Threaded-Execution and CPS Provide Smooth Switching Between 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 combination 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 approximately 4 times as fast. The execution models can be interleaved seamlessly even within a method.

Keywords

Continuation Passing Style, Threaded Execution, Debugging

1. Introduction

This paper explores different execution models for Smalltalk[1].

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.[2]

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 other 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: Section 2 describes Continuation Passing Style and Threaded Execution; Section 3 talks about some of the key implementation parameters that enable seamless transition between the execution models; Section 4 presents some preliminary validation of the principles; references to related work are throught the paper; finally, Section 5 provides some concluding thoughts 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

There are a range of execution models that can be used to implement a programming language. The most common are native code and bytecode interpreters.

An intermediate option between those extremes is threaded execution. The basic idea is that programs are constructed by stringing together "words", where each word is the address of some code that performs its operation and then passing control to the next "word" without using the traditional procedure calls. This makes threaded code as easy to generate as bytecode. While it is not quite as compact as bytecode, it runs significantly faster than byte-code interpreters. Rather than traditional procedure calls/returns these threads pass along the current continuation (stack and context in our case), and execute the next "word" via a **tailcall**.

If we want to smoothly inter-operate native code with threaded code, we have to write the native code so it it uses the same parameters. The compilation technique that aligns with those parameters is continuation passing style.

2.1. Continuation Passing Style

Continuation Passing Style (CPS)[4, 5, 6] 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 (the return address in the Context).

In our system, CPS means that a call to a new method involves saving the address of the next code - that is the next word in the threaded representation, and the next native function - 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[7, 8], 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 function in CPS. The context.save and context.get are storing and retrieving the variables n and sum at their temp locations (0 for n and 1 for sum) in the context. 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 use **tailcall**. 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 at arbitrary locations in memory, 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() u64 {
  var n:u64 = 10;
  var sum: u64 = 0;
  while (n>0) {
    sum=sum+bar(n);
    n=n-1;
  }
  return sum;
}
fn bar(v: u64) u64 {
  return v+1;
}
```

Figure 1: Normal-style function

```
const n = 0;     // position in context
const sum = 1; // position in context
fn foo(caller: Context) void {
  const newContext = caller.push();
  newContext.save(n,10);
  newContext.save(sum,0);
  tailcall foo1(newContext);
}
fn foo1(context: Context) void {
  if (context.get(n)>0) {
    context.setReturn(foo2);
    tailcall bar(context,context.get(n));
  }
  const returnC = context.pop();
  tailcall returnC.getReturn()(returnC,
     context.get(sum));
}
fn foo2(context: Context, result: u64)
    void {
  context.save(sum,context.get(sum)+result
     );
  context.save(n,context.get(n)-1);
  tailcall foo1(context);
}
fn bar(caller: Context, v: u64) void {
  tailcall caller.getReturn()(caller,v+1);
}
```

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, as long as there i no net change). 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.[1]

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

[1]Note that the [*] Zig syntax means a pointer to multiple values.

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

Figure 4: Threaded Execution

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 [9]. 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 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 [10, 11] implementation particularly performant, and made writing one's own implementation of FORTH a fun weekend project.

[12] 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 [13, 14] 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 [15].

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

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

3. Implementation

To enable seamless transition between CPS and threaded execution we have to make some particular implementation decisions.

1. the stack cannot reasonably be woven into the hardware stack with function calls;
2. contexts have to contain not only native return points, but also threaded return points;
3. CompiledMethods have to facilitate seamless switching between execution modes.

But first let's look at some background on the memory model.

3.1. Memory Model

The Smalltalk system we are building is designed to support multiple Smalltalk processes executing simultaneously on multiple cores.[2] Each process has a private stack. In addition there is a heap, details of which are outside the scope of this paper. Most objects are allocated on the heap, with the exception of three kinds of objects described below.

As was observed by Deutsch and Schiffman [18], 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 maintain the required semantics, it is a performance win. Objects that are allocated on the stack (in a Context) include the Context itself, referenced as `thisContext`, BlockClosure objects which may be created as part of a Context, or ClosureData objects referred to by the Context and some BlockClosure.

Values can escape in one of two ways:

1. a reference to an object is assigned to a heap object (since we must maintain the invariant that an object cannot point to a younger area of storage, the pointed-to object must be promoted to the 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 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 heap, the object may have already been promoted to the heap.

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

3.2. Interface Points and Data Structures

There are 3 data structures that are the connection points for moving between the execution models: the Stack, the Method, and the Context. Figure 5 shows the relationship among the

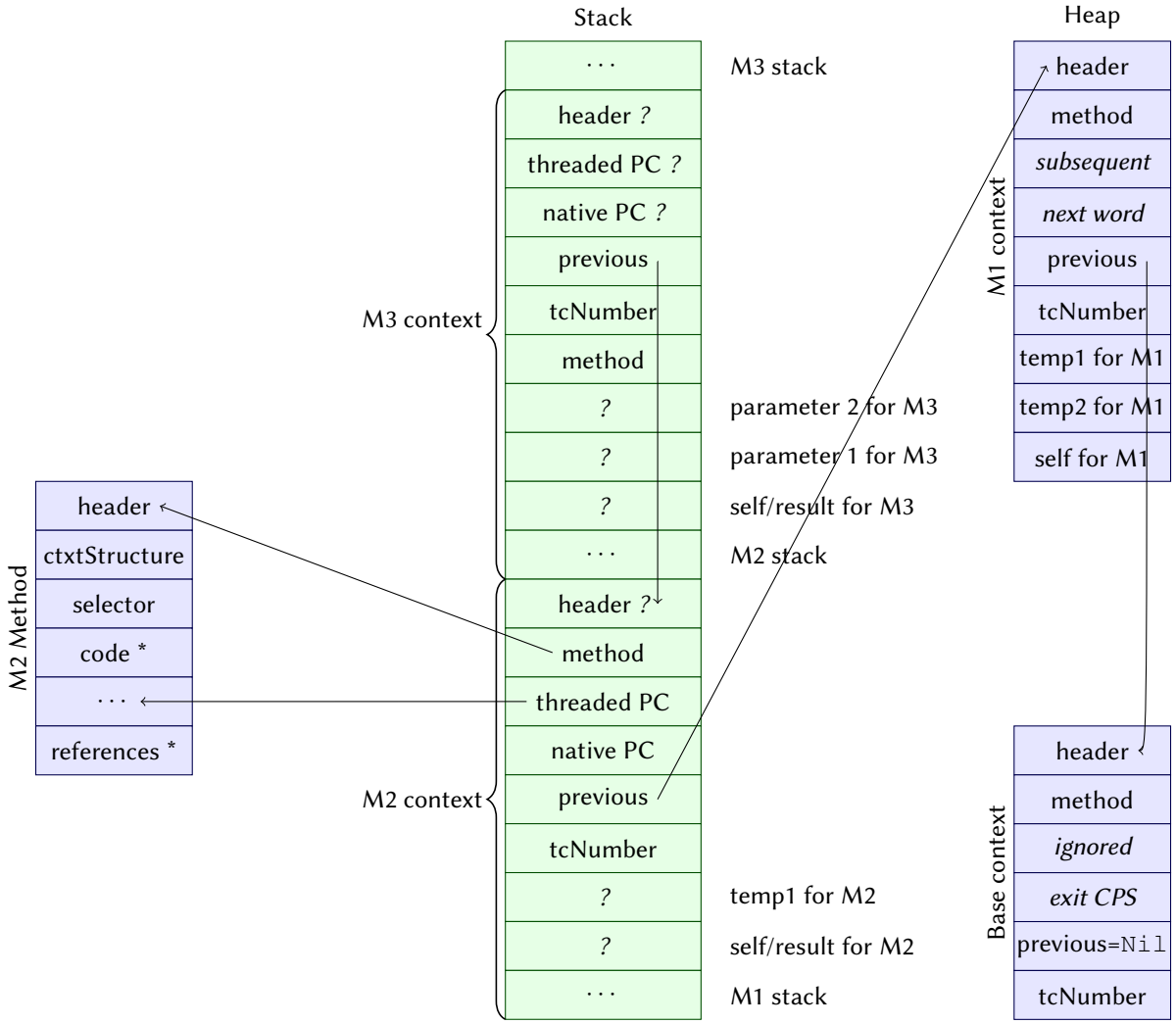

Figure 5: Stack/Heap structure of contexts

stack, heap, Contexts, and Methods. This represents the state of execution where a method, M1, has called M2, which has called M3 which is executing, but hasn't called any other method.

3.2.1. Stack

The first decision is to not use the hardware stack, but instead use a stack allocated per process, with overflow to the heap. In addition to supporting this multiple execution model, by putting

all roots in this stack there is no confusion or complexity of tagged values, tagged pointers, and native pointers; everything on the stack is tagged. This simplifies the garbage collector significantly.

3.2.2. Method

Every CompiledMethod regardless of execution model has a code area (at a fixed offset in the CompiledMethod object). Execution of a CompiledMethod begins with a threaded call to the first word. The interpretation of the remainder of the code area is completely defined by this function.

1. For a pure native code method, this will be an array of pointers to the CPS chunks.
2. For a pure threaded method, this will be the threaded code addresses, preceded by a selectorVerify function - which might be replaced at some point with a pointer to native code.
3. For native code backed by threaded code, this would be as in the previous example, except the first word points to the native code.
4. For interpreted code, such as the proof of concept mentioned in Section 4, the first word points to the interpreter, and the remainder of the area would contain the byte-codes. This could even support multiple interpreters for different byte-codes.

3.2.3. Contexts

A `Context` is the representation of a method/function activation record or stack frame. These are initially allocated on the top of the stack, but may migrate to the heap when the stack gets too large or the `Context` escapes, as explained in Section 3.1. When they are allocated on the stack, they are only partially populated, because they will likely be discarded before they need to be treated as proper objects. Before a `Context` migrates to the heap, the remaining fields are filled in and it can then be treated as a first-class object and can be used to implement Scheme's `call-with-current-continuatiion`, light-weight threads, or other control flows as well as return.

Method pointer points to the method/function for the context. This is set up when the context is created on the stack.

threaded PC is the address of the next pointer in the threaded version of the method. This is only filled in when a method/function is called.

native PC is the address of the next CPS part of the native implementation. This is only filled in when a method/function is called. For a native CPS method, this will be the next CPS chunk address If there is no native implementation of this method, this will be the next threaded word in the method. If this method were interpreted, this would be the address of the interpreter. Therefore return from a call always just invokes the native PC, passing the threaded PC.

Previous context points to the context that invoked the current method/function. This is set up when the context is created on the stack.

tcNumber is used to handle non-local returns efficiently in the face of handling exceptions. This is set up when the context is created on the stack.

temps are the temporary values for the method. They are initialized to `Nil` when the context is created on the stack.

BlockClosure and ClosureData references Any BlockClosure or ClosureData is allocated above the Context, with references here. They are accessed as temp indices.

parameters are the parameters being passed to the method. This is part of the caller's stack. These, and everything above are discarded when the method returns. They are accessed as temp indices.

result is the location of the result of the current function, and also is the `self` value in an object-oriented language or the first parameter in non-OO languages. As part of the caller's stack, it will be left on the stack when the method returns. It is accessed as the last temp index.

stack is the rest of the caller's stack. This is different from a context in traditional Smalltalk VMs, where the stack for the method is part of the method's context. The reason for the change is that it eliminates the need to move the stack values around in the normal context-on-stack case. On return from a Context that resides in the heap, the result value and stack will need to be copied to the stack, becoming the complete stack, as all of the Contexts implicitly reside in the heap.

If a Context is on the stack, then when it is returned from, all of the context before the result is discarded by simply adjusting the stack pointer. This makes the round-trip cost of creating and deleting a Context on the order of 20 instructions - only a small factor worse than a native function call.

If a Context is on the heap, then when it is returned from, the stack portion of the context is copied to the stack area. Combined with the creation and migration of the `Context` to the heap this is about 3-4 times the round-trip cost of the simpler on-stack cost.

Our contexts are similar to [19] except that they have a native and a threaded return address, and that the stack and parameters are from the sender in a more natural way.

3.3. Primitives

Smalltalk is conceptually a very simple language. The only operations are message sends, assignment of simple variables, and return. The connection to the underlying hardware all happens through primitives. Figure 6 shows the addition operation for SmallInteger. The annotation `<primitive: 1>` says that this method will attempt to evaluate by invoking primitive 1, which is implemented in the runtime/virtual-machine. If the primitive is successful,

```
    <primitive: 1>
    ^ super + aNumber
```

Figure 6: The canonical Smalltalk method `SmallInteger #+`

the parameters will be consumed and the result returned. If the primitive fails, then the following Smalltalk code will be executed with the parameters unchanged. This gets translated into the Zig code in Figure 7. Here the `&primitive.p1` refers to primitive 1. Normally a Context

```
&primitive.p1,
&embedded.superTailSend,sym.@"+",
```

Figure 7: The threaded code for `SmallInteger #+` (Figure 6)

would be created if the primitive failed. However in this case, since there are no local variables, and there is only one message send in tail-call position we avoid creating a Context.

The Zig code for the primitive is shown in Figure 8. It first checks that the message sent was

```
pub fn p1(pc: [*]const Code, sp: [*]Object, process: *Process,
    context: ContextPtr, selector: Object) void {// SmallInteger
    >>#+
    if (!sym.@"+".equals(selector))
        tailcall dnu(pc,sp,process,context,selector);
    sp[1] = inlined.p1(sp[1],sp[0]) catch
        tailcall pc[0].prim(pc+1,sp,process,context,selector);
    tailcall context.npc,(context.tpc,sp+1,process,context,
        selector);
}
```

Figure 8: Implementation of primitive 1

+[3]. Then it calls the code that does the tagged addition, passing it `self` and `aNumber`. If it fails (`aNumber` wasn't a SmallInteger, or the sum doesn't fit a SmallInteger), we fall through to the rest of the method, executed in threaded mode. The assumption is that failure is rare, so threaded code is adequate. If the addition succeeded, we return to the method that sent the + message in the first place.

For completeness, Figure 9 shows the inline code that does the sum operation. We assume that `self` is a SmallInteger, and if the `other` value is a SmallInteger, and the result of the summation is a SmallInteger, then we return that new object. Otherwise we return an error.

One of the key compilation/performance approaches for this system is aggressive inlining of Smalltalk code. The details are outside the scope of this paper, but if a method is inlined

[3]The reason we check is related to how dispatch is handled, and is outside the scope of this paper.

```
pub inline fn p1(self: Object, other: Object) !Object { // Add
    if (other.isInt()) {
        const result =
              @bitCast(Object,self.i()+%other.toUnchecked(i64))
                 ;
        if (result.isInt()) return result;
    }
    return error.primitiveError;
}
```

Figure 9: Inlined code that implements primitive 1

that starts with a primitive, we need to embed that primitive call in the method that sends the message. This looks like `&embedded.p1` and Figure 10 shows the code that is referenced in this case. For a method to invoke an embedded primitive that could fail, a Context must have

```
pub fn @"+"(pc: [*]const Code, sp: [*]Object, process: *Process
   , context: ContextPtr, selector: Object) [*]Object {
    sp[1] = inlines.p1(sp[1], sp[0]) catch
        tailcall fallback( pc + 1, sp, process, context, Sym.@"
           +" );
    tailcall pc[0].prim( pc + 1, sp + 1, process, context,
       selector );
}
```

Figure 10: Embedded version of primitive 1

already been created. Because this was not a message send, we don't have to check that the selector is correct. If the inline summation fails, we invoke a fallback - sending the + method to the object. Otherwise, we pass control to the next threaded word, having updated the stack.

3.4. Switching between Threaded and CPS

As mentioned in Section 3.2.2 when execution of a method commences, the first threaded word of the code area is invoked. This means that execution will proceed in the manner defined by that method, so threaded code can call native code and native code can call threaded code with no conditional code required.

Similarly, as mentioned in Section 3.2.3 when a return is made to a Context, the native PC field is invoked, passing the threaded PC field. Hence any kind of executing mode can return to any other with no conditional code required.

In fact, moving in both directions could easily support other execution models such as interpreters.

Starting Debug Mode The first three modes of program execution described in Section 3.2.2 seamlessly support interruption and single-stepping of code. Support for interpreter execution modes would need to be implemented slightly differently, e.g. with recognition that the return was an interpreter and setting the return to a debugging entry point to the interpreter.. The first word for each method, along with checking for the correct selector, checks to see if interruption is required. This interruption could be debugging, interaction with the global garbage collector, a user or other process requesting interruption, etc. Performance would be negatively affected if this checking is performed too frequently, so it currently is done at the entry point for a method, and at the top of loops.

While executing within native code, if an interruption is required, execution is switched to threaded execution, which can easily be single-stepped. The native PC field in a Context can similarly be set to the threaded function equivalent, which is accessible from the threaded PC field.

4. Validation & Results

This is still a work in progress, but here we will describe a hand-compiled implementation of fibonacci.

4.1. Versions of fibonacci code

```
1 fibonacci
2     self <= 2 ifTrue: [ ^ 1 ].
3     ^ (self - 1) fibonacci + (self - 2) fibonacci
```

Figure 11: Smalltalk implementation of fibonacci

Figure 11 shows a simple Smalltalk version.

Figure 12 shows the threaded implementation.[4] All of the benchmarks use direct early-binding of the recursive calls to make them as comparable as possible (dynamic dispatch is covered in another paper). This is similar to the IR code for the `Integer>>#fibonacci` function in Pharo. p5 is <= which leaves a boolean on top of the stack. If true, then lines 6-8 replace self with 1, and then return. Line 10 creates a Context (which we hadn't needed because no error was possible and we hadn't called anything). Then we subtract the literal 1 from self and call recursively in lines 11-13, then the same -2, and finally at lines 17-18 we add the values and return.

Figure 13 shows the first part of the CPS native code paralleling the threaded code. This covers lines 1-13 of Figure 12, Lines 3-6 check for the base case and return 1. Lines 7-8 create the Context (the 0, 2, 0 parameters say 0 locals, 2 max needed stack, and **self** is temp/local 0, and fibThread refers to the code in Figure 12). Line 9 tries to subtract 1. If it fails, line 10 switches

[4]This is somewhat de-optimized from the actual benchmark for explanatory purposes.

```
1         &embedded.verifySelector,
2         ":recurse",
3         &embedded.dup,              // self
4         &embedded.pushLiteral, Object.from(2),
5         &embedded.p5,               // <=
6         &embedded.ifFalse,"label3",
7         &embedded.drop,             // self
8         &embedded.pushLiteral1,
9         &embedded.returnNoContext,
10        ":label3",
11        &embedded.pushContext,"^",
12        &embedded.pushLocal0,       // self
13        &embedded.pushLiteral1,
14        &embedded.p2,               // -
15        &embedded.callRecursive, "recurse",
16        &embedded.pushLocal0,       //self
17        &embedded.pushLiteral2,
18        &embedded.p2,               // -
19        &embedded.callRecursive, "recurse",
20        &embedded.p1,               // +
21        &embedded.returnTop,
```

Figure 12: Threaded implementation of fibonacci

to the threaded code to try the operation, because we obviously have underflow.[5] Line 11-13 sets up for, and makes the recursive call. Line 11 is pointing to line 14 of Figure 12, and line 12 points to the actual code.

Figure 14 shows the second part of the CPS code, corresponding to lines 14-16 of Figure 12.

Finally, Figure 15 shows the end of the CPS code, including possible overflow. Line 8 calls back the native PC from the saved Context.

[5]Can't actually underflow here, but this is the correct code.

```
pub fn fibCPS(pc: [*]const Code, sp: [*]Object, process: *
    Process, context: ContextPtr, selector: Object) void {
    if (!fibSym.equals(selector)) tailCall dnu(pc,sp,process,
        context,selector);
    if (inlined.p5N(sp[0],Object.from(2))) {
        sp[0] = Object.from(1);
        tailcall context.npc(context.tpc,sp,process,context,
            selector);
    }
    const newContext = context.push(sp,process,fibThread.
        asCompiledMethodPtr(),0,2,0);
    const newSp = newContext.asObjectPtr()-1;
    newSp[0] = inlined.p2L(sp[0],1) catch tailcall pc[10].prim(
        pc+11,newSp+1,process,context,fibSym);
    newContext.setReturnBoth(fibCPS1, pc + 13); // after first
        callRecursive (line 15 above)
    tailcall fibCPS(fibCPST+1,newSp,process,newContext,fibSym);
}
```

Figure 13: CPS implementation of fibonacci - first function

```
fn fibCPS1(pc: [*]const Code, sp: [*]Object, process: *Process,
    context: ContextPtr, _: Object) void {
    const newSp = sp-1;
    newSp[0] = inlined.p2L(context.getTemp(0),2) catch tailcall
        pc[0].prim(pc+1,newSp,process,context,fibSym);
    context.setReturnBoth(fibCPS2, pc + 3); // after 2nd
        callRecursive (line 19 above)
    tailcall fibCPS(fibCPST+1,newSp,process,context,fibSym);
}
```

Figure 14: CPS implementation of fibonacci - second function

```
fn fibCPS2(pc: [*]const Code, sp: [*]Object, process: *Process,
    context: ContextPtr, selector: Object) void {
    const sum = inlined.p1(sp[1],sp[0]) catch tailcall pc[0].
       prim(pc+1,sp,process,context,fibSym);
    const result = context.pop(process);
    const newSp = result.sp;
    newSp[0] = sum;
    const callerContext = result.ctxt;
    tailcall callerContext.npc(callerContext.tpc,newSp,process,
       callerContext,selector);
}
```

Figure 15: CPS implementation of fibonacci - third function

4.2. Timings

Table 1 shows some preliminary timing for this micro-benchmark. The benchmarks were run 5 times each after 2 warmup runs and showed minimal variance (standard deviation less than 1.5%). The table records median timing values.

Table 1
Comparative execution times for `40 fibonacci`

Execution	Apple M1	Apple M2 Pro	2.8GHz Intel i7	Size AArch
Pharo JIT	591ms	559ms	695ms	68b*
Pharo Stack	3527ms	5033ms	3568ms	68b
Native	186ms	134ms	134ms	72b
Object	306ms	278ms	331ms	188b
CPS Object	393ms	348ms	563ms	528b
Threaded	1939ms	1094ms	1994ms	200b
ByteCode	4820ms	4730ms	4609ms	104b

* Note this is is just the bytecode, the JIT'ed code is in addition to this.

Figure 16 is a graph of the same data.

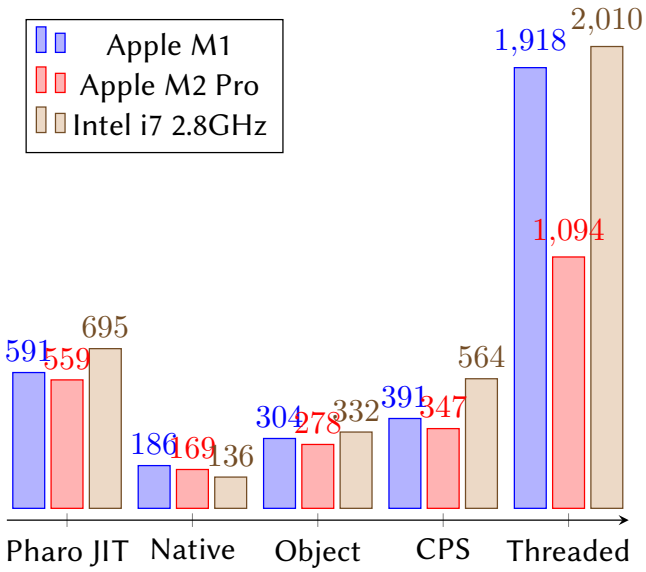

Figure 16: Native Timing in milliseconds

The Pharo data is from running on Pharo 10, on the Pharo version of the OpenSmalltalk VM. Pharo JIT is running on a JIT VM, and Pharo Stack is using the Stack Interpreter VM.

Native is the straightforward implementation in Zig using 64 bit integers. Not surprisingly it is very fast, but interestingly the native optimized code for this micro-benchmarks is also very small.

Object is the straightforward implementation in Zig using tagged Objects, with local Zig variables and recursion. This is as good as we could hope to be, using the well-supported hardware stack and call/return semantics. It also demonstrates that using tagged integers doesn't add *too* much overhead.

CPS is the CPS conversion of this using our Context objects. This doesn't use any of the standard conventions - stack, hardware recursion, static types. The fact that this is only 20% slower, at least on AArch64 is surprisingly good. It's also 10-40% faster than the Pharo JIT'ed code. We don't know the size of the Pharo JIT'ed code, but we suspect that the CPS code is larger.

Threaded is the fully threaded version. This is extremely easy to generate, and is rewarding that it is only 2-3 times slower than the Pharo JIT'ed code. There was almost no optimization available for such a simple method, so we are very optimistic about the potential for our system. The threaded code is about $\frac{2}{5}$ the size of the CPS code and 3-5 times slower, which seems like a good trade-off.

ByteCode is an unoptimized byte-code interpreter. It is about $\frac{1}{2}$ the size of the threaded version and almost 4 times slower. Since it doesn't save much space, and is actually slightly more difficult to generate than the threaded version, it doesn't seem worth pursuing at this time.

As a micro-benchmark of hand-compiled code, this should be taken as simply a validation that this a viable approach to a Smalltalk runtime.

5. Conclusion & Future Work

Summary We have shown how we support two (or more) modes of program execution, particularly the seamless transition between the modes. This supports code in either dense or fast modes. We also can easily enter and exit debugging modes.

The decisions to support the easy simultaneous support for native (CPS) and threaded execution (use of a separate stack, and our design for contexts and methods), seem to have been validated. Now we can work on the optimizations

The current benchmark cannot be taken too seriously, but appears to show, in some cases, that the CPS form is only 20% slower than optimal, and the threaded code is only a factor of 5 slower.

Future Work This describes work in progress. The next stage is to automatically compile code so we can get some real performance numbers.

Then we will be working on the optimizations that are enabled by the decisions that we've discussed here as well as others we're exploring. We are very optimistic.

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

References

[1] A. Goldberg, D. Robson, Smalltalk-80: The Language and its Implementation, Addison-Wesley, Don Mills, Ontario, 1983. URL: https://rmod-files.lille.inria.fr/FreeBooks/BlueBook/Bluebook.pdf.

[2] T. Ball, J. R. Larus, Branch prediction for free, in: [21], 1993, pp. 300–313.

[3] R. E. Johnson, J. O. Graver, L. W. Zurawski, Ts: An optimizing compiler for smalltalk, SIGPLAN Not. 23 (1988) 18–26. URL: https://doi.org/10.1145/62084.62086. doi:10.1145/62084.62086.

[4] 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.

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

[6] C. Flanagan, A. Sabry, B. F. Duba, M. Felleisen, The essence of compiling with continuations, in: [21], 1993, pp. 237–247.

[7] Z. Foundation, Zig is a general-purpose programming language and toolchain for maintaining robust, optimal, and reusable software, 2022. URL: https://ziglang.org.

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

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

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

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

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

[13] 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.

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

[15] 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.

[16] 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.

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

[18] L. P. Deutsch, A. M. Schiffman, Efficient implementation of the smalltalk-80 system, in: [22], 1984, p. 297–302. URL: https://doi.org/10.1145/800017.800542. doi:10.1145/800017.800542.

[19] E. Miranda, Under cover contexts and the big frame-up, 2009. URL: http://www.mirandabanda.org/cogblog/2009/01/14/under-cover-contexts-and-the-big-frame-up/.

[20] N. Suzuki, M. Terada, Creating efficient systems for object-oriented languages, in: [22],

1984, p. 290–296. URL: https://doi.org/10.1145/800017.800541. doi:10.1145/800017.800541.

[21] PLDI, Conference Record of the 1993 ACM SIGPLAN Conference on Programming Language Design and Implementation, volume 28, Association for Computing Machinery, Albuquerque, NM, USA, 1993.

[22] Proceedings of the 11$^{\text{th}}$ Annual ACM Symposium on Principles of Programming Languages, POPL '84, Association for Computing Machinery, 1984.