From folklore to fact: Comparing implementations of stacks and continuations

Kavon Farvardin
University of Chicago
kavon@cs.uchicago.edu

John Reppy
University of Chicago
jhr@cs.uchicago.edu

Abstract

The efficient implementation of function calls and non-local control transfers is a critical part of modern language implementations and is important in the implementation of everything from recursion, higher-order functions, concurrency and coroutines, to task-based parallelism. In a compiler, these features can be supported by a variety of mechanisms, including call stacks, segmented stacks, and heap-allocated continuation closures.

An implementor of a high-level language with advanced control features might ask the question "what is the best choice for my implementation?" Unfortunately, the current literature does not provide much guidance, since previous studies suffer from various flaws in methodology and are outdated for modern hardware. In the absence of recent, well-normalized measurements and a holistic overview of their implementation specifics, the path of least resistance when choosing a strategy is to trust folklore, but the folklore is also suspect.

This paper attempts to remedy this situation by providing an "apples-to-apples" comparison of five different approaches to implementing call stacks and continuations. This comparison uses the same source language, compiler pipeline, LLVM-backend, and runtime system, with the only differences being those required by the differences in implementation strategy. We compare the implementation challenges of the different approaches, their sequential performance, and their suitability to support advanced control mechanisms, including supporting heavily threaded code. In addition to the comparison of implementation strategies, the paper's contributions also include a number of useful implementation techniques that we discovered along the way.

ACM Reference Format:
Kavon Farvardin and John Reppy. 2020. From folklore to fact: Comparing implementations of stacks and continuations. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3385412.3385994

PLDI '20, June 15–20, 2020, London, UK
2020. ACM ISBN 978-1-4503-7613-6/20/06.
https://doi.org/10.1145/3385412.3385994

1 Introduction

The efficient implementation of function calls and non-local control transfers is a critical part of modern language implementations and is important in the implementation of everything from recursion, higher-order functions, concurrency and coroutines, to task-based parallelism. In a compiler, these features can be supported by a variety of mechanisms, including call stacks [19], segmented stacks [31], and heap-allocated continuation closures [5], but semantically they can all described in terms of *continuations* [46].

An implementor of a high-level language with advanced control features might ask the question, "what is the best choice for my implementation?" Much of the current understanding of performance trade-offs are based on cross-language and cross-compiler comparisons [17], simulations and theoretical analysis [10], or direct measurements performed 30 years ago [15]. In the absence of recent, well-normalized measurements and a holistic overview of 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 segmented stacks in part because of reports from RUST and GO developers suggesting poor performance owed to segment thrashing [1, 41, 49, 54]. It turns out that the problem of segment thrashing was solved by Bruggeman *et al.* in the Chez Scheme compiler [13]. 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 garbage-collected languages. Thus, when weighing one strategy against another, a deep understanding of the design space and constraints is crucial.

This paper attempts to remedy this situation by providing an "apples-to-apples" comparison of five different approaches to implementing call stacks and continuations. This comparison uses the same source language, compiler pipeline, LLVM-backend, and runtime system, with the only differences being those required by the differences in implementation strategy. We compare the implementation challenges of the different approaches, their sequential performance, and their suitability to support advanced control mechanisms, including supporting heavily threaded code as found in Concurrent ML [45], Erlang [11], and Go [23] programs. In addition to the comparison of implementation strategies, the paper's contributions also include a number of useful implementation techniques that we discovered along the way.

The remainder of the paper is organized as follows. We begin with a comparison of prior evaluations of various implementation schemes. We then describe the context of our work in Section 3 and the details of the implementation space in Section 4, including the challenges of each approach. Section 5 presents the main results of the paper. Finally, we conclude in Section 6.

2 Prior Work

The earliest attempt to compare implementation strategies was by Clinger *et al.* [15] in 1988, who focused on support for *first-class* continuations as found in Scheme. Their experiments were based on the MacScheme compiler with support for various implementation schemes (including several that we evaluate). Their conclusions were that contiguous stacks are ill-suited to efficient first-class continuations, because of the extra cost of stack copying, and that heap-allocated stack schemes are sensitive to the efficiency of the garbage collector.

At the same time, the SML of New Jersey compiler switched to using heap-allocated continuation closures in its implementation [7]. Appel had argued that a generational garbage collector could result in heap allocation being faster than stack allocation [3] and switching to heap-allocated continuations simplified the implementation. This choice was somewhat controversial [36]. In later work, Shao and Appel [10] presented a rigorous argument that continuation closures were more efficient than a traditional stack-based implementation. Their arguments regarding efficiency were supported by theoretical analysis and simulations produced using a modified compiler that measured cache effects and instruction counts. Their cost model was fairly simple, assuming single-cycle instructions, a direct-mapped data cache with a 10-cycle read-miss penalty, no write miss penalty, and an infinite instruction cache. The main weakness in their analysis was inflated instruction counts for stack-frame initialization because of no frame reuse for non-tail calls [see 17, Section 7].

In 1999, Clinger *et al.* [17] expanded their work from 1988 with deep analytical discussion that directly responded to Shao and Appel's study and was backed by proxy performance metrics. They argued that for most programs, a segmented stack or an incremental stack/heap strategy is better than the strategy put forward by Shao and Appel. But their direct performance evaluation was minimal, consisting of an experiment with four different compilers tested on one synthetic coroutine benchmark that makes heavy use of first-class continuations.

The main problem with these previous studies is that they are outdated for modern processors that have deep cache hierarchies, high clock rates, and sophisticated branch prediction hardware.

A number of other 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: improved cache locality and reduced garbage collector load.

Gonçalves and Appel [28] 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 [50] 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 [3]. Hertz and Berger [30] 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.

3 Background Concepts

Given any point in the program, its *continuation* is the abstract notion of the "rest of the computation." 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 an arithmetic 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.

When entering a function, the top of the *call stack* represents the return continuation of that invocation, *i.e.*, the context in which the function's result is needed. Concretely, the continuation is represented by the return address (on the top of the stack or in a register) and the stack pointer; with these two pieces of information, control can be returned to the call site. Of special interest are *tail* calls, which are calls that are the last thing a function does before returning. Because the continuation of a tail call is just the return continuation of the calling function, an implementation can optimize the call to save space. In a stack-based implementation, this optimization involves deallocating the caller's stack frame before making the tail call (which is implemented as a jump). For functional languages, which often express iteration as tail recursion, this optimization is critical.

3.1 Continuation-Passing Style

It is difficult to precisely discuss function calls and other control-flow mechanisms without introducing a standard notation for them. For purposes of this section, we introduce a factored continuation-passing style IR that makes non-local control flow explicit using continuations;[1] Figure 1 gives the abstract syntax of this IR. This IR distinguishes between user

[1] For compactness, the continuations of primitive operations and conditional control flow are implicit in the syntax.

$$
\begin{array}{rcl}
\mathit{exp} & ::= & \mathbf{let}\ (x_1,\ldots) = \mathit{prim}(y_1,\ldots) \\
 & | & \mathbf{fun}\ f(x_1,\ldots/k) = e_1\ \mathbf{in}\ e_2 \\
 & | & \mathbf{cont}\ k(x_1,\ldots) = e_1\ \mathbf{in}\ e_2 \\
 & | & \mathbf{if}\ x\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \\
 & | & \mathbf{apply}\ f(x_1,\ldots/k) \\
 & | & \mathbf{throw}\ k(x_1,\ldots) \\
\mathit{prim} & ::= & \text{primitive operations and values}
\end{array}
$$

Figure 1. A continuation-passing style intermediate representation.

functions (defined by **fun** and applied by **apply**) and continuations (defined by **cont** and applied by **throw**). For example, consider the recursive factorial function written in ML:

```
fun fact n = if n = 0
      then 1
      else n * fact (n - 1)
```

This function is represented as follows in the CPS IR:

```
fun fact (n / k) =                      1
  let isZero = n == 0 in                2
    if isZero                           3
      then throw k (1)                  4
      else cont retK (x) =              5
             let res = n * x in         6
               throw k (res)            7
in                                      8
  let arg = n - 1 in                    9
    apply fact (arg / retK)             10
```

The additional parameter `k` to `fact` is bound to the return continuation, which is used by an invocation of `factorial` to return a result to its caller in lines 4 and 7. The continuation `retK` (lines 5–7) is the return continuation for the non-tail-recursive call to `fact` in the ML code.

3.2 Reified Continuations

The slash used in the parameter list of `fun` expressions separates continuation parameters introduced by CPS conversion,[2] from uses of parameters that are bound to *reified continuations* in the original program. Reified continuations are continuations that are captured as concrete values and can be used to express various advanced control-flow mechanisms. In a compiler that supports advanced control-flow mechanisms, such as concurrency or task parallelism, reified continuations can be used to implement these mechanisms, even if they do not appear in the IR.

Reified continuations are typically classified based on their allowed number of invocations and their extent (or lifetime), as these factors affect their implementation. If a continuation can be invoked at most once, it is known as a *one-shot* continuation [13]; otherwise it is called a *multi-shot* continuation and can be invoked arbitrarily many times. A continuation can either have a stack extent (like stack-allocated variables) or an unlimited lifetime.

The *first-class* continuations produced by `callcc` (found in Scheme and some implementations of ML) are multi-shot continuations with unlimited lifetime. A restricted form of `callcc`, called `call1cc` [13], reifies one-shot continuations of unlimited lifetime.[3] An *escape* continuation is a one-shot continuation whose lifetime is limited to its lexical scope [25, 40]. Escape continuations are simpler to implement due to their restrictions, yet still powerful enough to implement context-switching operations.

3.3 Programming with Continuations

There are many examples in the literature of using reified continuations to implement a wide range of advanced control-flow mechanisms [24, 29, 33, 39, 43, 44, 53]. Such implementations could either be user-level code or part of a compiler's implementation of language features. To provide a bit of flavor of what such code looks like, we consider the implementation of coroutines using the weakest form of reified continuation: escape continuations. We assume that we have the following ML interface to this mechanism:

```
type 'a econt

val callec   : ('a econt -> 'a) -> 'a
val throw    : 'a econt -> 'a -> 'b
val newStack : ('a -> 'b) -> 'a econt
```

Escape continuations are reified using `callec` and applied using `throw`. Because escaping continuations have stack extent, they cannot be used to create new execution contexts, so we include the function `newstack` for this purpose.

A suspended coroutine's state can be represented as a unit-valued continuation and we define a global list of ready coroutines and a function to schedule the next ready coroutine.

```
type coroutine = unit cont
val ready : coroutine list ref = ref[]

fun dispatch () = let
      val (next, rest) = !ready
      in
        ready := rest;
        throw next ()
      end
```

Then, a function for switching coroutines can be implemented as

[2] Note that we could add additional continuation parameters to support exception handling or other non-local control transfers.

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

```
fun yield () = callec (fn k => let
      val _ = ready := !ready @ [k]
      in
        dispatch ()
      end)
```

And creating a new thread is implemented as

```
fun spawn (f : unit -> unit) = let
      val k = newStack f
      in
        ready := k :: !ready;
        yield()
      end
```

It is clear from this code that the continuations reified by `callec` are never invoked more than once and that they have stack extent. Because of the restrictions placed on escaping continuations, they can be easily implemented in a stack-based runtime model.

4 Implementation Details

In this section, we describe the details of our various implementations. We start by giving a brief overview of the compiler and runtime system used to implement our experiments. We then survey the different implementation strategies for call stacks and continuations. The sequel discusses the challenges that an implementation must address and how we address those in our implementation.

4.1 The Compiler

Our experimental system implements a parallel dialect of ML that includes support for Concurrent ML-style message passing. The compiler is a whole-program compiler that is structured as a pipeline of transformations between a sequence of intermediate representations. Code to implement concurrency and parallelism features is implemented using reified continuations and is available to the compiler to optimize.

For purposes of this paper, the last three stages are the most important. These start with an ANF-style representation that is then CPS converted to an IR similar to the one in Figure 1. We perform a number of optimizations on the CPS representation and then apply a flat, safe-for-space closure conversion to transform it to a first-order representation that can be used to generate LLVM code.

The first-order representation supports both closure-passing to implement call/return linkage and traditional function calls. For the stack-based runtime models, we we modify closure conversion using a combination of ideas from Danvy and Lawall [18], Kelsey [32], and Reppy [42] to (1) map non-tail calls to traditional call operations and (2) introduce `callec` operations to reify any escape continuations that are present.

Our runtime system is designed to support parallel execution on multicore processors. The garbage collector is based on a combination of Appel's semi-generational [4] collector and the DGL concurrent collector [21, 22]. While this collector is designed to maximize parallel scalability, instead of sequential performance, it works reasonably well for our benchmarks.

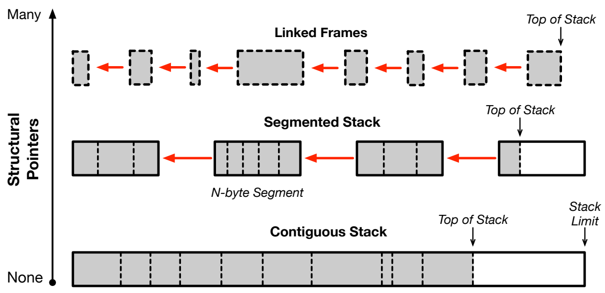

Figure 2. High-level spectrum of call stack implementations, as a function of the number of pointers used to maintain the structure. Dashed lines separate stack frames.

4.2 Strategies

We study five different implementation strategies, ranging from traditional contiguous stacks of fixed size to heap-allocated continuation closures. The five strategies are as follows; we tag each approach with a short name that is used when presenting the data in the next section.

- **contig** — stack frames are allocated in a fixed-size 128Mb contiguous region; it is the native stack discipline used by C.
- **resize** — stack frames are allocated in a contiguous region, but are copied to a larger region if more stack space is required. The initial size is 8Kb and is doubled each time it grows. Resizable stacks do not work for implementations that allow pointers into the stack.
- **segment** — stack frames are allocated in contiguous 64Kb segments, which are linked together. We chose 64Kb as the segment size because performance would drop with smaller segments, but did not improve much with larger segments.
- **linked** — stack frames are heap allocated and linked together. This scheme should not be confused with a linked closure representation for nested functions [47].
- **cps** — return continuations are represented by immutable, heap-allocated continuation closures that are passed as an extra argument to function calls.

The first four of these strategies, which we call the *stack-based strategies*, implement continuations using a call stack abstraction, where stacks are mutable and non-tail calls return to the caller's frame. They differ in how the call stack is represented in memory. The memory structure can be characterized by how often pointers are used to link frames together as shown in Figure 2. At one end of the spectrum, the contiguous stacks (including resizable stacks) require no inter-frame pointers if the frame sizes are statically known. Variants of segmented stacks allocate smaller segments of memory at a time, linking each *segment* together with pointers. 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 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.

A major concern for compiler designers is the efficiency of the call stack for sequential programs. The four stack strategies all avoid allocation for return continuations and are able to preserve live values across calls by saving them in their frames. Furthermore, these strategies can all use the hardware instructions for call/return, which can lead to better branch prediction. Where they differ is in their complexity of implementation, their memory footprint (especially for heavily threaded applications), and the overhead needed to check for and handle stack overflow. Continuation closures, on the other hand, require increased memory allocation rates and may have reduced cache locality, but they provide perhaps the simplest implementation, especially for advanced control-flow features.

Another concern that effects users of a language implementation is a limit on recursion depth. Contiguous stack implementations 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 avoids the requirement of updating pointers into the stack during a relocation, but it requires the implementer or programmer to correctly predict the maximum stack requirements. When stacks are allocated in the heap or in segments, recursion is no longer limited in depth by the size of the reserved stack space, but can use all memory available to the process. A resizing stack adds stack overflow checks while maintaining a fully contiguous layout of frames by copying the entire stack into a new, larger area of the heap during overflow.

4.3 Implementation Issues

4.3.1 Stack Overflow. Functional programs often make use of deep recursion, which may trigger a *stack overflow*. There are two issues that an implementation must address: how to detect overflow and what to do about it. The two primary methods of detecting overflow are page protection (to cause a fault on overflow) and explicit limit checks in the code.

Most implementations of contiguous stacks (including ours) uses a guard page at the end of the stack region that is memory protected. Stack overflow is detected by a memory fault and the program is terminated.

For resizable and segmented stacks, explicit checks are the usual approach, since it is difficult to robustly recover from a memory fault. For resizable stacks, we copy the stack into a new region that is twice the current region's size. For segmented stacks, we allocated a new segment on overflow. To avoid thrashing behavior, where control bounces back and forth between two segments, it is desirable to add some hysteresis. We do so by copying the top four frames from the caller's segment to the new segment.

For the linked frames and continuation-closure strategies, there are no stack overflow checks. Instead, we can use the heap-limit checks to achieve the same result.

4.3.2 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 [6], so they are crucial to optimize. Because capture-free functions do not capture their return continuation (they either return or pass their return continuation in a tail call), can omit frame allocation as long as they do not use stack memory (*e.g.*, for register spilling). This optimization is particularly useful for strategies that use explicit stack-limit checks, since it avoids the overhead of the limit check and the potential costly consequences of a failed check.

There are also techniques for reducing the amount of allocation required for continuation closures. These include adding dedicated registers to the calling convention that can be used to pass live values from the caller to its return continuation [9]. In effect, this technique implements a form of callee-save registers. Another effective technique uses linked representations of continuation closures, instead of flat closures [48]. Each of these techniques has non-trivial implementation costs in the compiler, so we have not implemented them in our testbed. It should be noted, however, that these techniques individually produced runtime improvements in the 10–20% range in the SML of New Jersey compiler.

4.4 Finding GC Roots

Accurately identifying live heap pointers, or *roots*, in the mutator's state is necessary in a garbage collected runtime system. For the stack-based strategies, this issue adds significant complexity to both the compiler and runtime system.

Assuming that garbage collection can only occur at well-defined program points where the location of potential roots is known, we just need a way to identify the live roots at each non-tail call site in the program. This information could be dynamically maintained as part of the function-call protocol, but we can avoid the runtime overhead by generating a PC map that maps return addresses to root-location data [20].[4] The garbage collector can then use this map 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 (finite map), or spatially by placing metadata data just before the instruction pointed to by the return address [31, see Figure 4].

[4] It is possible to generate location data for *every* instruction using data compression techniques [51].

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. [14] found that the presence of callee-saved registers requires a two-pass approach when scanning the stack to compute the root set. Because of this complexity, we did not implement callee-save registers in our testbed.

For the stack-based strategies, we rely on LLVM's support for precise garbage collection in the presence of frame sharing and reuse [35]. LLVM generates information alongside the code describing the layout of live pointers in the stack frame at every call site. We then build a hash table keyed on return addresses during runtime-system initialization that is used by the GC while scanning a stack.

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. [14] found that scanning stacks for GC roots can 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 [2], with the universal goal of detecting whether a given stack frame has been modified since the last collection cycle.

For the stack-based strategies, we place a *water mark* in each stack frame to avoid excessive stack scanning. A water mark 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 mark 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 mark in the frame established by that function, indicating that the frame may contain pointers into the nursery. During the collection of a generation, these water marks are overwritten by the collector with the indicator corresponding to the generation that the frame's pointers were promoted. 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-water mark* 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 return, will have an older watermark in its frame, with roots that might be in a younger generation.

With immutable frames used in the linked-closure stack, there is no extra effort needed to add generational stack scanning 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.

4.4.1 CPU and ABI Support.

Historically, instruction sets such as the x86 have contained dedicated instructions to assist with function calls. 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 [27]. Pettersson et al. [38] 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 the `call` instruction, 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 call stacks. Many foreign-function ABIs expect a large contiguous region of memory and rely on a guard page to detect stack overflow (Section 4.3.1), 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 associated with them, which is expensive. An alternative approach is to switch to a dedicated stack for for each foreign-function call, which incurs a few instructions per call to swap a different stack into the stack pointer register.

For contiguous stacks, foreign calls can be implemented using the same stack. Our implementation of contiguous stacks matches the expectations of the ABI, so there is minimal overhead in making a foreign call. The resizing and segmented-stack strategies cannot directly call foreign functions using the current stack, since it is possible that the amount of head room is insufficient to execute the foreign call [1]. Instead, we call a runtime-system shim that switches to a dedicated contiguous stack to make the call. The same strategy is essentially used for the linked and continuation-closure strategies.

4.4.2 Thread Creation.

The stack-based strategies are limited in their support of continuations to escape continuations. Because of the lifetime restrictions on escape continuation (Section 3) it is not possible to implement the `spawn` function from Section 3.3 using only `callec`. Instead, we need the `newStack` primitive to create a fresh execution context. First, a call stack is allocated, consisting of one frame that will execute an exit when invoked. Then, we push a frame that will apply the given function f to unit onto the stack. The resulting stack can be packaged as an escape continuation that will execute the function Now, the top of the stack is a primed-and-ready paused thread that executes f when it is thrown to.

There is an additional complication in the presence of parallelism. Consider, for example the `yield` function (ignoring the fact that there is no locking in the code).

```
fun yield () = callec (fn k => let
      val _ = ready := !ready @ [k]
      in
        dispatch ()
      end)
```

Notice that the scheduling code is running on the same stack as is reified by the continuation `k`. Thus, there is a race condition between the adding of `k` to the scheduling queue and the dispatching of the next thread. During this interval, it is possible that another thread might schedule `k` before the next thread is dispatched. At that point, there would be two active execution threads attempting to use the same stack frame. There are several possible solutions to this problem. One approach is to have a per-stack lock bit that is acquired when the continuation is captured by the `callec` and then released when the next thread is dispatched. Another processor attempting to schedule the continuation `k` would end up spinning on the lock until it had been released. Li et al. [34] proposed to avoid this race by modeling the exchange with software-transactional memory, although this approach turned out to be too costly in the common non-conflict case and was abandoned. Our approach, which is also what GHC does, is to switch to a scheduler stack before executing the scheduling code. Thus, by the time the original stack becomes visible to other processors, it is no longer being actively used to run the scheduling code.

The continuation-closure strategy does not suffer from these problems. First, it can support proper first-class continuations, which can be used to implement the `newStack` primitive directly. Second, because continuation closures are immutable, there is no race condition during scheduling.

4.5 Implementing `callec`

The implementation of `callec` and `throw` on contiguous stacks is fairly straightfoward. Essentially, `callec` is equivalent to C's `setjmp` function and `throw` is equivalent to `longjmp`.

The resizing and segmented-stack strategies were the main challenge because of the need for frame copying on overflow. The difficulty is that once a continuation is reified (or captured), a pointer into the stack exists in the heap, so it is no longer safe to move that frame during an overflow without keeping a remembered set of captured continuations. Hieb et al. [31] deal with this problem by sealing off the reified continuation by splitting the space for the segment into two halves, using a special underflow frame in the middle of the segment to separate them. This frame acts as a stopping-point for the stack walker during an overflow, to protect the captured frames from being moved.

Implementing the segment sealing technique as described would have been tricky given our existing implementation. Instead, our solution is to mark the segment as *not copyable on overflow* if a continuation capture occurs in the segment, with that mark being cleared once the segment is freed. This technique means that a resizing stack will also act like a segmented stack if a continuation capture occurs in the currently-used stack region.

Eager memory management was also crucial for the efficiency of segmented and resizing stacks. Freeing a segment on underflow by pushing it on the top of the stack cache is important, along with additional metadata to check whether it is safe to free a segment during a throw. We inspect inspect both the `from` and `to` segments during a throw to see if they're from the same or different continuation contexts, which is a unique ID assigned to each stack produced by `newStack`. During a throw, if the `from` and `to` segments are not equal but within the same continuation context, then it is always safe to free the `from` segment (because of the lifetime restriction of escape continuations). Otherwise, if they are from different contexts, then it is *not* safe, because we may simply be switching to another thread, and the current thread might still be live (and already on a scheduling queue).

Implementing `callec` on linked stacks was also tricky, because of the need to support promotion of mutable data in a DLG-style heap [21, 22]. The challenge comes from the fact that there may be arbitrary pointers to a linked frame that were captured via `callec`. We do not want to make every promotion trigger a GC to find and update these pointers, which is *not* required of immutable data. The trick we use is that during the capture of a linked frame, we allocate a small continuation launcher that contains a pointer to the frame, just as we do for the other stack strategies. The difference is that this launcher's code will first check the GC tag of the pointed-to frame, following the forwarding pointer if one exists to the latest version of that frame. Once garbage collection occurs, existing launchers will have their pointed-to frame updated by the GC to the correct location, removing the indirection on throws.

5 Evaluation

This section provides an empirical evaluation of the five different strategies for implementing continuations discussed in Section 4. Section 5.1 provides an analysis of objective metrics that are related to application performance, as measured from a single compiler and runtime system. Section 5.2 describes the development experience of implementing and tuning each implementation in a compiler.

5.1 Application Performance

Table 1 describes our suite of diverse benchmarks that stress recursion and the heavy use of continuations. Our suite is

Table 1. Details for all benchmarks. Origin indicates a external compiler test suite they were ported from, if any. LOC is the actual lines-of-code, minus the setup function *etc.* A star (★) means there is also an `ec` version that uses `callec` and `throw`.

Program	LOC	Origin	Description	Input (n)	Iters
`ack` ★	9	Larceny	Compute the Ackermann function	(3, 11)	4
`divrec`	13	Larceny	Recursively divide n by two; numbers are list of unit	1,000	10^6
`fib` ★	4	Larceny	Recursively find n^{th} Fibonacci number	40	5
`merge`	8	MLton	Merge two length n sorted lists into one	100,000	150
`motzkin` ★	18	–	Find n^{th} Motzkin number	21	4
`quicksort`	18	–	Quicksort a fixed random-number list of length n	2×10^6	1
`sudan` ★	11	–	Compute the Sudan function $F_2(x, y)$	(2, 2)	10^7
`tak` ★	9	Larceny	Compute the Tak function	(40, 20, 11)	1
`takl`	16	Larceny	Compute the Tak function using lists as numbers	(40, 20, 11)	1
`cpstak`	25	Larceny	CPS version of `tak`	(40, 20, 11)	1
`evenodd` ★	9	MLton	Mutually tail-recursive subtraction loop	5×10^8	4
`mandelbrot`	48	MLton	Tail-recursive Mandelbrot set loop	(32,768, 2,048)	1
`tailfib`	4	MLton	Tail-recursive version of `fib`	44	30
`deriv`	70	Larceny	Symbolic derivation	$3x^2 + ax^2 + bx + 5$	10^7
`life`	147	SML/NJ	Simulate n steps of two Game of Life grids	25,000	2
`mazefun`	194	Larceny	Constructs a maze in a purely functional way	(11, 11)	10^4
`minimax`	128	–	Tic-Tac-Toe solver using Minimax	3×3 board	10
`nqueens`	44	GHC	Enumerate the solutions to the n-queens problem	13	2
`perm`	75	Larceny	GC benchmark using Zaks's permutation generator	(5, 9)	10
`primes`	19	Larceny	Compute list of primes less than n	1,000	2×10^4
`mcray`	513	–	Functional raytracer; Monte Carlo algorithm	300×200 canvas	1
`scc`	44	GHC	Strongly-connected components of n vertex graph	5,000	3
`ffi-fib`	7	–	`fib` with FFI calls to integer identity function	40	1
`ffi-trigfib`	12	–	`fib` with FFI calls to trigonometry functions	40	1
`cml-spawn`	9	–	Fork a new thread then sync immediately n times	10^6	1
`cml-pingpong`	20	–	Two threads send n messages over a channel	4×10^6	1

organized into six different groups based on common characteristics of each program, with each group ordered from top to bottom and separated by horizontal lines in the table. The majority of the programs were ported from the benchmark suites of Larceny Scheme [16], the Glasgow Haskell Compiler [26], or the SML compilers MLton [37] and SML/NJ [8].

The number of iterations listed for each program is the number of times the program's computational kernel was executed on the input *within one trial*, which corresponds to one launch of the compiled program. Unless otherwise noted, any benchmark running times are reported as the mean of five trials, with 95% bootstrapped confidence intervals plotted. The running time of a trial is the measurement of only the time it took to execute the kernel for the given number of iterations, *i.e.*, it excludes the time to initialize the runtime system, generate the input, *etc.*We also use the Linux `perf-stat` tool ts access the CPU's performance monitoring units, providing average metrics from four complete-process executions.

Experiment Setup. All performance data was collected using a server equipped with two Intel Xeon Gold 6142 CPUs and 64GB RAM running Ubuntu 16.04.6 LTS. We compiled all benchmarks for x86-64 with a modified version of LLVM 9.0.1 that implements additional function prologues and epilogues that are compatible with the runtime system. LLVM's IR optimizations were limited to tail-call elimination to ensure a fair comparison, because its optimizer is better suited for direct-style code. The production-grade compilers MLton version 20180207 and SML/NJ 110.95 (64-bit) were also evaluated to provide context for the performance of our compiler. MLton is an example of a real-world compiler that uses a resizing stack, and SML/NJ uses continuation closures.

Each stack strategy's initial segment sizes are 64KB for segmented stacks and 8KB for resizing stacks. These sizes were chosen empirically by finding the smallest size such that a larger initial size for resizing stacks yielded no benefit for the deeply-recursive `ack` benchmark, with segmented stacks set eight times larger. For contig stacks, the size was set to 128MB because it is the smallest power-of-two size such that all programs execute without triggering a fatal overflow. During overflow for segmented stacks, frames are copied until either a maximum of four frames or one-eighth of a segment's data is encountered. Stack memory that is not allocated in the moving heap uses `malloc` (or `aligned_alloc`) as

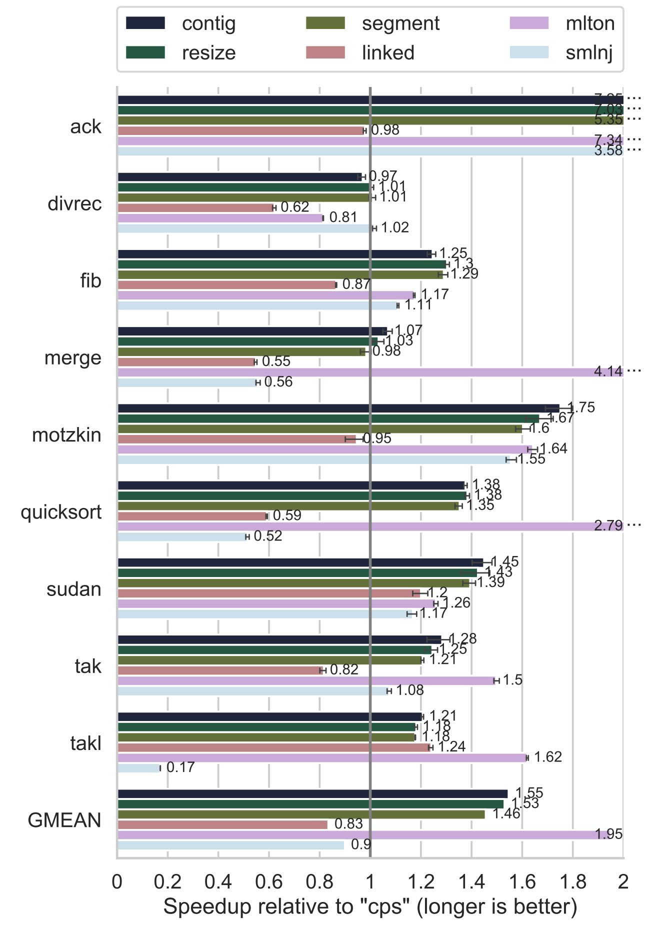

Figure 3. Comparison of run-times for **toy** group.

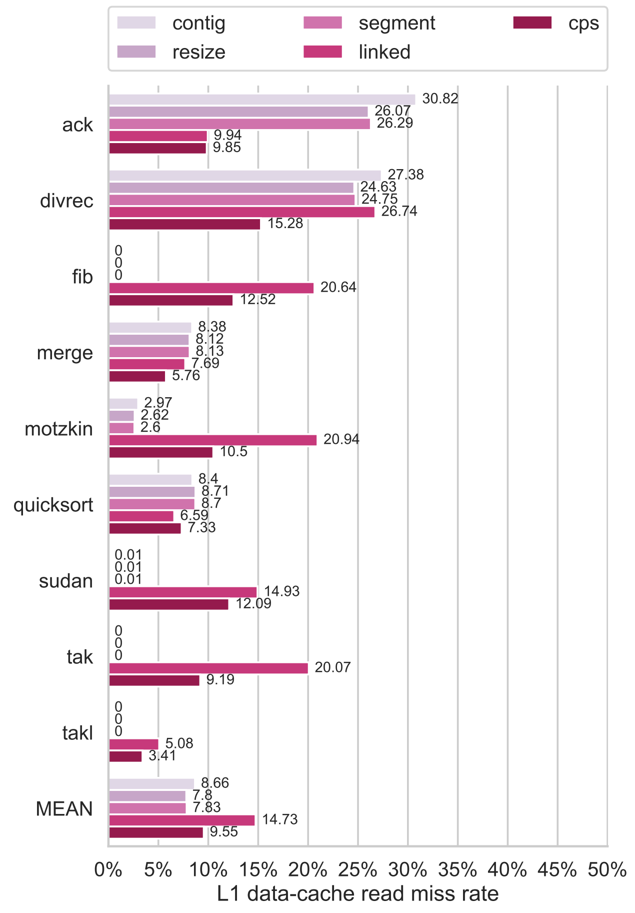

Figure 4. Proportion data reads that missed the L1 data cache.

provided by `jemalloc` 3.6. During program start-up (and prior to running-time measurement), the segment cache is preloaded with 64 free segments if using segmented or resizing stacks.

5.1.1 Recursion Performance.

Together the first three groups of functional programs listed in Table 1 evaluate all aspects of function call overhead. The first group consists of recursive **toys**, or microbenchmarks, that exhibit a variety of patterns for deep recursion where the work performed between each call is minimal. The second group of programs are referred to as **tail** because they exclusively make use of tail recursion, which does not grow the call stack but does stress tail-call optimization. The third group are **real** programs that perform work similar to those found in real applications.

Toy and Tail Programs. Figure 3 shows that on average the contig, segmented, and resizing stacks, are *significantly* faster for toy programs, exhibiting a 1.46× to 1.55× speedup relative to closure-based stacks. Linked stacks are the slowest, seeing a 0.83× speedup, *i.e.*, a 1.2× slowdown relative to closure-based stacks.

The `ack` results are a major outlier, with contig stacks running over 7× faster than closures, due to the program's unique pattern of recursion. A typical recursive pattern is similar to a depth-first traversal of a graph, *e.g.*, `fib` and `tak` traverse a complete k-ary tree of depth n, where k^n calls occur just to visit the leaves. While performing fewer calls per iteration (179 million vs 331 million) in comparison with `fib`, the `ack` program recurses much deeper (16,380 frame vs 40 frame maximum depth) and is only singly-recursive [52].

For smaller inputs, `ack` has a peak-and-valley pattern of recursive depth history, making it a nightmare scenario for closure and linked-frame stacks that allocate directly in the heap: 67% of running time for these strategies was spent in the garbage collector. For closure-based stacks, 24GB of data was allocated in the nursery, with 65.6% of that being promoted to the major heap and 40% later promoted to the global heap. Linked stacks saw similar rates of live data promotion, but even with frame sharing, it still allocated 2.7GB more data than the closure strategy because of the need for a frame watermark field and aligned allocation.

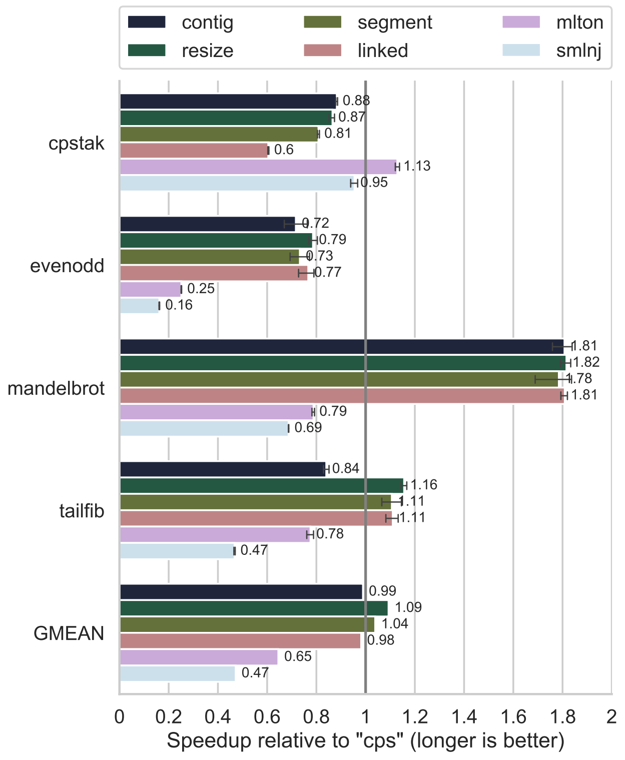

Figure 5. Relative run-times for tail-recursive programs.

For the toy programs in general, the cache miss rates appear to be the cause of poor performance for closure and linked stacks (Figure 4), but for certain programs there are more factors at play.

Beyond the large differences in performance, one takeaway here is that a resizing stack is slightly more efficient on average than a segmented stack. This difference is primarily owed to the fact that a resizing stack turns into a contiguous stack on-demand. In contrast, a segmented stack does not adapt to program behavior, leading to a high number of segment-overflow events in some cases: 349,103 vs 7 in `ack` and 32,020 vs 14 for `quicksort`.

For the tail-recursive programs closure-based stacks match the other strategies on average (Figure 5). Most of the stack-based strategies perform the same, which is expected since these tests do not stress stack allocation, though there are a few programs here with unusual results like for `mandelbrot`. The central piece of the `mandelbrot` benchmark is a triply-nested tail-recursive loop, where the inner two functions reference free variables, requiring many closure allocations. When closure conversion is applied to the `mandelbrot` program *without* direct-style translation, our compiler allocates an extra closure for the return-point of an inner tail-recursive function. When direct-style conversion is enabled, it eliminates such a heap-allocation, which explains why all of the strategies that use the conversion have equal performance for this particular benchmark. It is possible that a more sophisticated closure conversion algorithm would eliminate this extra allocation in all cases [48].

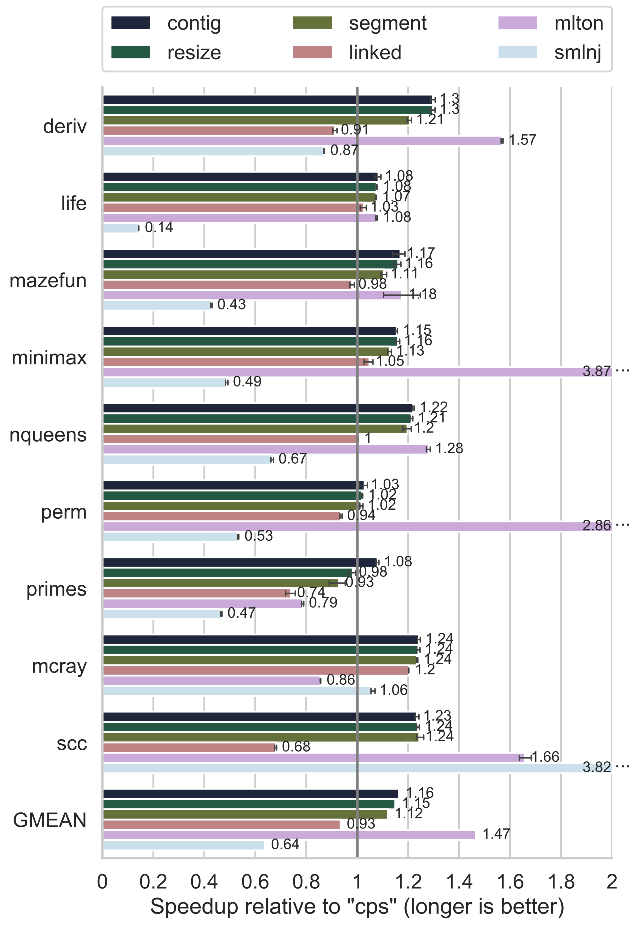

Figure 6. Relative run-times for real programs.

Real Programs. Figure 6 illustrates the performance of each implementation strategy under realistic computational workloads. Overall, this experiment demonstrates is that the magnitude of differences in performance are muted for real programs, *i.e.*, the speedups only vary from 0.93× to 1.16× on average relative to the cps strategy. However, the general ranking of fastest to slowest strategy remains exactly the same as with the toy programs: contig, resize, segment, cps, and linked. For our largest and most complex benchmark, `mcray`, the advantage that resizing stacks has over segmented stacks is erased, as is much of the inefficiency of linked stacks. Data cache miss rates are also nearly equal across the board for the real programs, so the sheer number of extra instructions the cps strategy executes to reallocate frames might be the 16% difference in performance.

5.1.2 Escape Continuation Performance.

Figure 7 shows the relative running times of programs that make heavy use of

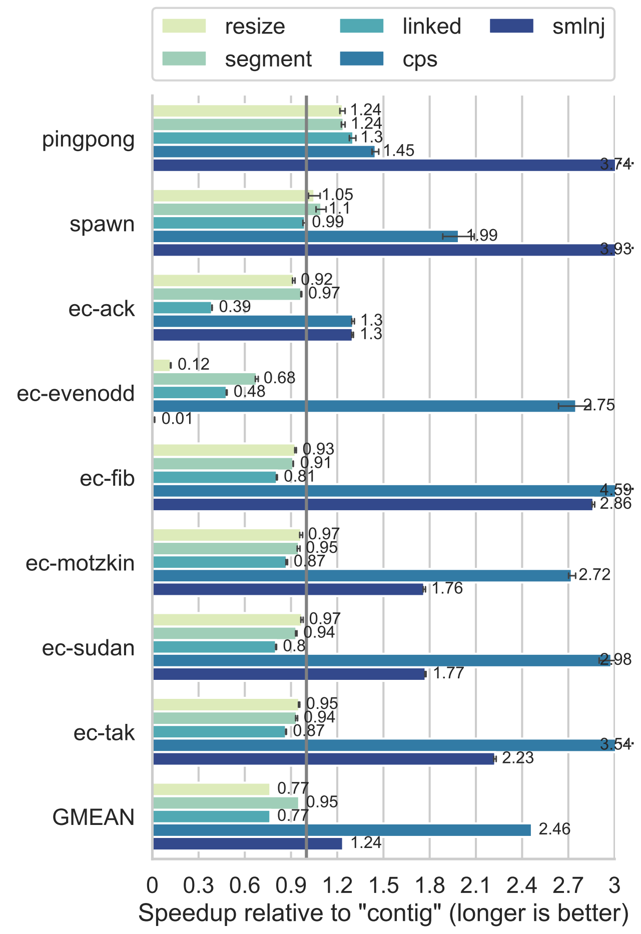

Figure 7. Comparison of trial run-times for the continuation program group.

continuations. MLton was not able to be evaluated because it does not implement the more efficient `callec` and `callcc` is too heavy-weight because it copies the entire stack. The cps strategy is far and away the best strategy among those found in our compiler across the entire suite of continuation-heavy programs. The next best strategy is contig, however, its ability to spawn a huge number of threads is limited by the number of guard pages allowed by the operating system. We had to reduce the number of fork-joins in `spawn` to 10^6 in order for contig to compete.

The `ec` programs replace some or all call and return operations with `callec` and `throw`, stressing the overhead of using continuations for exception handling. While based on a tail-recursive benchmark, `ec-evenodd` grows the stack very deeply (nearly 128MB) because it captures a frame on every other call while subtracting its input. Segmented stacks fare better than resizing stacks because continuation capture in a segment disables the movement of frames due to the creation of an external pointer to the stack (Section 4.5).

5.1.3 Design Trade-offs.

In this section we take a deeper look at different aspects of performance related to implementation strategies for stacks and continuations.

Foreign Function Calls. By default, segmented, resizing, and linked-frame stacks use a separate stack for foreign-function (FF) calls (*i.e.*, a stack-switching shim), whereas the contiguous and closure-based stacks make FF calls directly from the stack already available. We conducted an experiment to help gauge the overhead of stack switching with two benchmarks, `ffi-fib` and `trigfib`, that use the same `fib` program and workload. The integer constants in the original program was changed to foreign-function calls that produce the same value. For `ffi-fib` we call an integer identity function to emphasize a worst-case scenario. The goal of `trigfib` is to call some moderately expensive but common C functions to understand a realistic scenario. For trigfib, we use the trigonometric identities $sin(\pi) = 0$ and $tan(\pi/4) = 1$, which in our compiler are computed as calls to `libm`, to produce the integer constants. The speed-up of having native FFI calls relative to using a shim for `ffi-fib` were huge, 2.44× – 2.91×, but for `ffi-trigfib` the speedup was only 1.03× – 1.06×. Thus for non-pathological programs, trading off the overhead for a simpler and more flexible runtime system may be worthwhile.

Non-native Stack on x86-64. To help understand the benefit of CPU-specific instructions to initialize stack-allocated continuations, we compiled our sequential program suite (Table 1) in two configurations. In one configuration, we replaced all return instructions in ML functions with an equivalent pop-jump sequence:

$$\texttt{retq} \longrightarrow \texttt{popq \%rbx; jmpq *\%rbx}$$

This replacement effectively disables the CPU's return-address stack (RAS), which is an internal bookkeeping mechanism to help predict the target of a return [12]. After this transformation the CPU will rely on its indirect-branch predictor, which levels the playing-field with closure-based stacks. We found that for contig, resize, and segment stacks, using the native CPU instructions yields a 1.06× – 1.08× speedup for toy programs, a 1.01× – 1.03× speedup for real programs, and a 1.04× speedup overall. Linked stacks saw effectively no change in performance even though the implementation uses call and return. Our best guess as to why is that the frequent movement of linked frames might be confusing the predictor.

5.2 Developer's Manifesto

We fount it very difficult to implement *efficient* segmented stacks! Initially we thought eagerly freeing segments outside of a GC cycle on underflow was sufficient, but a number of additional memory-management optimizations were needed to improve their performance for `callec`. In particular, the interaction between continuation capture and frame movement were subtle (Section 4.5). Segmented stacks also have a number of tunable parameters both in terms of segment size, the amount of data copied on overflow, and the limit to set on the stack cache's size. For resizing stacks, we found that the stack-cache lookup operation needed an early-bailout mechanism for its first-fit search because `malloc` already has a better-tuned implementation.

Linked-frame stacks were tricky to implement with a generational garbage collector designed for immutable objects. Namely, mutable objects and their transitive references must normally be promoted to the last generation, in lieu of another write barrier scheme. To allow linked frames in all generations while supporting continuation capture, when returning to a captured frame, the metadata left behind by a promotion is checked for a forwarding pointer before resuming (Section 4.5).

From the perspective of the compiler, implementing the direct-style conversion and using LLVM's GC infrastructure were the only major difficulties of getting non-cps stacks to work, however, these were only needed as a consequence of our compiler's existing design.

The real challenge was in extending the parallel runtime system, where a whole new thread-safe memory management system with caching and a generation-aware mark-sweep collector was developed to deal with non-moving large-objects (*i.e.*, stack regions) for contig, resize, and segment stacks. After implementing a few different custom designs for the allocator, the best allocator ended up being `jemalloc`, because it offered good performance over `glibc`'s malloc.

The other difficulty in the runtime system was with building a correct and efficient stack walker, because garbage-collection occurs frequently and was the bottleneck of several benchmarks. Correctness challenges came from generational scanning and understanding the frame layout metadata output by LLVM. While optimizing the implementation, we found that the most expensive part of scanning was the hash-table lookup to identify the layout of the frame corresponding to the return address. One optimization we made was to replace a modulo with a bitwise-and in the hash-function, which led to a 1.25× speedup for one benchmark.

All of this effort essentially duplicates what the garbage collector will need to do for normal objects in the heap: find pointers, tenure objects, and track modifications. It is certainly the case that continuation closures are much simpler choice for a garbage collected runtime system.

6 Conclusion

If one's primary concern is sequential performance without advanced control-flow mechanisms, then contiguous stacks (or possibly resizing stacks) are clearly the best choice (although the implementation cost of garbage collection support is high). In this work, however, we are interested in compilers for languages that have concurrency, parallelism, or other advanced control-flow mechanisms. In this case, there is no simple answer to the question, "what is the best choice of strategy for my compiler?" It depends on the priorities of the language implementor, which we summarize in the following table, where ✓ means *yes*, ~ means *somewhat*, and ✗ means *no*.

Feature	**contig**	**resize**	**segment**	**linked**	**cps**
Simple (compiler)	✓	~	~	✗	✓
Simple (runtime)	✗	✗	✗	✗	✓
Fast (toys)	✓	✓	✓	✗	✗
Fast (real)	✓	✓	✓	~	~
Space efficient	✗	✓	~	✓	✓
Escape Cont.	~	✗	✗	✗	✓
Foreign Functions	✓	~	~	✗	~
Deep recursion	✗	✓	✓	✓	✓
Tool-compatible	✓	✓	✓	✓	✗

The first two rows compare implementation complexity, the next three compare performance, and the remainder compare feature support.

First, if the ease of implementation in the compiler and runtime system are of utmost priority, continuation closures provide the simplest implementation. They are also well suited to implement concurrency features without any of the limitations of escape continuations. Their downside is poorer sequential performance[5] and the fact that they are not compatible with existing profilers and debuggers, which expect a traditional call stack.

While resizing and segmented stacks share much of the same overhead and characteristics, a resizing stack is the better choice because of its space efficiency. The segmented stack is not space efficient because the segment size is constant and constrained by the efficiency of the overflow and underflow handlers. Whereas we can pick a small initial size for a resizing stack with the knowledge that with only a few overflows, it will become large enough to support the program's full call stack.

A design that we considered, but did not have time to implement is a resizing segmented stack. In this strategy, a thread is created with a small, resizing stack. Once the stack grows to the size of a segment (*i.e.*, 64Kb), we then switch modes to behave like a segmented stack. This approach has the advantage that it can support vary large numbers of threads at fairly low memory cost (assuming that they are not all executing some deeply recursive computation), while avoiding the increasing cost of stack copying when there is deep recursion. Furthermore, we can reclaim stack memory (*i.e.*, segments) after a deep recursion terminates.

Finally, we conclude that linked-frame stacks should always be avoided in favor of other approaches, because the mutability of linked-frames is more of a curse than a benefit relative to closure-based stacks.

[5] A substantial part of this cost may be regained by more sophisticated compilation techniques [9, 48], but that increases the compiler's complexity.

References

[1] Brian Anderson. 2013. Abandoning segmented stacks in Rust. https://mail.mozilla.org/pipermail/rust-dev/2013-November/006314.html

[2] Todd A. Anderson. 2010. Optimizations in a private nursery-based garbage collector. In *Proceedings of the 2010 international symposium on Memory management* (Toronto, Ontario, Canada). ACM, New York, NY, 21–30. https://doi.org/10.1145/1806651.1806655

[3] Andrew W. Appel. 1987. Garbage collection can be faster than stack allocation. *Inform. Process. Lett.* 25, 4 (1987), 275 – 279. https://doi.org/10.1016/0020-0190(87)90175-X

[4] Andrew W. Appel. 1989. Simple generational garbage collection and fast allocation. *Software – Practice and Experience* 19, 2 (1989), 171–183.

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

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

[7] A. W. Appel and T. Jim. 1989. Continuation-passing, Closure-passing Style. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '89)*. ACM, New York, NY, USA, 293–302. https://doi.org/10.1145/75277.75303

[8] Andrew W. Appel and D. B. MacQueen. 1991. Standard ML of New Jersey. In *Programming Language Implementation and Logic Programming (Lecture Notes in Computer Science)*, Vol. 528. Springer-Verlag, New York, NY, 1–26.

[9] Andrew W. Appel and Zhong Shao. 1992. Callee-save Registers in Continuation-passing Style. *Lisp and Symbolic Computation* 5, 3 (Sept. 1992), 191–221. https://doi.org/10.1007/BF01807505

[10] Andrew W Appel and Zhong Shao. 1996. An Empirical and Analytic Study of Stack vs. Heap Cost for Languages with Closures. *Journal of Functional Programming* 6, 01 (1996), 47–74.

[11] Joe Armstrong. 2007. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, Raleigh, NC.

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

[13] Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. 1996. Representing control in the presence of one-shot continuations. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96)*. ACM, New York, NY, 99–107.

[14] Perry Cheng, Robert Harper, and Peter Lee. 1998. Generational Stack Collection and Profile-driven Pretenuring. *SIGPLAN Not.* 33, 5 (May 1998), 162–173. https://doi.org/10.1145/277652.277718

[15] Will Clinger, Anne Hartheimer, and Eric Ost. 1988. Implementation Strategies for Continuations. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming* (Snowbird, Utah, USA) *(LFP '88)*. ACM, New York, NY, USA, 124–131. https://doi.org/10.1145/62678.62692

[16] William D. Clinger and Lars T. Hansen. [n.d.]. The Larceny Project. ([n. d.]). Available at http://www.larcenists.org.

[17] William D. Clinger, Anne H. Hartheimer, and Eric M. Ost. 1999. Implementation Strategies for First-Class Continuations. *Higher-Order and Symbolic Computation* 12, 1 (1999), 7–45. https://doi.org/10.1023/A:1010016816429

[18] Olivier Danvy and Julia L. Lawall. 1992. Back to Direct Style II: First-class Continuations. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming* (San Francisco, California, USA) *(LFP '92)*. ACM, New York, NY, USA, 299–310. https://doi.org/10.1145/141471.141564

[19] Edsger W Dijkstra. 1960. Recursive programming. *Numer. Math.* 2, 1 (1960), 312–318.

[20] Amer Diwan, J. Eliot B. Moss, and Richard L. Hudson. 1992. Compiler support for garbage collection in a statically typed language. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '92)*. ACM, New York, NY, 273–282.

[21] Damien Doligez and Georges Gonthier. 1994. Portable, unobtrusive garbage collection for multiprocessor systems. In *Conference Record of the 21st Annual ACM Symposium on Principles of Programming Languages (POPL '94)* (Portland, Oregon, United States). ACM, New York, NY, 70–83.

[22] Damien Doligez and Xavier Leroy. 1993. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Conference Record of the 20th Annual ACM Symposium on Principles of Programming Languages (POPL '93)* (Charleston, South Carolina, United States). ACM, New York, NY, 113–123.

[23] Alan A.A. Donovan and Brian W. Kernighan. 2015. *The Go Programming Language* (1st ed.). Addison-Wesley, Reading, MA.

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

[25] Kathleen Fisher and John Reppy. 2002. *Compiler support for lightweight concurrency*. Technical Memorandum. Bell Labs. Available from http://moby.cs.uchicago.edu/.

[26] GHC. [n.d.]. The Glasgow Haskell Compiler. ([n. d.]). Available at http://www.haskell.org/ghc.

[27] Simcha Gochman, Ronny Ronen, Ittai Anati, Ariel Berkovits, Tsvika Kurts, Alon Naveh, Ali Saeed, Zeev Sperber, and Robert C Valentine. 2003. The Intel Pentium M Processor: Microarchitecture and Performance. *Intel Technology Journal* 7, 2 (2003), 21–36.

[28] Marcelo J. R. Gonçalves and Andrew W. Appel. 1995. Cache Performance of Fast-allocating Programs. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture* (La Jolla, California, USA) *(FPCA '95)*. ACM, New York, NY, USA, 293–305. https://doi.org/10.1145/224164.224219

[29] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. 1984. Continuations and coroutines. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*. ACM, New York, NY, 293–298.

[30] Matthew Hertz and Emery D. Berger. 2005. 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* (San Diego, CA, USA) *(OOPSLA '05)*. ACM, New York, NY, USA, 313–326. https://doi.org/10.1145/1094811.1094836

[31] R. Hieb, R. Kent Dybvig, and C. Bruggeman. 1990. Representing control in the presence of first-class continuations. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '90)*. ACM, New York, NY, 66–77.

[32] Richard A. Kelsey. 1995. A Correspondence Between Continuation Passing Style and Static Single Assignment Form. In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations* (San Francisco, California, USA) *(IR '95)*. ACM, New York, NY, USA, 13–22. https://doi.org/10.1145/202529.202532

[33] Matthew Le and Matthew Fluet. 2015. Partial Aborts for Transactions via First-class Continuations. In *ICFP '15* (Vancouver, BC, Canada). ACM, New York, NY, 230–242. https://doi.org/10.1145/2784731.2784736

[34] Peng Li, Simon Marlow, Simon Peyton Jones, and Andrew Tolmach. 2007. Lightweight concurrency primitives for GHC. In *Proceedings of the 2007 Haskell Workshop* (Freiburg, Germany). ACM, New York, NY, 107–118.

[35] LLVM Developers. 2019. Garbage Collection Safepoints in LLVM. https://releases.llvm.org/9.0.0/docs/Statepoints.html

[36] James S. Miller and Guillermo J. Rozas. 1994. *Garbage Collection is Fast, but a Stack is Faster,*. Technical Report AIM-1462. MIT AI Laboratory. https://apps.dtic.mil/docs/citations/ADA290099

[37] MLton. [n.d.]. The MLton Standard ML compiler. ([n. d.]). Available at http://mlton.org.

[38] Mikael Pettersson, Konstantinos Sagonas, and Erik Johansson. 2002. *The HiPE/x86 Erlang Compiler: System Description and Performance Evaluation*. Springer Berlin Heidelberg, Berlin, Heidelberg, 228–244. https://doi.org/10.1007/3-540-45788-7_14

[39] Norman Ramsey. 1990. *Concurrent programming in ML*. Technical Report CS-TR-262-90. Department of Computer Science, Princeton University.

[40] Norman Ramsey and Simon Peyton Jones. 2000. Featherweight concurrency in a portable assembly language. (Nov. 2000). Unpublished paper available at https://www.cs.tufts.edu/~nr/pubs/c--con-abstract.html.

[41] Keith Randall. 2013. Contiguous Stacks in Go. http://golang.org/s/contigstacks

[42] John Reppy. 2002. Optimizing nested loops using local CPS conversion. *Higher-order and Symbolic Computation* 15 (2002), 161–180.

[43] John Reppy, Claudio Russo, and Yingqi Xiao. 2009. Parallel Concurrent ML. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming ICFP '09* (Edinburgh, Scotland, UK). ACM, New York, NY, 257–268. https://doi.org/10.1145/1596550.1596588

[44] John H. Reppy. 1991. CML: A higher-order concurrent language. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '91)*. ACM, New York, NY, 293–305.

[45] John H. Reppy. 1999. *Concurrent Programming in ML*. Cambridge University Press, Cambridge, England.

[46] John C. Reynolds. 1993. The discoveries of continuations. *Lisp and Symbolic Computation* 6, 3-4 (1993), 233–248.

[47] Zhong Shao and Andrew W. Appel. 1994. Space-efficient Closure Representations. *SIGPLAN Lisp Pointers* VII, 3 (July 1994), 150–161. https://doi.org/10.1145/182590.156783

[48] Zhong Shao and Andrew W. Appel. 2000. Efficient and safe-for-space closure conversion. *ACM Transactions on Programming Languages and Systems* 22, 1 (2000), 129–161.

[49] K. C. Sivaramakrishnan, Lukasz Ziarek, and Suresh Jagannathan. 2014. MultiMLton: A multicore-aware runtime for standard ML. *Journal of Functional Programming* FirstView (July 2014), 1–62.

[50] Darko Stefanovic and J. Eliot B. Moss. 1994. Characterization of Object Behaviour in Standard ML of New Jersey. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming* (Orlando, Florida, USA) *(LFP '94)*. ACM, New York, NY, USA, 43–54. https://doi.org/10.1145/182409.182428

[51] James M. Stichnoth, Guei-Yuan Lueh, and Michal Cierniak. 1999. Support for Garbage Collection at Every Instruction in a Java Compiler. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '99)*. ACM, New York, NY, 118–127.

[52] Yngve Sundblad. 1971. The Ackermann function. a theoretical, computational, and formula manipulative study. *BIT Numerical Mathematics* 11, 1 (March 1971), 107–119. https://doi.org/10.1007/BF01935330

[53] Mitchell Wand. 1980. Continuation-based multiprocessing. In *Conference Record of the 1980 ACM Conference on Lisp and Functional Programming*. ACM, New York, NY, 19–28.

[54] Andy Wingo. 2014. stack overflow. https://wingolog.org/archives/2014/03/17/stack-overflow