Threaded Execution as a Dual to Native Code

Dave Mason
dmason@torontomu.ca
Toronto Metropolitan University
Toronto, Canada

ABSTRACT

Threaded execution has been used as a higher performance alternative to a byte-code interpreter, by utilizing hardware dispatch to replace software interpreter dispatch.

Traditional JIT code is compiled from byte-code to native code for the current machine, with an even higher performance result. Unfortunately, when debugging is required for a method, most JIT-based interpreters discard the JIT code and revert to the byte-code interpreter. Additionally, switching between interpreted code and native code requires clever trampolines to bring the models into alignment.

We describe a technique that maintains the threaded code and native code as parallel implementations of the program. This provides seamless transitions between the implementations and supports full debugging, while providing near full native execution performance.

CCS CONCEPTS

• **Software and its engineering** → Object oriented languages; *Procedures, functions and subroutines*; **Control structures**.

KEYWORDS

execution model, threaded execution

ACM Reference Format:
Dave Mason. 2023. Threaded Execution as a Dual to Native Code. In *Companion Proceedings of the 7th International Conference on the Art, Science, and Engineering of Programming (‹Programming› '23), March 13–17, 2023, Tokyo, Japan.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3594671.3594673

1 INTRODUCTION

We are building a new VM, initially for Smalltalk, using the systems language Zig for the core VM functionality. There are several novel ideas being explored in the VM, but this paper will focus on the execution model that uses a combination of byte-coded, direct-threaded[1] and continuation-passing-style (CPS) native code.

Others have explored threaded code, but as a just-in-time (JIT) target generated from the language byte-code. Our system compiles from source directly to threaded code (effectively treating the threaded code as the byte-code), and then translates the threaded code to CPS native code where appropriate. We can also execute OpenSmalltalk byte codes directly, and all 3 execution models seamlessly inter-operate.

Although our initial target is Smalltalk, and we refer frequently to code as "methods", the VM is targeting multiple dynamic languages and the technique is not dependant on any particular model of dispatch, so everything we are saying applies just as well to functions.

The major contributions of this work are the seamless inter-operation of three different execution models which are described in §2. There are 2 key data structures where the inter-operation occurs, described in section §3. §4 gives performance data for a micro-benchmark. §5 contains a brief discussion of related work. Finally, §6 provides some conclusions.

2 EXECUTION MODEL

The execution model for the VM combines byte-code, threaded, and continuation-passing-style (CPS)[1] native code execution. Within the VM language code, the hardware stack is unused; rather, a language specific stack is used for program execution and temporary storage. This simplifies garbage collection and inter-operation with limited cost to execution performance. The VM also supports multiple hardware execution Threads, but the implementation of those is beyond the scope of this paper, so despite the occasional mention of the `Thread` structure, which contains a stack, we are talking here about the execution of a single Thread.

2.1 Threaded Execution

A threaded implementation of a method is simply a sequence of addresses of functions that take a set of values, perform an operation and pass control to the next function as a thread, rather than as a procedure call/return. Various related work VMs had to do workarounds such as compiling to assembly code and then editting that code. Fortunately, Zig[7] supports tail-calls directly which resolves these problems and makes a threaded execution model very easy to write.

Figure 1 shows the code for a very simple example, a threaded instruction to push a literal value from the instruction stream onto the stack.

pc is a pointer into the list of threaded instructions. It always points to the word after this instruction. In this case, that word is a literal object, so the next instruction will be offset by 1, which is done in the tail-call.

[1]The term "thread" in this paper is unfortunately ambiguous. As described in §2.1 threaded execution strings together a sequence of code blocks without standard procedure call/return; rather than the currently more common use referring to multiple CPU threads of execution.

‹Programming› '23, March 13–17, 2023, Tokyo, Japan

ACM ISBN 979-8-4007-0755-1/23/03…$15.00
https://doi.org/10.1145/3594671.3594673

```
pub fn pushLiteral(pc: [*]const Code, sp: [*]Object, hp: Hp, thread: *Thread,
                                    context: ContextPtr, selectorHash: u32) void {
    const newSp = sp-1;
    newSp[0]=pc[0].object;
    return @call(tailCall,pc[1].prim,.{pc+2,newSp,hp,thread,context,0});
}
```

Figure 1: Push literal

sp is a pointer to the current top of stack. In this case we are pushing a value onto the stack, so we first have to create room on the stack, and then store at that address.

hp is the pointer to the free area between the heap and stack. This method should check that there is room for the stack to grow, but we have omitted that to simplify the explanation.

thread is the per-hardware-Thread structure that contains the stack, but is otherwise outside the scope of this paper.

context points to the context for the current executing method, or the calling method if no context is created yet.

selectorHash contains the hash for the selector, which is checked at the start of a method to handle mis-hashed selectors and undefined methods.

Each threaded instruction has exactly the same 6 parameters, and passes the same values (possibly modified) on to the next instruction. On modern architectures, all parameters will be in registers, so threaded instructions are largely manipulating registers, with some interaction with the stack.

The last instruction is a tail-call to the next threaded instruction. The tail call skips one word (the literal) and passes along an updated *pc* that points after the next instruction. It also passes along the new *sp* value because that's now the top of the stack. The remaining values are passed unchanged.

The threaded tail call becomes at the machine level a simple load followed by a branch-register instruction.

2.2 Native CPS Execution

When native code is generated, each method is broken into a series of CPS chunks (with boundaries at method calls and some loop headers). The chunks communicate via the stack and heap, but within a chunk many optimizations can occur.

Each CPS chunk takes exactly the same parameters and passes control to the next CPS function with the same parameters. Both the native code and the threaded instruction functions may call other functions, using the normal stack, but when passing control to the next CPS chunk or passing control to the next threaded-function, it does a tail-call with the parameters from the same function. In other words, at the start of every CPS function or threaded instruction, the hardware stack will be the same. All call/return of the VM language is done through the use of `Contexts` (see §3.3).

The CPS chunks pass the correct *pc* as if the CPS chunk were a threaded instruction. This means that at any point execution can switch to threaded mode.

2.3 Byte Code Execution

The byte-code interpreter takes exactly the same parameters as the threaded and native execution so that it can be called anywhere. It also maintains the Context consistently so that any method-call/message-send can be to any of the three kinds of execution.

3 INTERFACE POINTS AND DATA STRUCTURES

There are 3 data structures that are the connection points for moving among the execution models.

3.1 Stack

The first decision is to not use the hardware stack, but instead use a stack allocated in the nursery, 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. This simplifies the garbage collector significantly.

3.2 Method

Every Method regardless of execution model has a code area (at a common offset in the Method object). For a pure native code method, this will be an array of pointers to the CPS chunks. 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. For native code backed by threaded code, this would be as in the previous example, except the first word points to the native code. For a byte-coded method the first word is a pointer to the interpreter, followed by the byte sequences.[2]

3.3 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` is explicitly referenced (via the pseudo-variable `thisContext` in Smalltalk). When they are allocated on the stack, they are only partially populated (only 2 of the 6 header words are filled in, as well as all local values being set to `Nil`), 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

[2]Note that this would easily support multiple interpreters for different byte-codes.

implement Scheme's `call-with-current-continuatiion`, light-weight threads, or other control flows as well as return.

header is required because a Context is an object and is defined by the memory management system, so will not be further described here. This is only partially filled in when a context is created on the stack.

threaded PC is the address of the next word 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 is interpreted, this will be the address of the interpreter. Therefore return from a call always just invokes the native PC passing the threaded PC..

Size is the number of words remaining in the `Context`. This is only filled in upon migration to the heap. In any normal context (except the base context) size will be at least 2, for the pointer to the previous context, the pointer to the method, and in an object-oriented language, the `self` value (which will make the size at least 3).

address points to the following word (for GC purposes). This is only filled in upon migration to the heap.

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

Method pointer points to the method/function for the context. 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.

*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.

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.

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, the non-stack portion of the context is simply discarded by 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.

3.4 Switching between Execution Models

As can be seen, the execution model is completely compatible among the byte-coded, threaded and CPS code. With all active program data residing in the stack and heap, the mode of execution can be switched with very fine granularity.

The CPS code is aware of what threaded code it replaces, and this is stored in the context. This means that when a method is waiting for the return of another method, the context contains the address of the CPS function to continue with in the `nPc` field, as well as the address of the threaded code that corresponds in the `tPc` field. Simply by replacing the `nPc` with the address of the next threaded block, the method can revert to threaded execution upon return. CPS methods typically represent many threaded blocks, broken up only on method calls that can't be inlined, and on loops that contain such calls. This means that significant optimizations can be applied within each CPS block.

Going the other way, compiling from threaded to CPS can be done in the background by another thread. When the compilation is complete, the first word (a no-op) can be replaced with the address of the first CPS function, and on the next call the method will run in CPS mode. Note that this completely obviates the need for self-modifying-code mechanisms beyond a simple multi-core-aware store of this address.[6]

Every threaded instruction checks to see if it is supposed to be single-stepping, so a thread can get interrupted very quickly. Every CPS block also checks at the beginning of the block. If it sees that a more interactive form of execution is required, it can jump to the next threaded block and resume threaded execution immediately.

Another use of this is to handle low-probability code paths. Rather than inline all error handling the CPS code can hand off execution to the threaded code. This can have a significant space saving, with very little impact on performance. An example of such a low-frequency path is for primitives such as integer addition, where adding 2 small integers is handled by the primitive, but if the result would overflow or one of the values is another kind of number, then a variety of options is tried. This can easily and efficiently be shunted off to threaded code, because at the next message send or return native execution will resume seamlessly.

Switching to and from byte-code does not support as fine granularity, but an interpreted method can be called from any of the 3 execution models, and can be returned to directly. This supports method call granularity for this switch.

4 EXAMPLE AND PERFORMANCE

Because this work is currently in development, we are unable to automatically generate code for either implementation. Therefore these section describes work with hand-compiled and optimized code.

4.1 Fibonacci

(1) Figure 2 shows the original Smalltalk implementation for Pharo. We created 6 implementations of the Fibonacci number calculation (some omitted for space):
(2) a straight-forward Native implementation in Zig;

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

Figure 2: Smalltalk version of Fibonacci

```
    verifySelctor,
"fibonacci:",
    dup,
    pushLiteral, 2,
    lessOrEqual,
    ifFalse,"label3",
    pop,
    pushLiteral, 1,
    returnDirect,
"label3:",
    pushContext,"^",
    pushTemp1,
    pushLiteral, 1,
    sub,
    call,"fibonacci",
    pushTemp1,
    pushLiteral, 2,
    sub,
    call,"fibonacci",
    add,
    returnTop,
```

Figure 3: Direct-threaded version of Fibonacci

(3) a similar implementation, with tagged Object values using primitives(the best that we could reasonably expect to match, since we need to maintain our dynamically typed objects);
(4) a CPS-converted version translated from the threaded implementation to Zig shown in figure 4.
(5) our direct-threaded implementation, with instructions similar to those in many stack-based VMs, such as the Java VM or the standard Smalltalk VM shown in figure 3.
(6) a byte-coded version - similar to the threaded version, except operations are more restricted (there can only be 256) and references to object values require an extra level of indirection.

4.2 Performance

Recognize that this is a micro-benchmark and does not necessarily translate directly to other, larger programs. Table 1 shows the raw execution times for this micro-benchmark.

Pharo is a high-quality implementation of Smalltalk, and for this particular benchmark is able to achieve 40% of the speed of the native Zig implementation. The CPS version is about 11% faster than the Pharo/OpenSmalltalk version, and the threaded version is about 3.5 times slower.

From very preliminary evidence, even on 64 bit architectures, the threaded code is very compact: aproximately 20% to 25% of the size of the native (CPS) code.

Table 1: Comparative execution times for `40 fibonacci`

Execution	Apple M1	2.8GHz Intel i7
Pharo	591ms	695ms
Native	213ms	165ms
Object	243ms	352ms
CPS Object	545ms	596ms
Threaded	2185ms	2336ms
ByteCoded	5129ms	4948ms

5 RELATED WORK

The first example of threaded program execution known to the author, was the FORTRAN compiler for the PDP-11 [3]. 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 [11, 13] implementation particularly performant, and made writing one's own implementation of FORTH a fun weekend project.

[8] 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 [9, 10] 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 [5].

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

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

[2] describes speculation and deoptimization of JIT code.

6 CONCLUSIONS

Byte-coded, threaded and CPS code all have important advantages. Byte-code is extremely compact - useful for very low-execution methods. Threaded code is easier to generate than native code and can reasonably be single-stepped (or more complex operations) for debugging purposes. CPS code can be optimized to run at near native execution speeds.

By keeping the three models compatible, we can switch back and forth among them with very little complexity or slowdown. This gives a near-perfect blend of the advantages of each.

```
fn fibcomp(pc: [*]const Code, sp: [*]Object, hp: HeapPtr, thread: *Thread,
           context: ContextPtr, selectorHash: u32) void {
    if (selectorHash!=) {
    if (i.lessOrEqual(sp[0],Object.from(2))) {
        sp[0] = Object.from(1);
        return @call(tailCall,context.npc,.{context.tpc,sp,hp,thread,context});
    }
    const result = context.push(pc,sp,1);
    const newContext = result.ctxt;
    const newHp = result.hp;
    const newSp = newContext.asObjectPtr()-1;
    if (i.subtract(sp[0],Object.from(1))) |m1| {
        newSp[0] = m1;
        newContext.tpc = pc+4;
        newContext.npc = fibComp1;
        const fib = fibCompT;
        return @call(tailCall,fib[0].prim,.{fib+1,newSp,newHp,thread,newContext});
    }
}
fn fibComp1(pc: [*]const Code, sp: [*]Object, hp: HeapPtr, thread: *Thread, context: ContextPtr) void {
    const newSp = sp-1;
    if (i.subtract(context.getTemp(0),Object.from(2))) |m2| {
        newSp[0] = m2;
        context.tpc = pc+3;
        context.npc = fibComp2;
        const fib = fibCompT;
        return @call(tailCall,fib[0].prim,.{fib+1,newSp,hp,thread,context});
    }
}
fn fibComp2(_: [*]const Code, sp: [*]Object, hp: HeapPtr, thread: *Thread, context: ContextPtr) void {
    if (i.add(sp[1],sp[0])) |sum| {
        context.setTemp(0,sum);
        const result = context.pop(sp,thread,0);
        const newSp = result.sp;
        const caller = result.context;
        return @call(tailCall,caller.npc,.{callerContext.tpc,newSp,hp,thread,caller});
    }
}
```

Figure 4: CPS-converted version of Fibonacci

REFERENCES

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

[2] Aurèle Barrière, Sandrine Blazy, Olivier Flückiger, David Pichardie, and Jan Vitek. 2021. Formally Verified Speculation and Deoptimization in a JIT Compiler. *Proc. ACM Program. Lang.* 5, POPL, Article 46 (Jan. 2021), 26 pages. https://doi.org/10.1145/3434327

[3] James Bell. 1973. Threaded code. *Commun. ACM* 16, 6 (1973), 370–372.

[4] Robert Dewar. 1975. Indirect threaded code. *Commun. ACM* 18, 6 (1975), 330–331.

[5] Etienne M. Gagnon and Laurie J. Hendren. 2001. 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. https://www.usenix.org/conference/jvm-01/sablevm-research-framework-efficient-execution-java-bytecode

[6] Tim Hartley, Foivos S. Zakkak, Andy Nisbet, Christos Kotselidis, and Mikel Luján. 2022. Just-In-Time Compilation on ARM—A Closer Look at Call-Site Code Consistency. *ACM Trans. Archit. Code Optim.* 19, 4, Article 54 (Sept. 2022), 23 pages. https://doi.org/10.1145/3546568

[7] Andrew Kelly. 2022. *Zig*. Retrieved 2022-11-10 from https://ziglang.org/

[8] Paul Klint. 1981. Interpretation Techniques. *Software: Practice and Experience* 11 (1981). https://doi.org/10.1002/spe.4380110908

[9] Eliot Miranda. 1987. BrouHaHa- A Portable Smalltalk Interpreter. *SIGPLAN Not.* 22, 12 (Dec. 1987), 354–365. https://doi.org/10.1145/38807.38839

[10] Elliot Miranda. 1991. *Portable Fast Direct Threaded Code*. Retrieved 2022-11-10 from https://compilers.iecc.com/comparch/article/91-03-121

[11] Charles H. Moore. 1974. FORTH: a new Way to Program a Mini Computer. *Astronomy and Astrophysics Supplement* 15 (1974), 497–511.

[12] Ian Piumarta and Fabio Riccardi. 1998. Optimizing Direct Threaded Code by Selective Inlining. *SIGPLAN Not.* 33, 5 (May 1998), 291–300. https://doi.org/10.1145/277652.277743

[13] Elizabeth D. Rather, Donald R. Colburn, and Charles H. Moore. 1993. The evolution of Forth. In *History of Programming Languages II*. 177–199.

Received 2023-01-22; accepted 2023-02-06