3

# Manticore Implementation

> "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 detail the extensions made to the Manticore system to evaluate different strategies for implementing escape continuations.

## 3.1 Overview

One of the key problems with Appel and Shao (1996)'s study is the lack of precision in the cost model that simulates the execution of programs using stack (or heap) allocated call stacks (Section 1.3.1). We take a different approach in that the Manticore compiler and runtime system now offers *real* support for compiling programs that use stack (or heap) allocated call stacks.

Manticore uses a CPS IR in its middle end for optimizations and code generation (Figure 1.1). To produce efficient stack-allocated continuations, Manticore uses a *direct-style* (DS) transformation (Chapter 4) to undo CPS conversion before targeting a modified version of the LLVM compiler toolchain (Lattner and Adve 2004). LLVM already produces code that uses the stack efficiently; our minor extensions to LLVM improve its support for garbage-collected languages.

The continuation strategies supported by Manticore are:

- Contiguous Stack (Section 3.2)
- Segmented Stack (Section 3.3)
- Immutable Heap Frames (Section 3.4)

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

## 3.2 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. The frame is reused

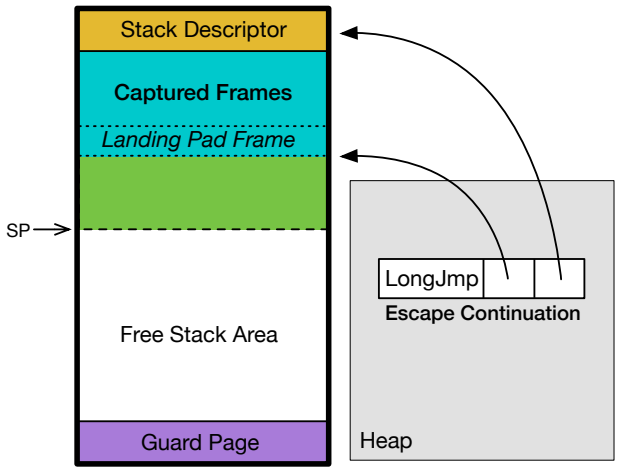


Figure 3.1: Layout of a contiguous stack, with an escape continuation in the heap.

```
_some_function:
  ; prologue
  subq    $SpillSz, %rsp
  pushq   $0    ; watermark

  ...

  ; epilogue
  addq    $(SpillSz+8), %rsp
  retq
```

Figure 3.2: Prologue/epilogue for a contiguous stack.

by every non-tail call in the function, and objects allocated in the frame are shared across multiple calls. Stack overflow is detected by protecting the last page of the stack from memory access (Figure 3.1), which makes recovery in the event of overflow from deep recursion impractical (Section 2.1), in exchange for omitting limit tests in each function's prologue.

### 3.2.1 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 3.2). The prologue also initializes a slot in each frame with a *watermark* value to support generational stack collection (Section 3.2.4). Callees do not preserve any registers because we have not implemented the corresponding callee-saved-register optimization for immutable heap frames (Section 3.4).

### 3.2.2 Escape Continuations

Escape continuation capture is implemented using a short runtime system routine. This assembly routine allocates a small object in the heap containing the stack pointer, the stack's descriptor, and the address of a routine, `LongJmp`, that performs a continuation throw (Figure 3.1). The object represents an escape continuation and is similar to a function closure, but the garbage collector must treat it specially (Section 3.3.3). When the continuation is invoked, `LongJmp` simply changes the current stack pointer to the one saved in the object

and performs a return, indicating that the return is from an escape throw.

### 3.2.3 Space Complexity

In order to preserve space complexity in the presence of frame reuse and sharing (Section 2.3), we output stack layout information during compilation 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. All tail calls use constant stack space and are preserved by our direct-style transformation.

### 3.2.4 Garbage Collection

In Manticore's runtime system, a `VProc`, or virtual processor, corresponds to a single POSIX thread (Fluet et al. 2008a). The garbage collector is modeled after the Doligez-Leroy-Gonthier parallel collector, with each `VProc` assigned its own local heap and there is one global heap shared among all `VProcs` (Doligez and Leroy 1993; Doligez and Gonthier 1994). Local heaps are collected using Appel (1989)'s semi-generational collector, while the global heap uses a parallel 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.

**High-water Marks** With immutable, heap-allocated frames, there is no extra effort needed to add generational stack scanning (Section 2.5) 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 implementation strategies that reuse frames, a *watermark* is placed 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 the frame established by that function, indicating that the frame may contain pointers into the nursery (Figures 3.2 and 3.4).

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 being reclaimed. 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.

Critically, the collector must still scan the first older frame that is encountered. This is because the mutator only adds a fresh nursery watermark *when the frame is first setup*. So, a function that has completed multiple non-tail calls between collections, but did not yet

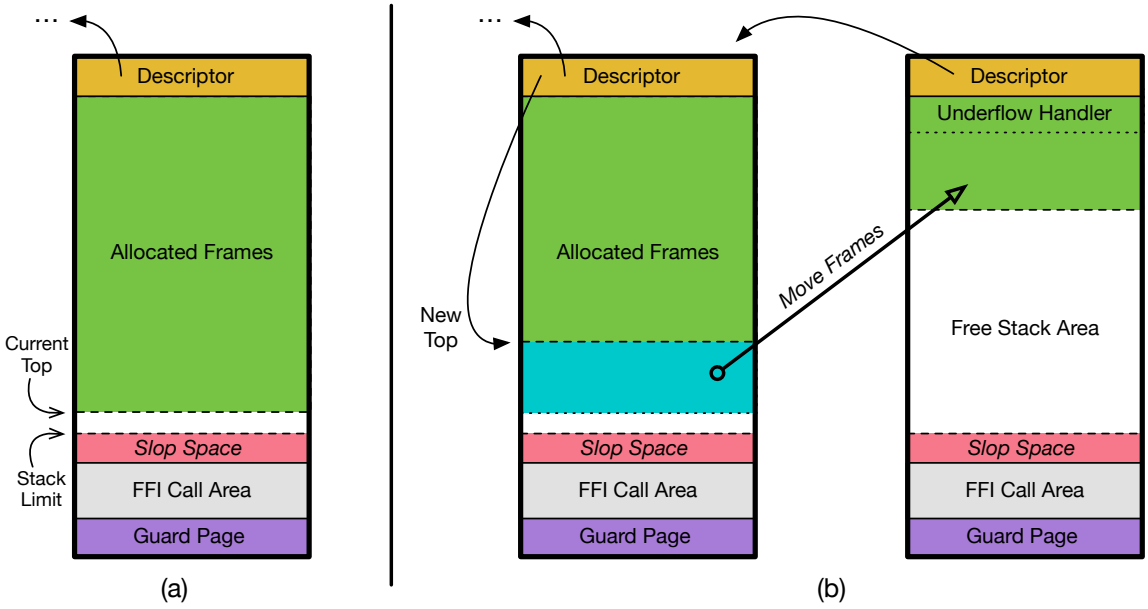


Figure 3.3: A stack segment has insufficient space in (a). Then, in (b) the overflow handler moves a bounded number frames to a new segment to avoid bouncing.

return, will have an older watermark in its frame, with roots that might be in a younger generation.

## 3.3 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 3.2.1. Stack overflow is a recoverable event that allocates and links another stack segment (Section 3.3.1). 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 3.3.2).

### 3.3.1 Overflow and Continuation Capture

**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 3.4). As an added optimization, there are 128 bytes of stack spillover, or *slop*, available past the limit of every segment (Figure 3.3). 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. 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

```
_no_slop_optimization:
  movq    120(%r11), %rbp
  addq    $168, %rbp
  cmpq    %rbp, %rsp
  jge     allocFrame_1
  callq   ___manti_growstack    ; overflow handler
allocFrame_1:
  subq    $152, %rsp    ; spill area
  pushq   $168          ; total frame size
  pushq   $0            ; watermark
  ; ... body ...


_with_slop_optimization:
  cmpq    120(%r11), %rsp
  jge     allocFrame_2
  callq   ___manti_growstack
allocFrame_2:
  subq    $56, %rsp
  pushq   $72
  pushq   $0
  ; ... body ...
```

Figure 3.4: Example function prologues for a segmented stack.

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 3.3b), 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 3.4).

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. Thus, Bruggeman et al. (1996)'s solution to segment bouncing seems impossible to implement in RUST and impractical in GO (Section 1.1). In Manticore, there is no challenge when moving frames since there are no pointers into the middle of a stack.

**Continuation Capture** Capturing an escape continuation is almost identical to handling overflow. For overflow, the stack pointer is saved in a segment descriptor instead of a heap object, and the underflow handler corresponds to `LongJmp`.

The major difference is that frames are *not* moved from the old segment to the new one,

to avoid having pointers into the middle of a stack. Otherwise, a heap object is allocated to represent the escape continuation as is done for a contiguous stack (Section 3.2.2).

### 3.3.2 Segment Allocation

Rather than allocating each segment in our normal heap, we allocate segments in their own non-moving, mark-sweep managed heap region. Note that frames *within* a segment are still moved, but not the segment itself. The reason for this design 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 if segments are kept in a moving heap. 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 emitted so that we switch to a page-protected stack immediately before the call, and right afterwards we switch back to the current stack segment. While this should have only a minor overhead, this solution would be annoying to implement in practice, especially in LLVM.

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 3.3a). This way, C calls can be safely performed at any point in a segment.

### 3.3.3 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 3.3.2). 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. This is because a `VProc`'s scheduling queue does not reach every live continuation, as some may be stored in an object that is located in an older part of the heap.

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 reclaim a stack after tracing only part of the heap. During the sweep phase, the collector for a given generation will only reclaim an unmarked stack if its age is younger or equal to the current generation.

## 3.4 Immutable Heap Frames

Some compilers, *e.g.*, SML/NJ, use ordinary, immutable function closures to represent frames of the call stack (Appel and Jim 1989). Thus, the closure conversion strategy plays an important rôle in determining the structure and efficiency of such a stack.

A *flat closure* (Cardelli 1983) is an array consisting of a function pointer and the free variables of that function. If part of the call stack, the function is the point of return for a call, and a pointer to the previous frame is one of its free variables. Thus, if flat closures are to represent frames of the call stack, its structure is equivalent to linking individual frames together with pointers. Shao and Appel (2000) discuss a more sophisticated closure strategy, *safely linked closures*, that yields significant efficiency gains over flat closures (Section 2.3). In Appel and Shao (1996)'s evaluation of call stacks, their heap strategy corresponds to using safely linked closures for continuations.

Manticore uses comparatively less efficient flat closures, which exhibit worst-case performance for a sequence of non-tail calls. For continuations that are bound and used locally within a function, also called a *join continuation*, closure capture and dispatch is not used. Instead, throwing to a join continuation is the same as a function-local jump, or *goto*, and the closure's free variables are passed as additional parameters. All continuations captured using this strategy, including those for standard function calls, are immutable and thus could be used as a first-class continuation.

Chapter 4

# Direct-style Conversion

It is possible to extend the closure conversion described in Section 3.4 so that stack frames are reused and shared across non-tail calls within a function (Adams et al. 1986). However, this optimization alone would not produce the type of code typically output by compilers that stack-allocate frames. For example, CPU-optimized stack manipulation instructions (Gochman et al. 2003) that incrementally build and modify each frame are normally used.

It is not possible for LLVM to stack-allocate continuations while the program is in CPS, as every call is in tail position. Thus, we extended Manticore to *undo* CPS conversion during closure conversion, in order to use LLVM as a baseline for stack-based continuations. While converting a program *to* CPS is widely known, comparatively less has been said about converting CPS programs *back* to their original form, direct style (DS).

In this section, we describe a practical application of direct-style conversion in a CPS-based compiler to target LLVM.

## 4.1 Overview

Before diving into direct-style conversion, we briefly discuss the compilation steps and intermediate representations (IRs) used in the Manticore compiler (Figure 4.1). The BOM IR is a direct-style $\lambda$-calculus in A-normal form (Flanagan et al. 1993), with support for reified continuations. Higher-level concurrency features in PML are implemented using continuations in BOM. Figures 1.1 and 4.2 show the relevant parts of the CPS and CFG IRs.

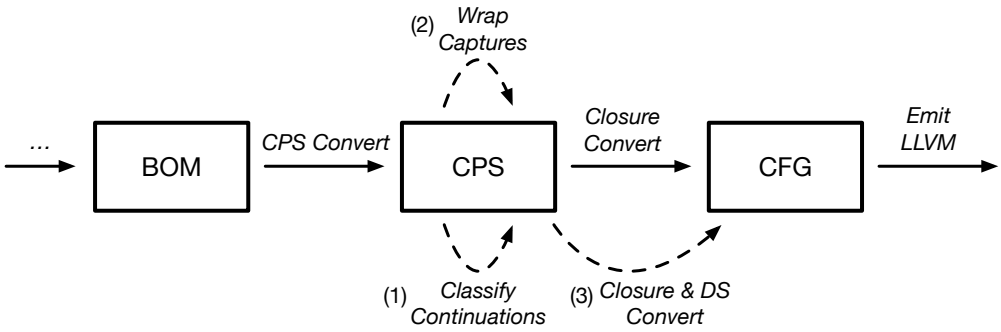


Figure 4.1: Manticore's compilation steps. Dashed arrows are involved in direct-style conversion.

$$
\begin{array}{lll}
\langle prog \rangle & ::= & \langle func \rangle + \\
\langle func \rangle & ::= & f\ \{\ \langle blk \rangle +\ \} \\
\langle blk \rangle & ::= & b\ (x_1, ..., x_n) = \langle stmt \rangle *\ \langle transfer \rangle \\
\langle transfer \rangle & ::= & \texttt{goto}\ b\ (x_1, ..., x_n) \\
 & | & \texttt{if}\ x\ \texttt{then}\ b\ (x_1, ..., x_n)\ \texttt{else}\ b'\ (y_1, ..., y_m) \\
 & | & \texttt{tailcall}\ f\ (x_1, ..., x_n) \\
 & | & \texttt{call}\ (r_1, ..., r_m) = f\ (x_1, ..., x_n)\ \texttt{then goto}\ b\ (y_1, ..., y_p) \\
 & | & \texttt{return}\ (x_1, ..., x_n) \\
 & | & \texttt{throw}\ k\ (x_1, ..., x_n) \\
 & | & ... \\
\langle stmt \rangle & ::= & \text{primitive operations}
\end{array}
$$

Figure 4.2: Manticore's control-flow graph (CFG) intermediate representation.

During CPS conversion, we separate the additional continuation parameters that are added to each function and include their kind (*e.g.*, an exception handler). After optimizations have been applied to the CPS program, direct-style conversion is performed.

The three-step DS conversion process is a combination of ideas from Reppy (2002); Kelsey (1995); Danvy and Lawall (1992). First, an analysis pass leveraging information retained by CPS conversion classifies continuations (Section 4.2). Then, a transformation pass introduces **`callec`** expressions (Figure 1.1) for reified continuations (Section 4.3). Finally, an alternate closure conversion pass converts the CPS program to a direct-style CFG program (Section 4.4).

## 4.2 Classifying Continuations

In our first step of direct-style conversion, we run an analysis pass over the CPS program (Figure 1.1) to assign a classification to continuation bindings (*i.e.*, **`cont`** expressions) and their uses. We define the *function context* of a non-function expression to be the innermost function in which the expression resides. The *current continuation* is the continuation bound as a parameter of the function context that is used for local returns. A continuation is an *escaping value* if it is saved to memory, or passed as a value argument (instead of a return 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 4.2.1.** (Second-class Restriction)

1. All uses of the continuation must occur in the function 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 reified continuations that are used in a first-class way.

During the analysis, we further distinguish the kind of second-class continuation bindings. A second-class continuation is a *return continuation* if it appears as such a continuation argument in at least one **`apply`** expression. Otherwise, the second-class continuation is called a *join continuation*. Undoing CPS conversion is essentially turning all return continuations into join continuations.

**Taming CPS Optimizations** Because we are performing direct-style conversion *after* CPS optimizations have been applied to the program, assumptions about the well-behavedness of the current continuation may not hold. In particular, CPS IR inliner can introduce situations where a return continuation is both returned to normally with a local throw, and thrown to as an escape continuation in a non-local throw.

```
fun f (x / retK1) =
  cont k () = throw retK1 ()
  in
    fun g (y / retK2) = throw k ()
    in
     ...
```

In the example above, it makes sense to optimize the throw to `k` by inlining its body, but in doing so we would introduce a use of the parameter-bound return continuation `retK1` inside of the function context `g`, which turns `retK1` into a non-local return. This transformation makes it harder to analyze and perform direct-style conversion.

Instead of making the inliner smarter about this situation, we avoided the problem by disabling the inliner's ability to inline continuation throws, whether stack-allocated continuations are desired or not. In future work, we plan to extend the inliner to include additional information about which continuations were bound locally within the function context of the throw. Then, for stack-based strategies it will only be valid to inline a continuation throw whose target is a locally-bound continuation.

## 4.3 Dealing with Escape Continuations

After classification has identified all second-class continuations, we transform the program so that reified escape continuations are captured efficiently in LLVM. The difficulty in LLVM is that we cannot efficiently capture an optional stack frame.[1] Thus, we implement escape continuation capture as a call to a special function, following the design of `setjmp` and `callcc` in other languages. These special calls do not exist in the CPS program, as we use the continuation binder **`cont`** to define continuations explicitly. Thus, we transform the CPS program by wrapping escape continuation captures using a new construct **`callec`** `(f / k)`, which represents the application of function `f` to the reified return continuation `k`.

[1] While coroutine support is currently in development for LLVM (Nishanov 2016), its compatibility with this work is unknown.

```
(*** before transform ***)
fun outerF(.. / retK) =
  ...
  cont k (x) = B  (* <- an escape continuation *)
  in
    C

⟹

(*** after transform ***)
fun outerF (.. / retK) =
  ...
  cont k (x) = B  (* <- now a join continuation! *)
  in
    cont landingPad (dispatchCode : int, arg : any) =
      if dispatchCode == REGULAR_RET
        then throw retK arg
        else throw k arg
    in
      fun manipK (k' : cont(any) / landingPad' : cont(int, any)) =
        cont manipRetk (x : any) =
          throw landingPad' (REGULAR_RET, x)
        in
          {C | k ↦ k' and retK ↦ manipRetk}
      in
      callec (manipK / landingPad)
```

Figure 4.3: Introduction of `callec` to capture an escape continuation when using LLVM.

In essence, the transformation is demoting all escape continuation definitions so that they are now join continuations. In the rewrite example shown in Figure 4.3, `k` is the escape continuation to be demoted, and `retK` is the current continuation relative to `k`.

To demote `k` to a join continuation, we introduce a new function `manipK` that receives the escape continuation `k` as the parameter `k'`. Then, we introduce a new return continuation, `landingPad`, that acts as a switch to determine which continuation of `outerF` to invoke with the given argument. We generate the switch by inspecting the free-variables of `C` for continuation uses and replacing them as-needed to pass through the landing pad with its appropriate dispatch code. This landing pad acts just like the point at which a `setjmp` returns to determine whether its returning from a `longjmp` or the initial call, but handles multiple continuation scenarios.

There are some type-signature changes needed to unify the argument types of `retK` and `k`. The type system of the CPS IR has a polymorpic `any` type that we use to pass any single uniform value through to the landing pad, and then cast the argument back once the dispatch code is inspected. For continuations taking multiple arguments, one could pad out the argument types of those continuations and pass unit in place of a missing value, but we did not need to do this since such continuations are rare in our compiler.

## 4.4 Closure Conversion

Flat closure conversion mostly proceeds as usual, with some exceptions. Join continuations *and* return continuations turned into basic blocks, since continuations will no longer be explicitly passed to the callee.

Each **apply** expression is transformed differently, depending on whether the function call is a tail call or not. While every function application appears in tail position in CPS, and would normally turn into a **tailcall** in CFG, these calls are not necessarily tail calls (Section 1.2). In particular, only an **apply** that is given the current continuation as the return-continuation argument is a tail call; otherwise, the **apply** is a non-tail call. The **callec** expression also represents a non-tail call, however, the function called is a special runtime system function (Section 3.2.2).

Consider the non-tail call **apply** $f\ (x/retK)$. For immutable heap frames, we would allocate a function closure that contains the free variables and code address of `retK`, and pass the closure to $f$. We are forced to determine how to preserve those values by the nature of a continuation-passing style program. Thus, to undo the CPS call, we simply choose to *not* explicitly capture or otherwise define how to save and restore those free variables. Instead, we allow those values to remain live across a CFG **call** transfer (Figure 4.2). The free variables, along with any returned values and arguments, are passed to the block representing `retK` in the sequel of the **call** as a function-local jump. LLVM will later decide how to stack-allocate the frame needed for the values in the sequel when performing the call to $f$.

When a CPS **throw** is transformed, we inspect the kind of continuation being thrown. A CFG **return** to the function's caller is emitted only if the throw is to the current continuation. Any other second-class throw, to either a return or join continuation, is simply a function-local jump that does not change the current continuation. A throw to an escape continuation becomes the only situation where a CFG **throw** is still emitted. These CFG throws eventually turn into a tail call that invokes a routine corresponding to C's `longjmp` (Section 3.2.2).

# Chapter 5

# Analysis

> "The real performance cost of first class continuations is the time and money required to implement them."
>
> *(Clinger et al. (1988))*

In this chapter, we evaluate the implementations of escape continuations in Manticore under both performance and relative burden to compiler writers.

In Section 5.2 we explore the effects of these strategies on sequential programs. Then, we compare the overhead of using these continuations for explicit threading in Section 5.3. Finally, we provide an empirical report on the difficulty of implementing various continuations in Manticore (Section 5.4), and draw conclusions in Section 5.5.

## 5.1 Experiment Setup

All performance data was collected using a Linux workstation equipped with two Intel Xeon E5-2687W CPUs and 64GB RAM. A modified version of LLVM 3.8.1 was used by Manticore to compile benchmarks for the x86-64 architecture. During program start-up, the stack cache is pre-loaded with free stacks. Running times exclude program start-up and all numbers reported are averaged from multiple trials, with at most 1.5% standard error. The size of each contiguous stack was 100MB, and stack segments were 16KB each, unless otherwise noted.

| **Program** | **LOC** | **Description** |
|---|---|---|
| ackermann | 9 | Compute the Ackermann function, input (3, 11). |
| takeuchi | 11 | Compute the Tak function, input (33, 22, 12). |
| life | 147 | Game of Life using lists, Queen Bee Shuttle. |
| minimax | 128 | Tic-Tac-Toe solver using Minimax. |
| queens | 41 | N-Queens puzzle solver, N = 12. |
| quicksort | 56 | Quicksort random integers, using lists. |
| nbody | 81 | N-body gravity simulation, N = 5. |

Figure 5.1: Description of sequential benchmarks. LOC is actual lines of code.

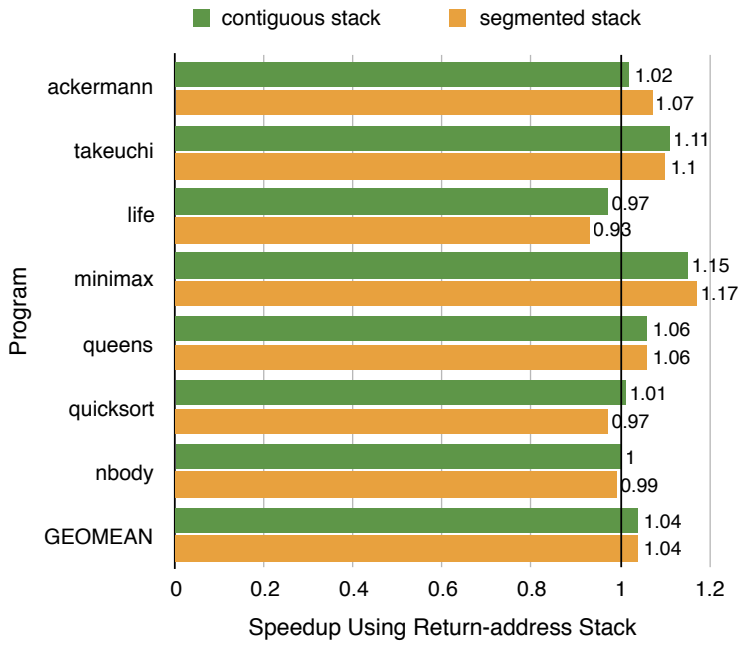


Figure 5.2: Speedups are relative to the same program compiled *without* the use of the RAS. Thus, the RAS improves performance if speedup $> 1$.

## 5.2 Efficient Recursion

To help understand the effect of CPU-specific instructions to initialize stack-allocated continuations, we compiled our sequential program suite (Figure 5.1) in two configurations. In one configuration, we replaced all return instructions in PML functions with an equivalent pop-jump sequence:

```
retq  ⟹  popq %rbp; jmpq *%rbp
```

This replacement effectively disables the CPU's *return-address stack* (RAS), which is an internal stack of return addresses recently placed on the stack that the CPU uses to predict the target of a return (Baer 2009).

Figure 5.2 shows that the return-address stack improves the performance of sequential programs by roughly 4%. Speedups for each benchmark program are shown relative to the same program that was compiled *with* the pop-jump replacement to disable the RAS. We are not sure why the `life` program performs notably worse when using the RAS. Our best guess is that the indirect-branch predictor happens to work better for function returns in that program.

Figure 5.3 highlights the performance of two programs whose performance depends solely on the speed of non-tail calls. In all of our Figures, the "CPS" strategy, which stands for closure-passing style, refers to immutable, heap-allocated frames (Section 3.4).

The efficient frame sharing and reuse used in the stack-allocated strategies is likely the reason for the large difference in these benchmarks. Because the CPS strategy is imple-

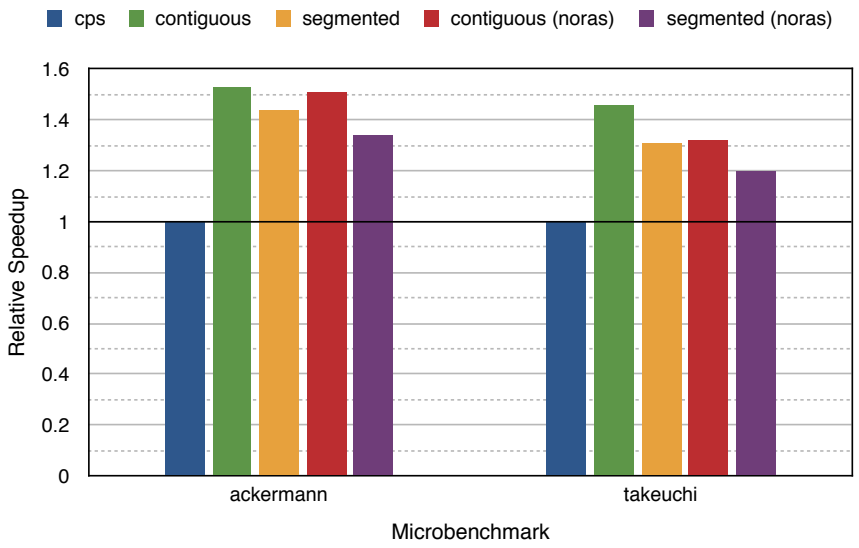


Figure 5.3: Sequential benchmarks that heavily stress the performance of function call and return.

mented with flat closures, there is no frame sharing or reuse, so these results show the worst-case scenario. Shao and Appel (2000) show that there are modest performance gains when using safely-linked closures, which feature environment sharing.

Figure 5.4 shows the performance of a set of more realistic sequential programs. The poor performance of CPS is less noticeable in these programs, but stack-allocation is still up to 18% faster.

**Garbage Collection Overhead** In Figure 5.5, we break down the proportion of time spent in the in the mutator and collector, showing that for many of the benchmarks, the load placed on the garbage collector when using the CPS strategy does *not* appear to be the reason for the performance difference. The Figure also shows that `ackermann` and `quicksort` spend most of their time garbage collecting.

Figures 5.6 and 5.7 dig deeper into those two benchmarks, showing the amount of live data that was promoted (copied) to the next generation while scavenging each generation of the heap.[1] The amount of copied data gives us an idea of the amount of work being performed by the garbage collector in each phase.

The basis of Appel (1987)'s argument for heap-allocated frames is that, with a tracing garbage collector and a sufficiently large heap, no additional load is placed on the garbage collector because frames are short-lived. Stefanovic and Moss (1994) concurs with this assessment, showing that even with heap-allocated frames, only 2% of the nursery is live during garbage collection.

But, in the case of our `ackermann` and `quicksort` benchmarks, a larger portion of the nursery is live when using heap-allocated frames (44% and 30%, respectively) verses the

[1] A Minor GC collects the nursery, where all allocations are created, and the Major and Global heaps are used for older objects, respectively.

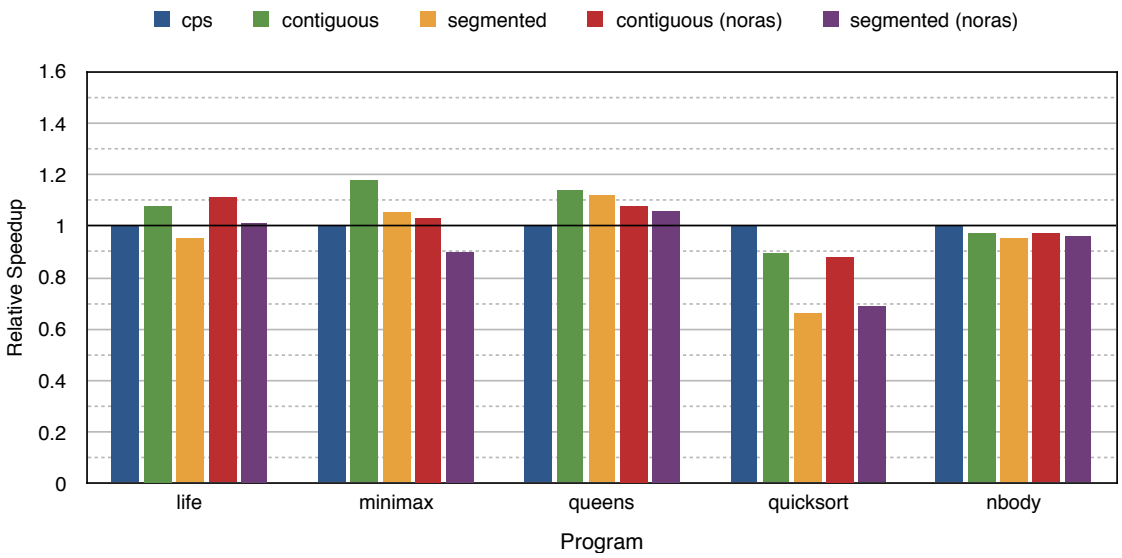


Figure 5.4: The performance of a set of larger sequential programs.

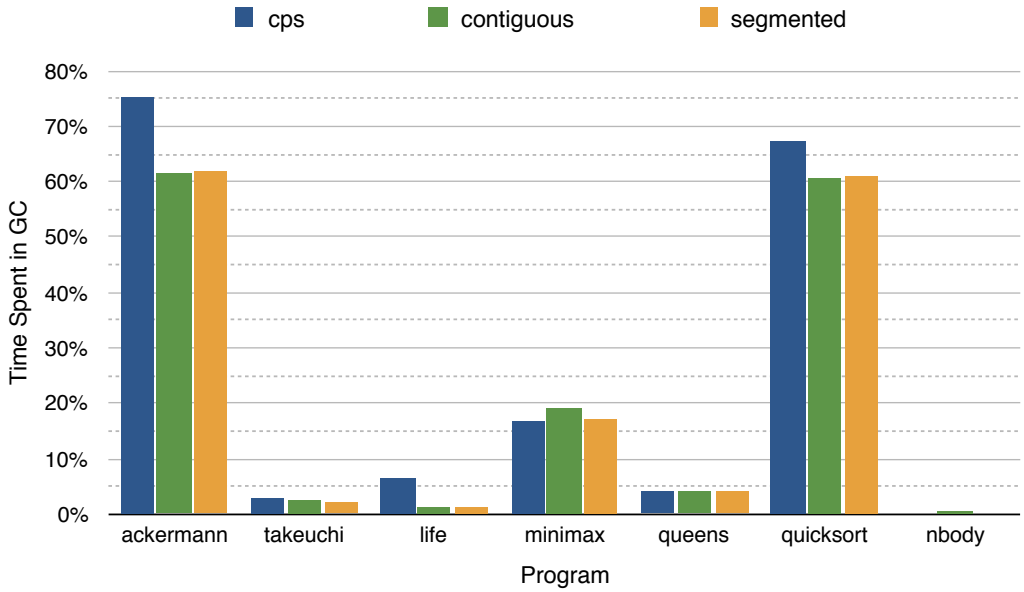


Figure 5.5: The proportion of running time that was spent in the garbage collector.

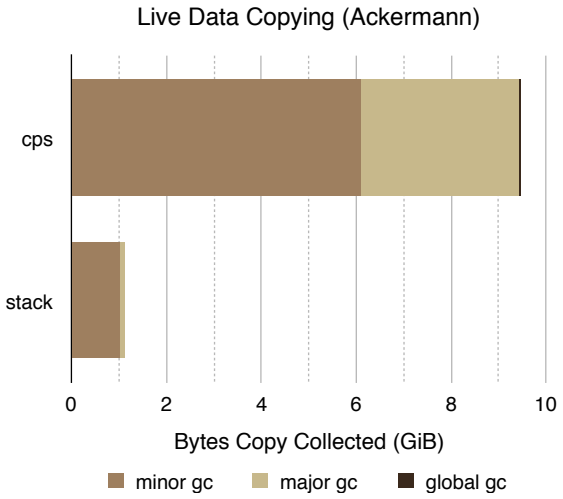


Figure 5.6

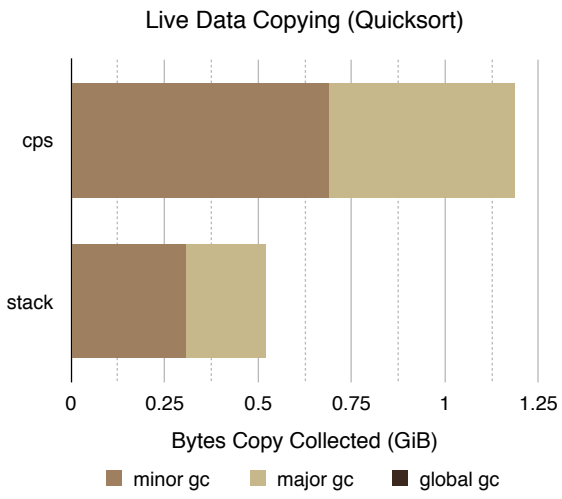


Figure 5.7

stack-allocated versions (13% and 20%, respectively). These two benchmarks exhibit atypical recursion patterns, which causes a large number of allocated frames to be long lived.

Surprisingly, even though the CPS version of `quicksort` does more work in the collector, its overall running time is *better* than any other strategy (Figure 5.4). We believe this is due to the lucky cache-locality of having frames and list elements compacted and placed next to each other in the heap during collection.

## 5.3 Low-overhead Concurrency

To highlight the differences between each implementation of escape continuations when used for concurrency, we measured the overhead of essential CML operations. For these benchmarks, we had to first manually tune both stack-based strategies so they do not perform poorly due to high memory usage. The size of contiguous stacks were lowered to 1MB, and each stack segment was set to 2KB. We believe these these values are not large enough to perform well, or actually execute, the sequential suite.

Figure 5.8 shows the overhead of spawning and synchronizing CML threads. During each iteration, a new CML thread is spawned, which then synchronizes with the spawning thread before exiting. The CPS strategy is the fastest in this case, even with a warmed up stack cache for the other two strategies. Segmented stacks are unusually bad in this benchmark, for reasons we do not completely understand.

The other crucial aspect of CML performance is the overhead of synchronous message passing. Figure 5.9 shows the performance of sending empty messages between two threads. During every iteration, one thread sends a message to the other, and waits for another message in reply. Each message sent in our implementation of CML is a form of `yield`: we capture an escape continuation and place it, along with the message, in the channel before entering the scheduler.

We believe that contiguous stacks outperform the CPS strategy in this test due to cache

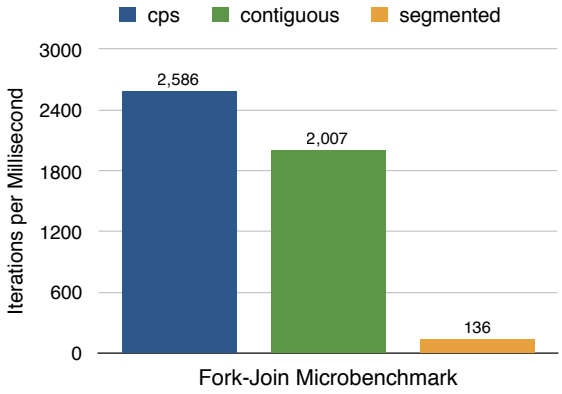


Figure 5.8: A stress test of the overhead to create and destroy threads backed by continuations. Higher is better.

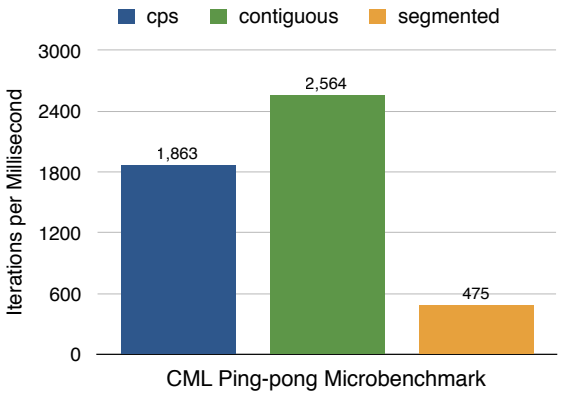


Figure 5.9: The overhead of synchronous message passing over a CML channel. Higher is better.

locality: every iteration reuses the same part of memory to capture escape continuations. With segmented stacks, the overflow handler is fired for every capture to obtain a new segment, which is significantly slower than the other strategies.

## 5.4 Implementation Complexity

The main effect of using any continuation strategy where frames are reused is that it adds to the size and complexity of the runtime system. In particular, both contiguous and segmented stacks require a separate stack area and stack cache. Since many runtime systems already feature generational garbage collection, generational stack scanning for the CPS strategy comes "for free." When frames are reused, a watermarking system is needed.

The other quirk comes in the form of the "tuning" required to use contiguous and segmented stacks. Some of our sequential benchmarks required nearly 100MB of stack space in order to execute, which is not a tenable size if concurrency is also present when using a contiguous stack. In addition, the performance of our implementation of segmented stacks was unusually poor when used for concurrency, despite our best efforts to optimize it.

## 5.5 Conclusion

Of the three strategies available to implement escape continuations, there seems to be no ideal strategy. Contiguous stacks work well for sequential and concurrent programs, but have the downside of a fixed recursion depth and increased runtime system complexity. The CPS strategy with flat closures performs quite poorly in sequential workloads, but are very easy to implement if the priority is efficient concurrency. We see no reason to use segmented stacks at this time, as they are more difficult to implement than contiguous stacks, yet perform worse despite our best attempts to optimize them.

We would like to extend Manticore with one more strategy: mutable, heap-allocated frames. A larger suite of sequential and concurrent programs would provide a better picture of the performance. In addition, we would like to perform a deeper investigation of what causes some strategies to falter by analyzing cache behavior and using dynamic profiling.

# Bibliography


Norman Adams, David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, and James Philbin. Orbit: An optimizing compiler for scheme. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '86, pages 219–233, New York, NY, USA, 1986. ACM. ISBN 0-89791-197-0. doi: 10.1145/12276.13333. URL http://doi.acm.org/10.1145/12276.13333.

Brian Anderson. Abandoning segmented stacks in Rust. Available at https://mail.mozilla.org/pipermail/rust-dev/2013-November/006314.html, 2013. Accessed: 2016-12-3.

Todd A. Anderson. Optimizations in a private nursery-based garbage collector. In *ISMM '10*, pages 21–30, New York, NY, 2010. ACM. doi: 10.1145/1806651.1806655.

A. W. Appel and T. Jim. Continuation-passing, closure-passing style. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 293–302, New York, NY, USA, 1989. ACM. ISBN 0-89791-294-2. doi: 10.1145/75277.75303. URL http://doi.acm.org/10.1145/75277.75303.

Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275 – 279, 1987. ISSN 0020-0190. doi: http://dx.doi.org/10.1016/0020-0190(87)90175-X. URL http://www.sciencedirect.com/science/article/pii/002001908790175X.

Andrew W. Appel. Simple generational garbage collection and fast allocation. *SP&E*, 19 (2):171–183, February 1989. doi: 10.1002/spe.4380190206.

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

Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, Cambridge, England, 1998.

Andrew W Appel and Zhong Shao. An empirical and analytic study of stack vs. heap cost for languages with closures. *Journal of Functional Programming*, 6(01):47–74, 1996.

Sven Auhagen, Lars Bergstrom, Matthew Fluet, and John Reppy. Garbage Collection for Multicore NUMA Machines. In *MSPC 2011*, page 5157, New York, NY, June 2011. ACM. doi: 10.1145/1988915.1988929.

Jean-Loup Baer. *Microprocessor architecture: from simple pipelines to chip multiprocessors*. Cambridge University Press, 2009.

Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. Representing control in the presence of one-shot continuations. In *PLDI '96*, pages 99–107, New York, NY, May 1996. ACM. doi: 10.1145/249069.231395.

Luca Cardelli. The functional abstract machine. Technical Report TR-107, Bell Laboratories, 1983.

Perry Cheng, Robert Harper, and Peter Lee. Generational stack collection and profile-driven pretenuring. *SIGPLAN Not.*, 33(5):162–173, May 1998. ISSN 0362-1340. doi: 10.1145/277652.277718. URL http://doi.acm.org/10.1145/277652.277718.

F. C. Chow. Minimizing register usage penalty at procedure calls. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 85–94, New York, NY, USA, 1988. ACM. ISBN 0-89791-269-1. doi: 10.1145/53990.53999. URL http://doi.acm.org/10.1145/53990.53999.

Will Clinger, Anne Hartheimer, and Eric Ost. Implementation strategies for continuations. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, LFP '88, pages 124–131, New York, NY, USA, 1988. ACM. ISBN 0-89791-273-X. doi: 10.1145/62678.62692. URL http://doi.acm.org/10.1145/62678.62692.

William D. Clinger, Anne H. Hartheimer, and Eric M. Ost. Implementation strategies for first-class continuations. *Higher-Order and Symbolic Computation*, 12(1):7–45, 1999. ISSN 1573-0557. doi: 10.1023/A:1010016816429. URL http://dx.doi.org/10.1023/A:1010016816429.

Olivier Danvy and Julia L. Lawall. Back to Direct Style II: First-class Continuations. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, LFP '92, pages 299–310, New York, NY, USA, 1992. ACM. ISBN 0-89791-481-3. doi: 10.1145/141471.141564. URL http://doi.acm.org/10.1145/141471.141564.

Robertson Davies. *The Manticore*. The Deptford Trilogy. Penguin Books, 1977. ISBN 9780140167931.

Edsger W Dijkstra. Recursive programming. *Numerische Mathematik*, 2(1):312–318, 1960.

Amer Diwan, J. Eliot B. Moss, and Richard L. Hudson. Compiler support for garbage collection in a statically typed language. In *PLDI '92*, pages 273–282, New York, NY, June 1992. ACM. doi: 10.1145/143095.143140.

Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *POPL '94*, pages 70–83, New York, NY, January 1994. ACM. doi: 10.1145/174675.174673.

Damien Doligez and Xavier Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *POPL '93*, pages 113–123, New York, NY, January 1993. ACM. doi: 10.1145/158511.158611.

Kathleen Fisher and John Reppy. Compiler support for lightweight concurrency. Technical memorandum, Bell Labs, March 2002. URL http://moby.cs.uchicago.edu/papers/2002/tm-lightweight-concur.pdf.

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *PLDI '93*, pages 237–247, New York, NY, June 1993. ACM.

Matthew Fluet, Mike Rainey, and John Reppy. A scheduling framework for general-purpose parallel languages. In *ICFP '08*, pages 241–252, New York, NY, September 2008a. ACM. doi: 10.1145/1411203.1411239.

Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly-threaded parallelism in Manticore. In *ICFP '08*, pages 119–130, New York, NY, September 2008b. ACM.

The GHC Team. GHC Commentary: The Layout of Heap Objects. https://ghc.haskell.org/trac/ghc/wiki/Commentary/Rts/Storage/HeapObjects#InfoTables, 2006. Accessed 2017-10-09.

Simcha Gochman, Ronny Ronen, Ittai Anati, Ariel Berkovits, Tsvika Kurts, Alon Naveh, Ali Saeed, Zeev Sperber, and Robert C Valentine. The intel pentium m processor: Microarchitecture and performance. *Intel Technology Journal*, 7(2):21–36, 2003.

Marcelo J. R. Gonçalves and Andrew W. Appel. Cache performance of fast-allocating programs. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, FPCA '95, pages 293–305, New York, NY, USA, 1995. ACM. ISBN 0-89791-719-7. doi: 10.1145/224164.224219. URL http://doi.acm.org/10.1145/224164.224219.

Matthew Hertz and Emery D. Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 313–326, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0. doi: 10.1145/1094811.1094836. URL http://doi.acm.org/10.1145/1094811.1094836.

Richard A. Kelsey. A correspondence between continuation passing style and static single assignment form. In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*, IR '95, pages 13–22, New York, NY, USA, 1995. ACM. ISBN 0-89791-754-5. doi: 10.1145/202529.202532. URL http://doi.acm.org/10.1145/202529.202532.

Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO '04*, pages 75–, March 2004. ISBN 0-7695-2102-9. URL http://dl.acm.org/citation.cfm?id=977395.977673.

Michael Matz, Jan Hubička, Andreas Jaeger, and Mark Mitchell. System V Application Binary Interface – AMD64 Architecture Processor Supplement, July 2012. URL https://refspecs.linuxfoundation.org/elf/x86-64-abi-0.99.pdf.

Gor Nishanov. LLVM Coroutines: Bringing resumable functions to LLVM. https://llvm.org/devmtg/2016-11/Slides/Nishanov-LLVMCoroutines.pdf, 2016. Accessed 2017-10-12.

Mikael Pettersson, Konstantinos Sagonas, and Erik Johansson. *The HiPE/x86 Erlang Compiler: System Description and Performance Evaluation*, pages 228–244. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. ISBN 978-3-540-45788-6. doi: 10.1007/3-540-45788-7_14. URL http://dx.doi.org/10.1007/3-540-45788-7_14.

Simon L. Peyton Jones and Jon Salkild. The spineless tagless g-machine. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 184–201, New York, NY, USA, 1989. ACM. ISBN 0-89791-328-0. doi: 10.1145/99370.99385. URL http://doi.acm.org/10.1145/99370.99385.

Norman Ramsey. Concurrent programming in ML. Technical Report CS-TR-262-90, Dept. of C.S., Princeton University, April 1990.

Norman Ramsey and Simon Peyton Jones. Featherweight concurrency in a portable assembly language. November 2000. URL https://www.cs.tufts.edu/~nr/pubs/c--con.pdf.

Keith Randall. Contiguous Stacks in Go. Available at http://golang.org/s/contigstacks, 2013. Accessed: 2016-12-3.

John Reppy. Optimizing nested loops using local CPS conversion. *HOSC*, 15:161–180, 2002. doi: 10.1023/A:1020839128338.

John H. Reppy. CML: A higher-order concurrent language. In *PLDI '91*, pages 293–305, New York, NY, June 1991. ACM. doi: 10.1145/113446.113470.

Zhong Shao and Andrew W. Appel. Efficient and safe-for-space closure conversion. *ACM TOPLAS*, 22(1):129–161, 2000.

Olin Shivers, James W. Clark, and Roland McGrath. Atomic heap transactions and fine-grain interrupts. In *ICFP '99*, pages 48–59, New York, NY, September 1999. ACM. doi: 10.1145/317765.317783.

K. C. Sivaramakrishnan, Lukasz Ziarek, and Suresh Jagannathan. Multimlton: A multicore-aware runtime for Standard ML. *JFP*, 24:613–674, November 2014. ISSN 1469-7653. doi: 10.1017/S0956796814000161.

Jonathan Sobel and Daniel P. Friedman. Recycling continuations. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 251–260, New York, NY, USA, 1998. ACM. ISBN 1-58113-024-4. doi: 10.1145/289423.289452. URL http://doi.acm.org/10.1145/289423.289452.

Darko Stefanovic and J. Eliot B. Moss. Characterization of object behaviour in standard ml of new jersey. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, LFP '94, pages 43–54, New York, NY, USA, 1994. ACM. ISBN 0-89791-643-3. doi: 10.1145/182409.182428. URL http://doi.acm.org/10.1145/182409.182428.

James M. Stichnoth, Guei-Yuan Lueh, and Michal Cierniak. Support for garbage collection at every instruction in a Java compiler. In *PLDI '99*, pages 118–127, New York, NY, May 1999. ACM. doi: 10.1145/301618.301652.

Mitchell Wand. Continuation-based multiprocessing. In *LFP '80*, pages 19–28, New York, NY, August 1980. ACM. doi: 10.1145/800087.802786.

Andy Wingo. stack overflow. Available at `https://wingolog.org/archives/2014/03/17/stack-overflow`, 2014. Accessed: 2017-3-15.