Threaded-Execution and CPS Provide Smooth Switching Among Execution Modes

Dave Mason

Toronto Metropolitan University, Toronto, Canada

Abstract

In executing programs, there is a tension among the need to execute quickly, to not take excessive space, and to be able to debug. This paper discusses using a combination of Continuation-Passing-Style for native code in combinattion with a Threaded-Execution model to address this tension and provide the best of all worlds. Programs can execute at full native speed and then drop instantly into a fully-debuggable execution. Threaded code can be the first level of compilation and then can be easily translated into CPS-style native code that runs approxiamately 4 times as fast. The execution models can be interleaved seamlessly even within a method.

Keywords

Continuation Passing Style, Threaded Execution, Debuggging

1. Introduction

One of the intrinsic tensions in computer science is the space-time tradeoff. Nowhere is this more obvious than in the compilation process. Many of the most important optimizations in the compiler-writer's toolkit exhibit this tradeoff and many heuristics have been developed to address this.

Beyond optimizations, code representation and execution models both exhibit this. Bytecode interpreters lie near one end of the spectrum - very compact representation, but slow execution. Native code lies at the otther end of the spectrum - very fast execution, but significantly larger. Ideally we would like to have small code where it's not performance critical, and fast code where it is.

The rest of this paper is structured as follows: §2 describes Continuation Passing Style and Threaded Execution; §3 talks about some of the key implementation parameters that enable seamless transition among the execution models; §4 presents some preliminary validation of the principles; Finally, §5 provides some concluding thouughts and plans for future work.

IWST 2023: International Workshop on Smalltalk Technologies. Lyon, France; August 29th-31st, 2023

dmason@torontomu.ca (D. Mason)

https://sarg.torontomu.ca/dmason/ (D. Mason)

2. Execution Models

2.1. Continuation Passing Style

Continuation Passing Style (CPS)[1, 2, 3] is a style of programming where, rather than calling other functions/methods with an implicit return address, the continuation (the return and the rest of the computation) is passed explicitly. In the original papers, which were describing functional programming languages, the continuation was passed as a closure. In our case the continuation is passed as a stack pointer and a Context.

In CPS, the control flow is explicit in the form of jumps. Thus all returns are in the form of tail calls that pass control to the next function without pushing any parameters or return addresses onto a stack. Return is explicit by tail-calling a continuation.

In our system, CPS means that a call to a new method involves saving the address of the next code in the Context, and then tail-calling the new method. Returning is simply tail-calling the saved address. In assembler/machine code the address of the next code is directly accessible. In a higher-level language such as our implementation language, Zig[4, 5], the address must be of a function, so functions must be split at call points. If a call point is within a loop, the loop now would span multiple functions, so the loop head is another point where functions must be split. Figure 1 shows a simple function in Zig written in normal style. Figure 2 shows the same

```
fn foo() usize {
  var n = @as(usize,10);
  var tot = @as(usize,0);
  while (n>0) {
    tot = tot + bar(n);
    n = n - 1;
  }
  return tot;
}
fn bar(v: usize) usize {
  return v+1;
}
```

Figure 1: Normal-style function

function in CPS. This is simplified from the CPS we actually use, for expository purposes. For the same reasons, we are not using the actual Zig syntax for the tail-calls, but rather assume all tailcalls are recognized and removed. Here foo is split at the top of the loop because the loop contains a call. foo1 contains the loop test followed either by saving the return address of foo2 and calling bar, or by passing the result back to the calling Context. foo2 has the tail of the loop, and then goes back to the top of the loop. Since on most architectures there are more efficient ways to access values on the stack than generally, there may be a small cost for using the CPS, but if all the methods on Context are inlined (which they can be in Zig), the cost will be minimal.

```
fn foo(caller: Context) void {
  const newContext = caller.push();
  newContext.save(0,10); // n
  newContext.save(1,0);  // tot
  return foo1(newContext);
}
fn foo1(context: Context) void {
  if (context.get(0)>0) {
    context.setReturn(foo2);
    return bar(context,context.get(0));
  }
  const returnCtxt = context.pop();
  return returnCtxt.getReturn()(returnCtxt, context.get(1));
}
fn foo2(context: Context, result: usize) void {
  context.save(1,context.get(1)+result); // update tot
  context.save(0,context.get(0)-1);      // update n
  return foo1(context);
}
fn bar(caller: Context, v: usize) void {
  caller.setResult(v+1);
  return caller.getReturn()(returnContext);
}
```

Figure 2: Continuation-Passing-Style function

2.2. Threaded Execution

Threaded Execution is a form of execution where rather than a sequence of calls to other functions, a function is a sequence of addresses of functions. The hardware stack is unchanged by the sequence of functions (although internally any of them may do so). Figure 3 gives an example of a normal function that calls a sequence of functions. For each call, the language would push the parameters (just ptr in this case) and the return addresses onto the hardware stack.

Figure 4 shows the same sequence in threaded mode. Note that since each threaded function passes control to the next, there is no activity on the hardware stack. Since the pc and ptr parameters are likely to be passed in registers, there is no overhead of memory traffic apart from fetching the next function address, and the only other overhead is advancing pc to point to the next threaded function. However that is additional overhead, as there is a level of indirection not found in normal execution, and the advancing of the program counter is something that is automatically done by the hardware for normal program execution.

The first example of threaded program execution known to the author, was the FORTRAN compiler for the PDP-11 [6]. In order to get the compiler available as quickly as possible after the introduction of the machine, the manufacturer generated threaded code where some of the threaded words were boilerplate and others did simple combinations of operations. This was certainly not an optimizing compiler, but it performed quite well, partially because the PDP-11

```
fn foo(_ptr: [*]data)  [*]data {
    var ptr = _ptr;
    ptr = foo1(ptr);
    ptr = foo2(ptr);
    ptr = foo3(ptr);
    return ptr;
}
fn foo1(ptr: [*]data) [*]data {
    // do something using the data at ptr,
    // possibly modifying to newPtr
    return newPtr;
}
```

Figure 3: Normal Execution

```
const foo = [_]ThreadedFn {&foo1,&foo2,&foo3};
fn executeFoo(ptr: [*]data) void {
     return foo[0].*(&foo[1],ptr);
}
fn foo1(pc: [*]ThreadedFn, ptr: [*]data) void {
    // do something using the data at ptr,
    // possibly modifying to newPtr
    return pc[0].*(pc+1,newPtr);
}
```

Figure 4: Threaded Execution

had an addressing mode (`jmp @(r5)+`) that made the transfer instruction at the end of a word be a single instruction.

The same instruction made the first FORTH [7, 8] implementation particularly performant, and made writing one's own implementation of FORTH a fun weekend project.

[9] describes 3 kinds of interpreters including byte-coded, direct threaded (what we describe in this work), and indirect threaded. While direct threaded produces the best results, they characterize indirect threaded as more flexible. We attain that flexibility with other mechanisms that are beyond the scope of this paper.

Threaded code has been used in Smalltalk compilers [10, 11] as well as OCaml. In both of these systems, the native byte-code has been translated to threaded code to good effect.

The SableVM compiler converts Java byte codes to a threaded execution model [12].

[13] describes using selective inlining to make direct threaded code close to native performance.

[14] describes a related technique they call indirect-threaded code.

3. Implementation

3.1. Memory Model

This system is designed to support multiple Smalltalk processes executing simultaneously.[1] Each process has a private stack and a private nursery heap. In addition there is a shared global heap. Most objects are allocated on a heap, with the exception of three kinds of objects described below.

The nursery heap is actually two arenas, and a copying collector copies live objects from the stack and one arena to the other arena. If an object has been copied back and forth a certain number of times it will be promoted to the global heap. Any pointers in an object cannot point to a younger heap (including the stack) so if any object is promoted, all of the objects it points to must also be promoted.

The global heap is a non-moving arena which uses a mark-and-sweep garbage collector. The details of the global heap are beyond the scope of this paper, but it is worth pointing out that the roots of the global collector primarily come from the collection of stacks from all processes (and transitively the nurseries).

As was observed by Deutsch and Schiffman [15], while Context objects (Smalltalk activation records) semantically are heap allocated, in practice they are almost always used in a strictly last-in-first-out pattern and can therefore profitably be allocated on a stack. As long as we can reliably recognize escaping values and promote them to the heap and thus mantain the required semantics, it is a performance win. The same logic applies to BlockClosure objects which may be created as part of a Context.

Values can escape in one of two ways:

1. a reference to an object is assigned to a heap object, but as mentioned above we already must maintain the invariant that an object cannot point to a younger heap, so the pointed-to object must be promoted to the relevant heap;
2. a reference to an object is returned from the Context in which it is defined, in which case we must promote the object to the nursery heap.

Promotion of BlockClosure or Context objects may force the promotion of ClosureData objects that they reference. Promotion of a Context object may force the promotion of other Context objects that it references. The word 'may' is because, while the object **must** come to reside in the appropriate heap, the object may have already been promoted to a heap (but possibly not the correct heap, in which case it will need to be promoted again).

[1] The Smalltalk proccesses are executed with operating system threads, but we will use 'process' to avoid confusion with 'thread'ed execution.

3.2. Structure of Context

3.3. Switching between Threaded and CPS

3.4. Starting Debug Mode

4. Validation & Results

for 40 runs - x86-64 fibNative: 0.167s fibObject: 0.422s fibComp: 0.586s fibThread: 1.917s

5. Conclusion & Future Work

Summary

Future Work As mentioned in §3.1, Context, BlockClosure, and ClosureData objects are initially allocated on the stack. This could also apply to small, known-size objects, and experiments should be run to see if this is advantageous.

References

[1] A. W. Appel, T. Jim, Continuation-passing, closure-passing style, in: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '89, Association for Computing Machinery, New York, NY, USA, 1989, p. 293–302. URL: https://doi.org/10.1145/75277.75303. doi:10.1145/75277.75303.

[2] A. W. Appel, Compiling with Continuations, Cambridge University Press, 1992.

[3] C. Flanagan, A. Sabry, B. F. Duba, M. Felleisen, The essence of compiling with continuations, in: Conference on Programming Language Design and Implementation 1993, volume 28, Association for Computing Machinery, Albuquerque, NM, USA, 1993, pp. 237–247.

[4] Z. Foundation, Zig is a general-purpose programming language and toolchain for maintaining robust, optimal, and reusable software, ???? URL: https://ziglang.org.

[5] A. Kelly, Zig, 2022. URL: https://ziglang.org/.

[6] J. Bell, Threaded code, Communications of the ACM 16 (1973) 370–372.

[7] C. H. Moore, Forth: a new way to program a mini computer, Astronomy and Astrophysics Supplement 15 (1974) 497–511.

[8] E. D. Rather, D. R. Colburn, C. H. Moore, The evolution of forth, in: History of Programming Languages II, 1993, pp. 177–199.

[9] P. Klint, Interpretation techniques, Software: Practice and Experience 11 (1981). URL: https://doi.org/10.1002/spe.4380110908. doi:10.1002/spe.4380110908.

[10] E. Miranda, Brouhaha- a portable smalltalk interpreter, SIGPLAN Not. 22 (1987) 354–365. URL: https://doi.org/10.1145/38807.38839. doi:10.1145/38807.38839.

[11] E. Miranda, Portable fast direct threaded code, 1991. URL: https://compilers.iecc.com/comparch/article/91-03-121.

[12] E. M. Gagnon, L. J. Hendren, SableVM: A research framework for the efficient execution of java bytecode, in: Java (TM) Virtual Machine Research and Technology Symposium (JVM

01), USENIX Association, Monterey, CA, 2001. URL: https://www.usenix.org/conference/jvm-01/sablevm-research-framework-efficient-execution-java-bytecode.

[13] I. Piumarta, F. Riccardi, Optimizing direct threaded code by selective inlining, SIGPLAN Not. 33 (1998) 291–300. URL: https://doi.org/10.1145/277652.277743. doi:10.1145/277652.277743.

[14] R. Dewar, Indirect threaded code, Communications of the ACM 18 (1975) 330–331.

[15] L. P. Deutsch, A. M. Schiffman, Efficient implementation of the smalltalk-80 system, in: Proceedings of the 11th Annual ACM Symposium on Principles of Programming Languages, POPL '84, Association for Computing Machinery, 1984, p. 297–302. URL: https://doi.org/10.1145/800017.800542. doi:10.1145/800017.800542.