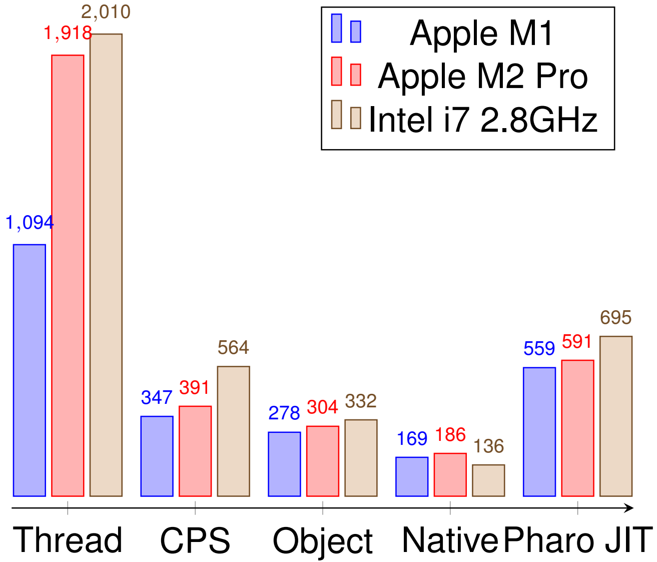
Threaded-Execution and CPS Provide Smooth Switching Between Execution Modes

Dave Mason
Toronto Metropolitan University

Execution Models

- Source Interpretation
- Bytecode Interpretation
- Threaded Execution
- Hardware Interpretation

Timing of Fibonacci



Threaded is 2.3-4.7 times faster than bytecode.

Zag Smalltalk

- supports 2 models: threaded and native CPS
- seamless transition
- threaded is smaller and fully supports step-debugging
- native is 3-5 times faster and can fallback to full debugging after a send

Threaded Execution

- sequence of addresses of native code
- like an extensible bytecode
- each word passes control to the next
- associated with Forth, originally in PDP-11 FORTRAN compiler, was used in BrouHaHa

fibonacci

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

Threaded `fibonacci`

```
            verifySelector,
        ":recurse",
            dup,                 // self
            pushLiteral, Object.from(2),
            p5,                  // <=
            ifFalse,"label3",
            drop,                // self
            pushLiteral1,
            returnNoContext,
        ":label3",
            pushContext,"^",
            pushLocal0,          // self
            pushLiteral1,
            p2,                  // -
            callRecursive, "recurse",
            pushLocal0,          //self
            pushLiteral2,
            p2,                  // -
            callRecursive, "recurse",
            p1,                  // +
            returnTop,
```

Some control words

```
pub fn drop(pc:PC, sp:Stack, process:*Process, context:ContextPtr, selector:Object) Stack {
    tailcall pc[0].prim(pc+1, sp+1, process, context, selector, cache);
}
pub fn dup(pc:PC, sp:Stack, process:*Process, context:ContextPtr, selector:Object) Stack {
    const newSp = sp-1;
    newSp[0] = newSp[1];
    tailcall pc[0].prim(pc+1, newSp, process, context, selector, cache);
}
pub fn ifFalse(pc:PC, sp:Stack, process:*Process, context:ContextPtr, selector:Object) Stack {
    const v = sp[0];
    if (False.equals(v)) tailcall branch(pc, sp+1, process, context, selector, cache );
    if (True.equals(v)) tailcall pc[1].prim(pc+2, sp+1, process, context, selector, cache );
    @panic("non␣boolean");
}
pub fn p1(pc:PC, sp:Stack, process:*Process, context:ContextPtr, selector:Object) Stack {
    if (!Sym.@"+".selectorEquals(selector)) tailcall dnu(pc, sp, process, context, selector);
    sp[1] = inlines.p1(sp[1], sp[0])
        catch tailcall pc[0].prim(pc+1, sp, process, context, selector, cache);
    tailcall context.npc(context.tpc, sp+1, process, context, selector, cache);
}
```

Continuation Passing Style

- continuation is the rest of the program
- comes from optimization of functional languages (continuation was a closure)
- no implicit stack frames - passed explicitly
- like the original Smalltalk passing Context (maybe not obvious that Context is a special kind of closure)

Normal Style

```
pub fn fibNative(self: i64) i64 {
    if (self <= 2) return 1;
    return fibNative(self - 1) + fibNative(self - 2);
}
const one = Object.from(1);
const two = Object.from(2);
pub fn fibObject(self: Object) Object {
    if (i.p5N(self,two)) return one;
    const m1 = i.p2L(self, 1) catch @panic("int␣subtract␣failed␣in␣fibObject");
    const fm1 = fibObject(m1);
    const m2 = i.p2L(self, 2) catch @panic("int␣subtract␣failed␣in␣fibObject");
    const fm2 = fibObject(m2);
    return i.p1(fm1, fm2) catch @panic("int␣add␣failed␣in␣fibObject");
}
```

Continuation Passing Style

```
pub fn fibCPS(pc:PC, sp:Stack, process:*Process, context:ContextPtr, selector:Object) Stack {
   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.sp();
   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);
}
```

Continuation Passing Style

```
fn fibCPS1(pc:PC, sp:Stack, process:*Process, context:ContextPtr, _:Object) Stack {
    const newSp = sp.push();
    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);
}
```

Continuation Passing Style

```
fn fibCPS2(pc:PC, sp:Stack, process:*Process, context:ContextPtr, selector:Object) Stack {
    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.put0(sum);
    const callerContext = result.ctxt;
    tailcall callerContext.npc(callerContext.tpc,newSp,process,callerContext,selector);
}
```

Implementation Decisions

- `Context` must contain not only native return points, but also threaded return points;
- `CompiledMethods` must facilitate seamless switching between execution modes;
- the stack cannot reasonably be woven into the hardware stack with function calls, so no native stack;
- as well as parameters, locals, and working space, stack is used to allocate Context and BlockClosure as usually released in LIFO pattern

Conclusions

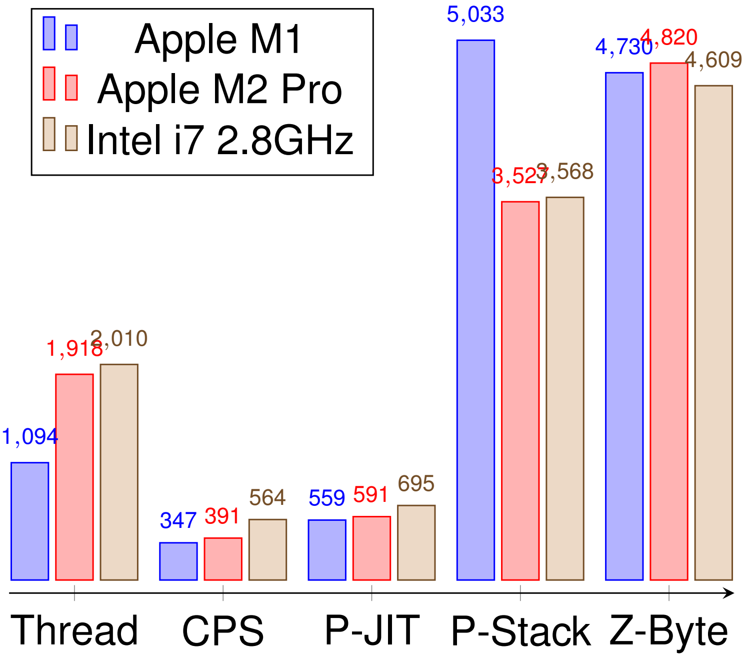
- with proper structures, can easily switch between threaded and native code
- threaded code is “good enough” for many purposes
- this is preliminary work, so some open questions
- many experiments to run to validate my intuitions
- many more details in the paper

Questions?

@DrDaveMason dmason@torontomu.ca

https://github.com/dvmason/Zag-Smalltalk

Timing of Fibonacci



M2 P-Stack is presumed to be mis-configured