From Folklore to Fact: Comparing Implementations of Stacks and Continuations

Kavon Farvardin
Computer Science
University of Chicago
Chicago, IL, USA
kavon@cs.uchicago.edu

John Reppy
Computer Science
University of Chicago
Chicago, IL, USA
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literature does not provide much guidance, since previous studies suffer from various flaws in methodology and are outdated for modern hardware.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 six 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.

CCS Concepts: • **Software and its engineering** → *Multithreading*; *Parallel programming languages*; **Concurrent programming languages**; *Functional languages*; *Recursion*; *Concurrent programming structures*; *Procedures, functions and subroutines*; **Runtime environments**.

Keywords: Call stacks, Concurrency, Continuations, Compilers, Functional Programming

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, 16 pages. 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 [21], segmented stacks [39], and heap-allocated continuation closures [5], but semantically they can all described in terms of *continuations* [55].

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 [19], simulations and theoretical analysis [10], or direct measurements performed 30 years ago [17]. 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! When weighing one strategy against another, it is crucial to have a deep understanding of the design space. 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, 50, 58, 62].

In fact, the problem of segment thrashing was solved by Bruggeman *et al.* in the Chez Scheme compiler [15]. A subtle aspect of their solution is its effectiveness *only* for runtime systems that do not allow pointers into the stack, which is typical for garbage-collected languages.

This paper attempts to remedy this situation by providing an "apples-to-apples" comparison of six different approaches to implementing call stacks and continuations. The comparison is performed using the Manticore system [14, 30], which implements a parallel and concurrent subset of Standard ML [32]. We use the same source language, compiler pipeline, LLVM-backend, and runtime system for every approach, with the only differences being those required by the particular 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 (CML) [54], Erlang [11], and Go [25] 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 a detailed discussion of the implementation space in Section 4, including a discussion of 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 comparison of implementation strategies was by Clinger *et al.* [17] 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 Standard ML of New Jersey (SML/NJ) compiler switched to using heap-allocated continuation closures in its implementation [7]. Appel 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 opinion was somewhat controversial [45]. In later work, Shao and Appel presented a rigorous argument that continuation closures were more efficient than a traditional stack-based implementation [10]. 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 that the instruction counts for stack-frame initialization was inflated because of no frame reuse for non-tail calls [19, Section 7].

In 1999, Clinger *et al.* [19] 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 made 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 [36] 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 [59] 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 [38] found that the regular compaction offered by such a nursery is also beneficial to cache locality versus other schemes for heap allocation, but it is unclear whether this benefit can match the efficiency of stack allocation.

3 Background

This section surveys some necessary background. We start by briefly introducing the concept of *continuations* and the use of continuation-passing style in a compiler's intermediate language (IR). We then motivate this choice with some simple examples of how such an IR can implement advanced control-flow mechanisms. We then describe the Manticore system, which provides our experimental testbed and finish with a description of the six call-stack implementation strategies that we compare.

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

Figure 1. A continuation-passing-style IR.

3.1 Continuations

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.

3.2 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 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 SML:

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

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

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 fact to return a result to its caller (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.

3.3 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.

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 [15]; 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 [15], reifies one-shot continuations of unlimited lifetime.[3] An *escape* continuation is a one-shot continuation whose lifetime is limited to its lexical scope [29, 49]. Escape continuations are simpler to implement due to their restrictions, yet still powerful enough to implement context-switching operations.

3.4 Programming with Continuations

There are many examples in the literature of using reified continuations to implement a wide range of advanced control-flow mechanisms [26, 37, 41, 48, 52, 53, 61]. Such implementations could either be user-level code or part of a compiler's implementation of language features. To provide a bit of

[2]Note that in our implementation, we add an additional continuation parameter to represent the current exception continuation.

[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.

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 -> unit) -> '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.

We illustrate these mechanisms with a simple coroutine implementation: A suspended coroutine's state can be represented as a unit-valued continuation and we assume a global scheduling queue with `enqueue` and `dequeue` operations.

```
type coroutine = unit econt
val enqueue : coroutine -> unit
val dequeue : unit -> coroutine
```

Then we can define a function to schedule the next ready coroutine.

```
fun dispatch () = throw (dequeue ()) ()
```

The `yield` function for switching coroutines is

```
fun yield () = callec (fn k => (
      enqueue k;
      dispatch ())
```

And creating a new thread is implemented as

```
fun spawn (f : unit -> unit) = let
      val k = newStack f
      in
        enqueue k;
        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.

3.5 The Manticore System

The Manticore system [14, 30] implements a parallel dialect of ML (called PML) that includes support for Concurrent ML-style message passing [52]. 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. The additional stack-based strategies added to Manticore only support Concurrent ML, because our compiler's current desugaring of implicit parallelism assumes there are no lifetime restrictions on continuations.

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 CPS representation with explicit continuation closures [7]. The first-order representation supports both closure passing to implement call/return linkage and traditional function calls.

For the stack-based runtime models, we modify closure conversion using a combination of ideas from Danvy and Lawall [20], Kelsey [40] and Reppy [51] to map non-tail calls to traditional call operations and to introduce `callec` operations to reify any escape continuations that are present.

The first step of this process involves an analysis pass over the CPS IR to classify continuation bindings and function applications. A local continuation is one whose uses are strictly limited to the function in which it is bound, and only as either the target of a **throw**, or passed as a return continuation in an **apply**. All non-local continuations are considered escape continuations, which are wrapped with `callec` in a transformation following analysis. First, the CPS IR is augmented with a new expression construct **callec** (f/k), which represents a special application of function f to a reified version of continuation k. Then, for an escape continuation binding **cont** $escK$(x) = e_1 **in** e_2, we replace e_2 with the expression:

```
fun f (escK′ / retK) = e₂
  in
    callec (f, escK)
```

Now, the escape continuation has only one use as the return continuation in a function application, so it is reclassified as a local continuation. Details about how we prevent previously local continuations used in e_2 from becoming non-local after this transformation are described by Farvardin [27]. Finally, closure conversion continues as usual, except that an **apply** that passes a return continuation different than the one bound in its enclosing function is mapped as a non-tail call.

The original Manticore compiler used the MLRisc framework [33, 42] to generate code for the Intel x86-64 architecture, but for this work we have replaced this code generator with LLVM [27, 28].

Each "virtual processor" (VProc) in the Manticore runtime system corresponds to one process-level thread, *i.e.* a `pthread`. A private thread-scheduling queue is assigned to each VProc, which relies on a timer for thread preemption. For the purposes of this paper, we use only one VProc for the CML benchmarks in order to compare fairly with SML/NJ and to focus on each strategy's overheads.

The Manticore garbage collector and heap architecture, which are optimized for parallel scalability [12], are the most relevant aspects of the runtime for this work. Each VProc in the system has a private heap that is divided into a nursery and an "old" space. In addition, there is a global heap that is shared across all VProcs. Minor collections copy data from the nursery into the old space and major collections copy data from the old space to the global heap (this part of the design is a modified version of Appel's semi-generational collector [4]). Global collections are done in parallel for the global heap. Following the approach of the Doligez-Leroy-Gonthier (DLG) collector [23, 24], we maintain the invariant that there are no pointers from the global heap into the private heaps (and no pointers from the old space into the nursery). To maintain this invariant, we must *promote* values to the global heap in order to make them globally visible (*e.g.*, when sending a message on a channel or when updating a mutable reference). For pure sequential code, like most of the programs that we benchmarked, this memory system basically works like a simple generational copying collector and performs well. The design does have a significant impact on a few of the benchmarks, which we discuss in Section 5.

3.6 Strategies

We study six different implementation strategies, ranging from traditional contiguous stacks of fixed size to heap-allocated continuation closures. The six 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 large, fixed-size contiguous region, which is the native stack discipline used by C.
- **resize** — stack frames are allocated in a contiguous region that is initially small. The stack is copied to a new region that is twice as large when the stack overflows.
- **segment** — stack frames are allocated in contiguous fixed-size segments, which are linked together.
- **hybrid** — a hybrid scheme that uses a resizable stack until the the stack grows to the size of a segment, at which point the call stack is managed as a segmented stack.
- **linked** — each individual stack frame is allocated as a mutable heap object and the stack is represented as a linked list of frames. This scheme should not be confused with a "linked closure" representation for nested functions [56].
- **cps** — is a direct translation of the first-order CPS representation where return continuations are represented by immutable, heap-allocated continuation closures. This approach was used in the original implementation of Manticore [28, 31] and is similar that used in SML/NJ [5].

The first five of these strategies, which we call the *stack strategies*, implement return 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

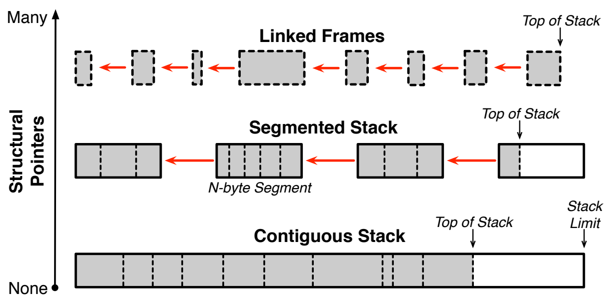

Figure 2. High-level spectrum of call stack implementations, as a function of the number of pointers used to maintain the structure. Dashed lines separate stack frames.

represented, where the representation can be characterized by how often pointers are used to link frames together as shown in Figure 2. At one end of the spectrum, contiguous stacks (including resizable stacks) require no pointers. The segmented and linked strategies require varying numbers of linking pointers: one per segment vs. one per frame. More pointers results in more runtime overhead to function calls, but also reduces space efficiency.

4 Implementation Issues

A major concern for compiler designers is the efficiency of the call stack for sequential programs. The five 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 use the hardware instructions for call/return, which can lead to better branch prediction (see Section 5.2.2). Where they differ is in their complexity of implementation (both compiler and runtime), their memory footprint for heavily threaded applications, and the overhead needed to check for and handle stack overflow. The **cps** strategy, on the other hand, provides the simplest implementation and very efficient support for advanced control-flow features, but it increases memory allocation rates.

In the remainder of this section, we discuss the technical issues and challenges posed by the six strategies.

4.1 Stack Overflow

Functional programs often make use of deep recursion, such as non-tail-recursive list traversals, which means that their implementations must be concerned with the potential for *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 function prologues.

The **contig** strategy reserves a fixed-size stack area that cannot be expanded 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 provisioning threads with very large stacks to reduce the likelihood of stack overflow. 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, which results in the program being terminated.

The **resize**, **segmented**, and **hybrid** strategies use explicit stack-limit checks to detect when the stack must grow.[4] For the **resize** strategy, we copy the stack into a new region that is twice the current region's size. For the **segmented** strategy, we allocate a new segment and copy a few frames from the caller's segment to the new segment, which reduces the likelihood of thrashing between two segments. We allocate segments from the *stack cache* of recently freed stack memory objects, which reduces allocation time and improves cache locality in some situations.

The **linked** and **cps** strategies do not require stack overflow checks, since they use heap allocation.

4.2 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), one can omit frame allocation for them 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 overhead of a failed check.

There are also techniques for reducing the amount of allocation required for continuation closures. These include adding dedicated "callee-save" registers to the calling convention that can be used to pass live values from the caller to its return continuation [9]. Another effective technique uses linked representations of continuation closures, instead of flat closures [57]. Each of these techniques has non-trivial implementation costs in the compiler, so we have not implemented them in our testbed, but these techniques individually produced runtime improvements in the 10–20% range in the SML/NJ compiler.

4.3 Finding GC Roots

Accurately identifying live heap pointers, or *roots*, in the mutator's state is necessary for garbage collected languages. 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 need a way to identify the live roots at each non-tail call site in the program. This information can be maintained without runtime overhead by generating a map from return addresses to root-location data at compile time [22]. The garbage collector uses this map to lookup a 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 [39, Figure 4].

The use of callee-save registers adds another layer of complexity to 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. [16] 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 compiler.

For the stack-based strategies, we rely on LLVM's support for precise garbage collection in the presence of frame sharing and reuse [44]. 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 allocation rate of ephemeral data. Cheng et al. [16] 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 allocated 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. The collector will stop scanning a stack once it sees a watermark that is older than the current generation being reclaimed. This point represents

[4]Guard pages are a potential alternative, but they suffer from two issues: first, the operating system may place limits on the number of protected pages in a process's address space and, second, it can be difficult to robustly recover from a guard-page memory fault, since it requires precise information about the state of the stack at the exact point of the fault.

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, because the mutator only adds a fresh nursery watermark *when the frame is allocated*. Thus, a function that has completed a non-tail call between collections, but has not yet returned, will have an older watermark in its frame than its data requires.

The **cps** strategy requires no special support for stack scanning, since the continuation closures are ordinary immutable heap objects that the garbage collector already handles. The root set are just the live pointer-containing registers at the point where GC is invoked.

4.4 CPU and ABI Support

Historically, instruction sets such as the x86 have contained dedicated instructions to assist with function calls and frame initialization. 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 [35]. We found the overhead to be lower than reported by Pettersson et al. [47] on modern x86-64 processors (Section 5.2.2).

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 4.1), 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 **resize**, **segmented**, and **hybrid** 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-frame stacks. In the **cps** strategy, the stack-pointer register is free and used to hold the foreign-function stack, which is where we spill registers between Manticore function calls and to perform foreign-function calls with no overhead.

4.5 Thread Creation

Implementing first-class continuations for the stack strategies is difficult, so we limit them to the escape continuations (*i.e.*, `callec`), which means that we must provide the `newStack` primitive to create a fresh execution context for these strategies. The `newStack` function allocates a fresh memory object for the stack (the size depends on the strategy) with one initial frame that exits when invoked. Then, we push a frame that contains the given function f and a return address for code that expects a value to be returned, after which it will apply f to that value. The resulting stack is packaged as an escape continuation that will execute the function. when it is thrown to.

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

```
fun yield () = callec (fn k => (
      enqueue k;
      dispatch ()))
```

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 [29]. Another processor attempting to schedule the continuation `k` would end up spinning on the lock until it had been released. Li et al. [43] proposed to avoid this race by modeling the exchange with software-transactional memory, although this approach turned out to be too costly 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 **cps** 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.

4.6 Implementing `callec`

The implementation of `callec` and `throw` for the **contig** strategy is fairly straightforward. Essentially, `callec` is equivalent to C's `setjmp` function and `throw` is equivalent to `longjmp`.

The **resize**, **segmented**, and **hybrid** strategies were the most challenging 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.

Originally we implemented `callec` just as Bruggeman et al. [15] describes `call1cc`, where an early-overflow with no frame movement occurs instead of segment splitting. We found this to be extremely slow in programs that make heavy use of `callec`. Bruggeman et al. [15]'s implementation of `callcc` seals 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 whole 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 marking technique acts as a conservative approximation of Bruggeman et al. [15]'s segment-splitting technique for `callcc`. For example, during overflow a resizing stack will allocate a region that is twice the size of the previous as usual, but if the previous region is marked as non-copyable, no frames are moved and instead an underflow frame is pushed to the new stack region that will dispatch control to the previous stack region when invoked.

Eager memory management was also crucial for the efficiency of the **resize**, **segmented**, and **hybrid** strategies. 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` for the **linked** strategy is also tricky, because of the need to support promotion to maintain the DLG heap invariant for mutable heap-allocated frames. The challenge again stems from the need to handle an additional pointer to a frame, which can only be produced via `callec`. If a continuation is used for synchronization in CML, then the continuation must be promoted to the global heap in order to be enqueued in the channel. During the capture of a linked frame with `callec`, 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 for linked stacks, the launcher now also serves as a read-barrier during a throw: the launcher's code always checks the GC tag of the frame pointed-to by the launcher, following the forwarding pointer if needed to find to the correct version of that frame. Once garbage collection occurs, live 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 evaluation of the six different strategies for implementing continuations discussed in Section 3.6. We start with an empirical analysis of performance using a variety of benchmark programs. We then discuss certain qualitative design trade offs in Section 5.2 and conclude with some recommendations for compiler developers in Section 5.3.

5.1 Performance

We use five different groups of benchmarks to evaluate the various strategies. The benchmarks are summarized in Table 1, with horizontal lines separating the groups. The majority of the benchmarks were ported from the benchmark suites of Larceny Scheme [18], the Glasgow Haskell Compiler [34], or the SML compilers MLton [46] 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 arithmetic mean of ten trials with 95% bootstrapped confidence intervals plotted. The running time of a trial is 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, load the input, *etc.*

5.1.1 Experiment Setup. All performance data was collected using a lightly-loaded server equipped with two Intel Xeon Gold 6142 CPUs and 64 GB 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 and clean-up (*e.g.*, `instcombine`, `simplifycfg`) to ensure a fair comparison, because LLVM's advanced optimizations are better suited for direct-style code. The production-grade compilers MLton version 20180207 and SML/NJ version 110.96 (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

Table 1. Details for all of the benchmarks. The columns are the benchmark name, lines of code, source benchmark suite, description, problem size, and number of iterations per run. The benchmarks are divided into **recursive**, **looping**, **mixed**, **continuation**, and **foreign function** groups. A name marked with a star (⋆) means there is also a version of the benchmark in the continuation group that uses `callec` and `throw`.

Program	LOC	Origin	Description	Input (n)	Iters (i)
ack ⋆	9	Larceny	Compute the Ackermann function	(3, 11)	4
divrec	13	Larceny	Recursively divide n by two, where n is represented by a list of length n.	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	Manticore	Find n^{th} Motzkin number	21	4
primes	19	Larceny	Compute list of primes less than n	1,000	20,000
quicksort	18	Manticore	Quicksort a fixed random-number list of length n	2×10^6	1
sudan ⋆	11	Manticore	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
barnes-hut	259	SML/NJ	Perform i steps of hierarchical n-body solver	400,000 particles	10
deriv	70	Larceny	Symbolic differentiation	$3x^2 + ax^2 + bx + 5$	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)	10,000
minimax	128	Manticore	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
mcray	513	Manticore	Functional raytracer; Monte Carlo algorithm	300×200 canvas	1
scc	44	GHC	Strongly-connected components of n vertex graph	5,000	3
cml-pingpong	20	Manticore	Two threads send n messages over a channel	4×10^6	1
cml-ring	20	KRoC	A ring of n threads forwarding a message for i cycles	128	15,000
cml-spawn	9	Manticore	Fork a new thread and then sync immediately n times	2^{22}	1
ec-⋆			The escape-continuation forms of benchmarks from above.		
ffi-fib	7	Manticore	`fib` with FFI calls to integer identity function	40	1
ffi-trigfib	12	Manticore	`fib` with FFI calls to trigonometry functions	40	1

stack,[5] and SML/NJ uses heap-allocated continuation closures.

Most of the stack strategies have various size parameters that need to be selected. For **contig**, we used a size of 128 MB, which was the smallest size that could accommodate all of the benchmarks without overflowing. The **resize** strategy starts with an initial stack size of 8 KB, which was chosen empirically by first finding the smallest size such that a larger initial size for yielded no benefit for the deeply-recursive **ack** benchmark. The **segmented** strategy uses a segment size of 64KB, which was picked because it is 8 times larger than the initial resize stack. During overflow for a segmented stack, frames are copied until either a maximum of four frames or one-eighth of a segment's data is encountered. Stack memory was allocated using `aligned_alloc` as provided by the **jemalloc** library (Version 3.6). During program start-up (and prior to running-time measurement), the segment cache is pre-loaded with 64 free segments if using **resize**, **segmented**, or **hybrid** strategies.

[5]In contrast to our resizing stack, MLton stack's grows in the opposite direction (upwards) on x86-64. Thus, MLton uses `jmp` instructions for call/return and does not use `push`/`pop` for frame initialization.

5.1.2 Recursion Performance.

Together the first three groups of functional benchmarks listed in Table 1 evaluate all aspects of function call overhead. The first group consists of **recursive** benchmarks that exhibit a variety of patterns for deep recursion where the work performed between each call is minimal. The second group of benchmarks are referred to as **looping** because they exclusively make use of tail recursion, which does not grow the call stack but does stress tail-call optimization. The third group consists of **mixed** benchmarks that provide a wider variety of workloads that are more typical of regular programs.

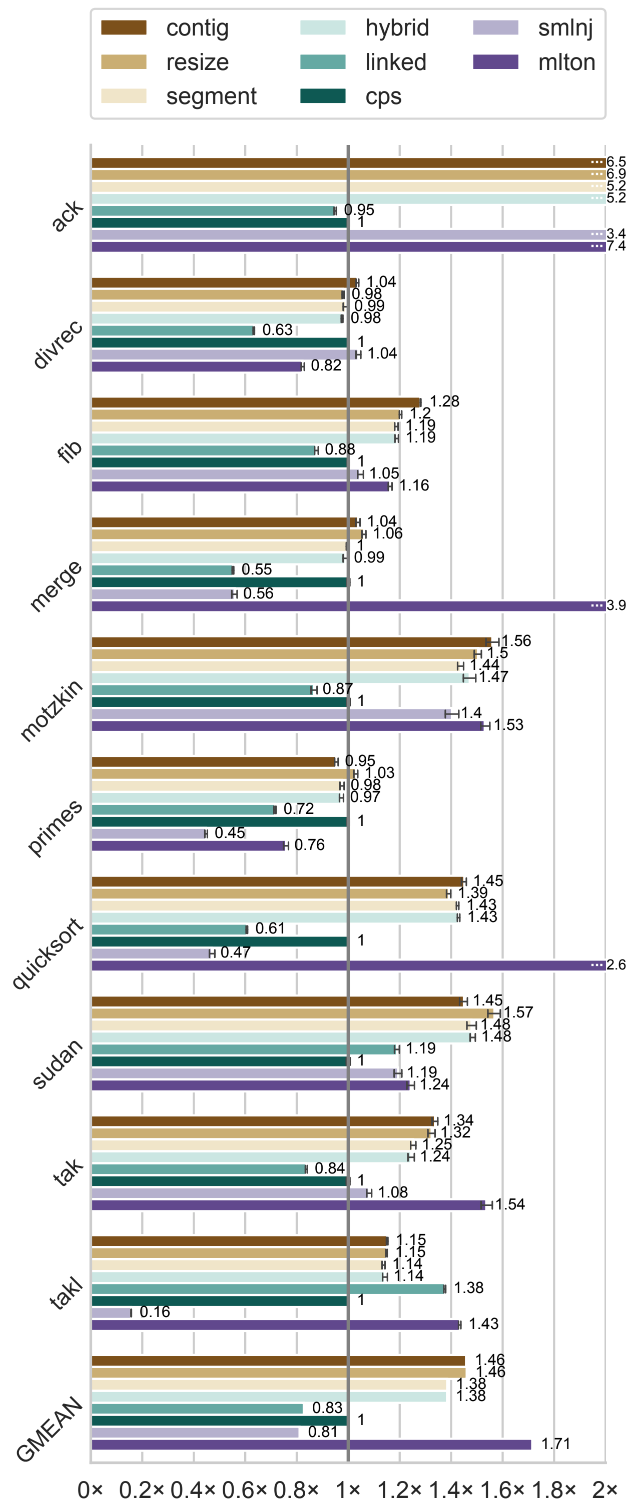

Figure 3. Speed-up comparison over **cps** for the **recursive** benchmarks.

Recursive Benchmarks. Figure 3 shows that on average the **contig**, segmented, resizing, and hybrid stacks, are *significantly* faster for the recursive benchmarks, exhibiting a 1.38× to 1.46× speedup relative to **cps**. Linked stacks are the slowest, seeing a 0.83× speedup, *i.e.*, a 1.2× slowdown relative to **cps**.

The **ack** results are a major outlier, with most strategies running over 5.2× faster than **cps**, owing to the program's unique pattern of recursion. A typical recursive pattern is similar to depth-first traversal of a tree; *e.g.*, **fib** and **tak** traverse a space similar to a complete k-ary tree of depth n, where up to k^n calls may 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 has a less predictable recursion pattern [60].

For smaller inputs, **ack** has a peak-and-valley pattern of recursive depth history, making it a nightmare scenario for the **linked** and **cps** strategies, which that allocate directly in the heap: 67% of running time for these two strategies was spent in the garbage collector, while for the other strategies it was less than 2%. For **cps**, 24 GB of data was allocated in the nursery, with 65.6% of that being copied to the old space and 40% later copied to the global heap. The **linked** strategy saw similar rates of live-data copying, but even with frame reuse, it still allocated 2.7 GB more data in the heap than the **cps** strategy.

Beyond the large differences in performance, one takeaway is that the **resize** strategy is slightly more efficient on average than **segmented** or **hybrid**. 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. For example, the **segmented** strategy handles 349,103 overflow events for **ack** and 32,020 for **quicksort**, compared with 7 and 14 (resp.) for the **resize** strategy. While the **hybrid** strategy is adaptive, its design goal is to reduce memory overhead for the situation where there are large numbers of small threads that can run in relatively small stacks.[6] It might make sense, however, to use larger segments for the **hybrid** strategy, since they are only used when the computation has already exhibited fairly deep recursion.

Looping Benchmarks. For the looping benchmarks, **cps** matches the other strategies on average (Figure 4). 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.

[6]This pattern is typical of many programs in languages like CML, Erlang, and Go.

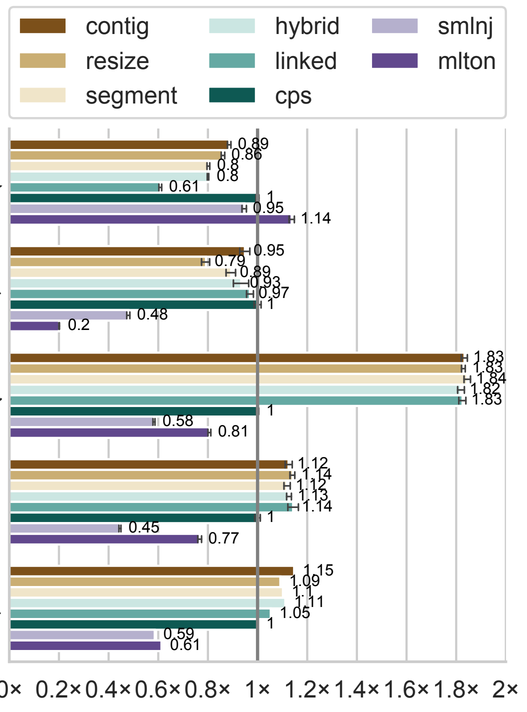

Figure 4. Speed-up comparison over **cps** for the **looping** benchmarks.

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 **mandelbrot** *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 [57].

From a theoretical point of view, one might expect that the performance should be exactly the same across the board for benchmarks such as the virtually no-op **evenodd** because no frame or heap allocation should be needed. When comparing the assembly code across the stack-based strategies for **cpstak**, **evenodd**, and **tailfib**, the only reason to which we can attribute the varying results are slightly different basic-block layout decisions made by LLVM's code generator, which is influenced by each strategy's unique function prologues.

A prologue to allocate a frame is needed in the looping benchmarks because in our runtime system, preemptions to handle signals occur at garbage collection safe-points. Thus even non-allocating tail-recursive loops must contain a heap-limit test and a possible call to enter the garbage collector. Tail-recursion elimination helps ensure that for stack-based strategies, the frame is only set-up once during the first iteration of the loop.

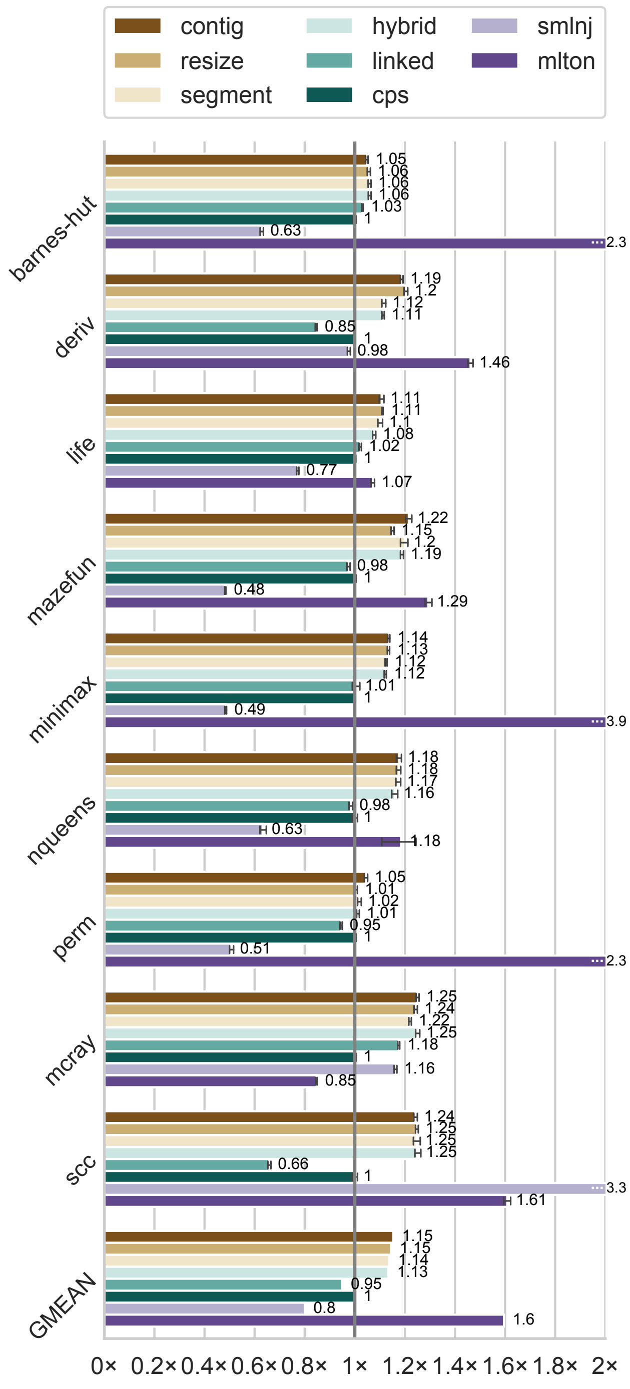

Figure 5. Speed-up comparison over **cps** for the **mixed** benchmarks.

Mixed Benchmarks. Figure 5 illustrates the performance of each implementation strategy for the mixed benchmarks, which are more typical of regular programs than the recursive or looping benchmarks. Overall, this experiment demonstrates is that the magnitude of differences in performance are muted for the mixed benchmarks; *i.e.*, the speedups only vary from 0.95× to 1.15× on average relative to the **cps** strategy. The general ranking of fastest to slowest remains roughly the same as with the recursive benchmarks: **contig** and **resize**, then **segmented** and **hybrid**, **cps**, and finally **linked**. For our largest and most complex benchmark, **mcray**, the advantage that **resize** has over **segmented** is virtually erased, and **linked** actually outperform **cps** by a solid margin.

5.1.3 Continuation Benchmarks.

Figure 6 shows the relative speed-ups of benchmarks that make heavy use of continuations. We did not evaluate MLton for these benchmarks, because it does not implement `callec` and its `callcc` is too slow. For the three CML benchmarks, SML/NJ performs significantly better than any of our strategies. We believe that most of this performance difference can be attributed to the overhead of promotion required to maintain the DLG heap invariant. Although we are only using a single VPROC for these experiments, we are paying for the mechanisms needed to support scalable multicore performance [52].

The **cps** strategy is far and away the best strategy across the entire suite of continuation-heavy benchmarks. The next best strategy is not clear. The **linked** strategy performs well for the CML benchmarks because it has an efficient `newstack` operation, but it is the weakest strategy for the **ec-⋆** group.

The **ec-⋆** benchmarks replace some or all call and return operations with `callec` and `throw`, exposing the overhead of using continuations for non-local control. While based on a tail-recursive benchmark, **ec-evenodd** grows the stack very deeply (nearly 128 MB) 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 owing to the creation of an external pointer to the stack (Section 4.6).

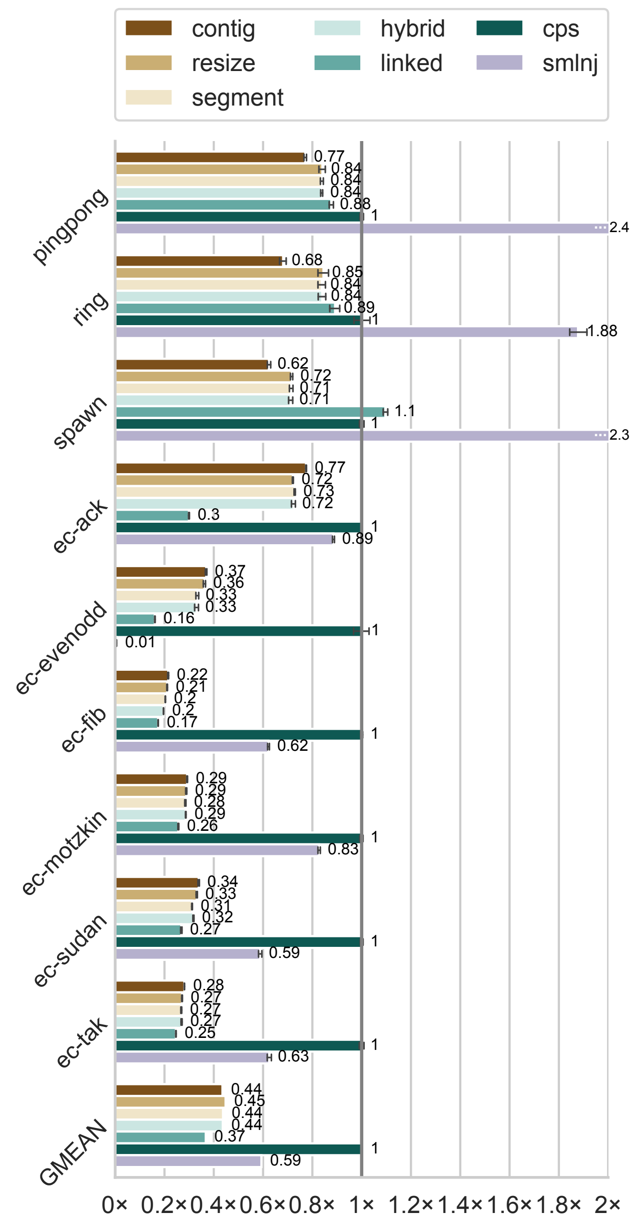

Figure 6. Speed-up comparison over **cps** for the **continuation** benchmarks.

5.2 Design Trade-offs

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

5.2.1 Foreign Function Calls.

By default, the **resize**, **segmented**, and **linked** strategies use a separate stack for foreign-function (FF) calls (*i.e.*, a stack-switching shim), whereas the **contig** and **cps** strategies 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, 2.5× – 2.98×, but for **ffi-trigfib** the speedup was only 1.01× – 1.05×. Thus for non-pathological programs, trading off the overhead for a simpler and more flexible runtime system may be worthwhile.

Table 2. Average L1 data-cache read miss-rates.

	recursive	looping	mixed	continuation
contig	8.61%	5.84%	8.68%	11.02%
resize	7.75%	5.43%	8.44%	9.39%
segmented	7.79%	5.43%	8.44%	9.45%
hybrid	7.83%	5.42%	8.44%	9.48%
linked	15.22%	8.58%	12.02%	14.58%
cps	9.76%	9.06%	11.04%	10.96%

5.2.2 Hardware Stack Support. The x86-64 provides the `call` and `ret` instructions to implement stack-based function calls and returns. To help understand the benefit of using these instructions, 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:

`ret` $\longrightarrow$ `pop %rbx; jmp *%rbx`

This rewrite effectively disables the CPU's return-address stack, which is an internal bookkeeping mechanism to help predict the target of a return [13]. After this transformation the CPU will rely on its indirect-branch predictor, which levels the playing-field with the **cps** strategy. We found that for the **contig**, **resize**, **segmented**, and **hybrid** strategies, using the native CPU instructions yields a 1.02× − 1.07× speedup for the recursive group on average, with **fib** and **motzkin** as consistent outliers seeing a 1.15× − 1.3× speedup. Those strategies see a 1.01× − 1.02× speedup for the mixed benchmarks, and a 1.02× speedup overall across recursive, looping, and mixed groups. Linked stacks saw effectively no change in performance across the board 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.

5.2.3 Cache Locality. One of the primary folklore arguments against the **cps** strategy is that it has poor cache performance because its frames are not reused. To test this, we use the Linux **perf-stat** tool to access the CPU's performance monitoring units to obtain L1 data-cache miss-rates from five complete-process executions for all benchmarks and stack strategies.

The results are in Table 2. This data suggests that while the **linked** and **cps** implementations have higher miss rates, the differences are fairly small in most cases. We believe that other factors also contribute to the performance differences, such as more frequent garbage collections and more copying of live data into return continuations.

5.2.4 Tools. In order for developer tools such as debuggers and profilers to be useful, they must be able to unwind (or walk) the call stack frame-by-frame to understand the calling context. It is a large effort to write such tools for any language implementation, so compatibility with existing tools like **gdb**, **lldb**, and **perf** may be helpful to language implementors and users.

We consider a stack strategy to be highly tool-compatible if existing tools can unwind the stack solely via frame-pointer unwinding, which is the quickest way to walk the stack.

Both the **contig**, **resize**, and **linked** strategies easily support frame-pointer unwinding, with **linked** stacks providing full backtraces in Manticore using **lldb** without any additional effort.[7] Our **segmented** and **hybrid** strategies would require some modification to allow the underflow frames to be parsed. Frame unwinding is not meaningful for the **cps** strategy and thus it is not compatible with existing tools.

5.3 Developer's Manifesto

We found 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 is subtle (Section 4.6). 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 4.6).

From the perspective of the compiler, implementing the direct-style conversion and using LLVM's GC infrastructure were the only major difficulties of getting the stack strategies to work, but 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 and tuned to deal with non-moving large-objects (*i.e.*, stack regions) for the **contig**, **resize**, **segmented**, and **hybrid** strategies. 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 mark-sweep collector then faced performance problems with contiguous stacks because the bookkeeping data for each stack was initially part

[7] Our frame pointers may not be monotonic, so **gdb** does not work.

Table 3. A comparison of the implementation strategies, where ✓ means *good*, ✗ means *bad*, and ● is neutral. The first two rows compare implementation complexity, the next three compare performance, and the remainder compare feature support.

Feature	**contig**	**resize**	**segment**	**hybrid**	**linked**	**cps**
Compiler complexity	✓	●	●	●	✗	✓
Runtime complexity	✗	✗	✗	✗	✗	✓
Fast on recursion benchmarks	✓	✓	✓	✓	✗	✗
Fast on looping benchmarks	✓	✓	✓	✓	✓	●
Fast on mixed benchmarks	✓	✓	✓	✓	●	●
Fast on continuation benchmarks	✗	✗	✗	✗	●	✓
Space efficient threads	✗	✓	●	✓	✓	✓
Support for escape continuations	✓	✗	●	●	✓	✓
Support for 1st-class continuations	✗	✗	●	●	✓	✓
Foreign functions	✓	●	●	●	✗	✓
Supports deep recursion	✗	✓	✓	✓	✓	✓
Tool-compatible	✓	✓	●	●	✓	✗

of the stack region itself. Separately `malloc`-ing the bookkeeping data yielded major cache locality benefits during the sweep; **cml-spawn** saw a 5× overall speed-up with this change.

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. 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 the **cps** strategy is a 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 the **contig** (or possibly **resize**) strategy is 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 fine-grain concurrency and/or parallelism, or other advanced control-flow mechanisms. For such languages, 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; Table 3 summarizes the trade-offs between the six implementation strategies evaluated in this paper.

First, if the ease of implementation in the compiler and runtime system are of utmost priority, **cps** is the simplest implementation. It is also well suited to implement concurrency features without any of the limitations of escape continuations. Its downside is poorer sequential performance[8] and the fact that it is not compatible with existing tools that expect a traditional call stack.

While the **resize**, **segmented**, and **hybrid** strategies share much of the same implementation 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. If we were going to reimplement the Manticore system from scratch, we would give serious consideration to the **hybrid** strategy.

Finally, we conclude that the **linked** strategy 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 **cps**.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grants CCF-1010568 and CCF-1718540. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.

[8]A substantial part of this cost may be regained by more sophisticated compilation techniques [9, 57], 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). Association for Computing Machinery, New York, NY, USA, 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 (Feb. 1989), 171–183. https://doi.org/10.1002/spe.4380190206

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

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

[7] A. W. Appel and T. Jim. 1989. Continuation-passing, Closure-passing Style. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages (POPL '89)* (Austin, TX, USA). Association for Computing Machinery, 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, USA, 1–26. https://doi.org/10.1007/3-540-54444-5_83

[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. https://doi.org/10.1017/S095679680000157X

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

[12] Sven Auhagen, Lars Bergstrom, Matthew Fluet, and John Reppy. 2011. Garbage Collection for Multicore NUMA Machines. In *MSPC 2011: Memory Systems Performance and Correctness* (San José, California, USA). Association for Computing Machinery, New York, NY, USA, 51–57. https://doi.org/10.1145/1988915.1988929

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

[14] Lars Bergstrom, Matthew Fluet, Mike Rainey, John Reppy, Stephen Rosen, Nora Sandler, and Adam Shaw. 2017. *The Manticore Project*. University of Chicago. Retrieved April 4, 2020 from https://manticore.cs.uchicago.edu

[15] 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)* (Philadelphia, PA, USA). Association for Computing Machinery, New York, NY, USA, 99–107. https://doi.org/10.1145/249069.231395

[16] Perry Cheng, Robert Harper, and Peter Lee. 1998. Generational Stack Collection and Profile-driven Pretenuring. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '98)* (Montreal, Quebec, Canada). Association for Computing Machinery, New York, NY, USA, 162–173. https://doi.org/10.1145/277652.277718

[17] Will Clinger, Anne Hartheimer, and Eric Ost. 1988. Implementation Strategies for Continuations. In *Conference record of the 1988 ACM Conference on Lisp and Functional Programming* (Snowbird, Utah, USA). Association for Computing Machinery, New York, NY, USA, 124–131. https://doi.org/10.1145/62678.62692

[18] William D. Clinger and Lars T. Hansen. 2017. *The Larceny Project*. Retrieved April 4, 2020 from http://www.larcenists.org

[19] 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

[20] Olivier Danvy and Julia L. Lawall. 1992. Back to Direct Style II: First-class Continuations. In *Conference record of the 1992 ACM Conference on Lisp and Functional Programming* (San Francisco, CA, USA). Association for Computing Machinery, New York, NY, USA, 299–310. https://doi.org/10.1145/141471.141564

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

[22] 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)* (San Francisco, CA, USA). Association for Computing Machinery, New York, NY, USA, 273–282. https://doi.org/10.1145/143095.143140

[23] 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, OR, USA). Association for Computing Machinery, New York, NY, USA, 70–83. https://doi.org/10.1145/174675.174673

[24] 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, SC, USA). Association for Computing Machinery, New York, NY, USA, 113–123. https://doi.org/10.1145/158511.158611

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

[26] R. Kent Dybvig and Robert Hieb. 1989. Engines from continuations. *Computer Languages* 14, 2 (1989), 109–123. https://doi.org/10.1016/0096-0551(89)90018-0

[27] Kavon Farvardin. 2017. *Weighing Continuations for Concurrency*. Master's thesis. Department of Computer Science, University of Chicago, Chicago IL.

[28] Kavon Farvardin and John Reppy. 2018. Compiling with Continuations and LLVM. In Proceedings *2016 ML Family Workshop / OCaml Users and Developers workshops* (Nara, Japan) *(Electronic Proceedings in Theoretical Computer Science)*, Kenichi Asai and Mark Shinwell (Eds.), Vol. 285. Open Publishing Association, Waterloo, NSW, Australia, 131–142. https://doi.org/10.4204/EPTCS.285.5

[29] Kathleen Fisher and John Reppy. 2002. *Compiler support for lightweight concurrency*. Technical Memorandum. Bell Labs. http://moby.cs.uchicago.edu/papers/2002/tm-lightweight-concur.pdf

[30] Matthew Fluet, Nic Ford, Mike Rainey, John Reppy, Adam Shaw, and Yingqi Xiao. 2007. Status Report: The Manticore Project. In *Proceedings of the 2007 ACM SIGPLAN Workshop on ML*. Association for Computing Machinery, New York, NY, USA, 15–24. https://doi.org/10.1145/1292535.1292539

[31] Matthew Fluet, Mike Rainey, and John Reppy. 2008. A Scheduling Framework for General-purpose Parallel Languages. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming ICFP '08* (Victoria, BC, Canada). Association for Computing Machinery, New York, NY, USA, 241–252. https://doi.org/10.1145/1411203.1411239

[32] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. 2011. Implicitly-threaded Parallelism in Manticore. *Journal of Functional Programming* 20, 5–6 (2011), 537–576. https://doi.org/10.1017/S0956796810000201

[33] Lal George, Florent Guillame, and John H. Reppy. 1994. A Portable and Optimizing Back End for the SML/NJ Compiler. In *Proceedings of the 5th International Conference on Compiler Construction (CC '94)*. Springer-Verlag, New York, NY, USA, 83–97. https://doi.org/10.1007/3-540-57877-3_6

[34] GHC. 2020. *The Glasgow Haskell Compiler*. Retrieved April 4, 2020 from http://www.haskell.org/ghc

[35] 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.

[36] Marcelo J. R. Gonçalves and Andrew W. Appel. 1995. Cache Performance of Fast-allocating Programs. In *Functional Programming Languages and Computer Architecture (FPCA '95)* (La Jolla, CA, USA). Association for Computing Machinery, New York, NY, USA, 293–305. https://doi.org/10.1145/224164.224219

[37] 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* (Austin, TX, USA). Association for Computing Machinery, New York, NY, USA, 293–298.

[38] Matthew Hertz and Emery D. Berger. 2005. Quantifying the Performance of Garbage Collection vs. Explicit Memory Management. In *Proceedings of the 2005 ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05)* (San Diego, CA, USA). Association for Computing Machinery, New York, NY, USA, 313–326. https://doi.org/10.1145/1094811.1094836

[39] 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)*. Association for Computing Machinery, New York, NY, USA, 66–77. https://doi.org/10.1145/93548.93554

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

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

[42] Allen Leung and Lal George. 1999. Static Single Assignment Form for Machine Code. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '99)* (Atlanta, GA, USA). Association for Computing Machinery, New York, NY, USA, 204–214. https://doi.org/10.1145/301618.301667

[43] 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). Association for Computing Machinery, New York, NY, USA, 107–118. https://doi.org/10.1145/1291201.1291217

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

[45] 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

[46] MLton. 2020. *The MLton Standard ML compiler*. Retrieved April 4, 2020 from http://mlton.org

[47] Mikael Pettersson, Konstantinos Sagonas, and Erik Johansson. 2002. The HiPE/x86 Erlang Compiler: System Description and Performance Evaluation. In *International Symposium on Functional and Logic Programming (FLOPS '02)* (Aizu, Japan) *(Lecture Notes in Computer Science)*. Springer-Verlag, New York, NY, USA, 228–244. https://doi.org/10.1007/3-540-45788-7_14

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

[49] Norman Ramsey and Simon Peyton Jones. 2000. Featherweight concurrency in a portable assembly language. (Nov. 2000). https://www.cs.tufts.edu/~nr/pubs/c--con.pdf

[50] Keith Randall. 2013. *Contiguous Stacks in Go*. `golang.org`. Retrieved April 4, 2020 from http://golang.org/s/contigstacks

[51] John Reppy. 2002. Optimizing nested loops using local CPS conversion. *Higher-order and Symbolic Computation* 15 (2002), 161–180. https://doi.org/10.1023/A:1020839128338

[52] 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). Association for Computing Machinery, New York, NY, USA, 257–268. https://doi.org/10.1145/1596550.1596588

[53] John H. Reppy. 1991. CML: A higher-order concurrent language. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '91)* (Toronto, Ontario, Canada). Association for Computing Machinery, New York, NY, USA, 293–305. https://doi.org/10.1145/113446.113470

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

[55] John C. Reynolds. 1993. The discoveries of continuations. *Lisp and Symbolic Computation* 6, 3 (1993), 233–247. https://doi.org/10.1007/BF01019459

[56] 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

[57] 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.

[58] K. C. Sivaramakrishnan, Lukasz Ziarek, and Suresh Jagannathan. 2014. MultiMLton: A multicore-aware runtime for Standard ML. *Journal of Functional Programming* 24 (Nov. 2014), 613–674. Issue 6. https://doi.org/10.1017/S0956796814000161

[59] Darko Stefanovic and J. Eliot B. Moss. 1994. Characterization of Object Behaviour in Standard ML of New Jersey. In *Conference record of the 1994 ACM Conference on Lisp and Functional Programming* (Orlando, Florida, USA). Association for Computing Machinery, New York, NY, USA, 43–54. https://doi.org/10.1145/182409.182428

[60] 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

[61] Mitchell Wand. 1980. Continuation-based multiprocessing. In *Conference Record of the 1980 ACM Conference on Lisp and Functional Programming* (Stanford University, CA, USA). Association for Computing Machinery, New York, NY, USA, 19–28. https://doi.org/10.1145/800087.802786

[62] Andy Wingo. 2014. *Stack Overflow*. Retrieved April 4, 2020 from https://wingolog.org/archives/2014/03/17/stack-overflow