Alternative dispatch techniques for the Tcl VM Interpreter

Benjamin Vitale and Mathew Zaleski
University of Toronto, Systems Research Lab
$\{$*bv,matz*$\}$*@cs.toronto.edu*

Abstract

We compare the performance of various virtual machine dispatch strategies in Tcl, including traditional highly-portable techniques, and newer techniques which sacrifice some portability for performance.

Tcl's high-level opcodes have large implementation bodies and contain C function calls. Compared to other VMs, the opcodes require many cycles to execute. Dispatch overhead is relatively low, because large bodies consume much more execution time than dispatch. Direct threaded code improves tclbench benchmarks by about 5% over switch dispatch. We review our catenation technique, which compiles bytecode using copied templates of Sparc code made from the normal C-compiled VM's implementation of each virtual opcode. This eliminates all dispatch, but is impractical due to poor portability, and because the copying amplifies Tcl's heavy instruction cache load.

Based on subroutine threading, context threading generates native call instructions for dispatch. Simpler than catenation, it imposes much lower I-cache load. It preserves more interpreter state, is a better vehicle for mixed-mode execution, and accomodates interesting optimizations. Our implementation for Tcl on Sparc improves 97% of benchmarks in the tclbench suite with more than 1000 dispatches, by an average of 9.5% over switch dispatch (12.0% and 16.5% for $>$ 10000 and $>$ 100000 dispatches, respectively.)

Keywords: Threaded Interpreter Dispatch, Virtual Machines, Tcl Performance, Context Threading, Subroutine Threading

1 Introduction

Like other scripting languages, Tcl is interpreted, yielding rich dynamic semantics but slow performance relative to native code. Lewis's bytecode compiler and virtual machine [11], introduced in Tcl version 8.0, provided enough performance for most users, but some still demand more speed. There are two categories of approaches to improving VM interpreter performance. First, just-in-time compilers (JITs) can translate bytecode to native code, but tend to be large, complex, and less portable than an interpreter. Furthermore, the Tcl VM is implemented with high-level bytecodes which are an excellent match for Tcl's semantics, but not the best starting point for a JIT, as we'll discuss briefly in Section 6.

A lighter weight approach is to apply clever interpretation techniques. These include stack caching, register machines, superinstructions, or the approach we take in this paper, faster dispatch. Every VM must dispatch virtual instructions, and Tcl uses a portable but slow technique: a C switch statement. Some other strategies are listed in Table 1, together with their execution times on an UltraSPARC III CPU.

Replacing switch with subroutine threading can save $17 - 6 = 11$ cycles per dispatch. The impact on overall performance depends on the number of cycles required to execute the real (non-dispatch) work of the average opcode. As shown in Figure 1, Tcl opcodes have long execution time. They take much longer than a dispatch takes, and thus dispatch overhead is low. If the average opcode executes in, for example, 100 cycles, the best speedup we can achieve by saving eleven cycles is 11%.

Interpreters for some other language systems, such as Ocaml, have quick low-level opcodes. There, a 17 cycle switch dispatch is unacceptable overhead. Instead, Ocaml uses direct threaded dispatch. In previous work [3], we

Dispatch type	Cycles
switch	17.3
indirect threading	14.2
direct threading	11.2
subroutine threading	6.3
catenation	0

Table 1. Speed of various dispatch techniques on UltraSPARC III CPU, measured using microbenchmarks, based on Ertl's [5], which dispatch an empty `nop` opcode body.

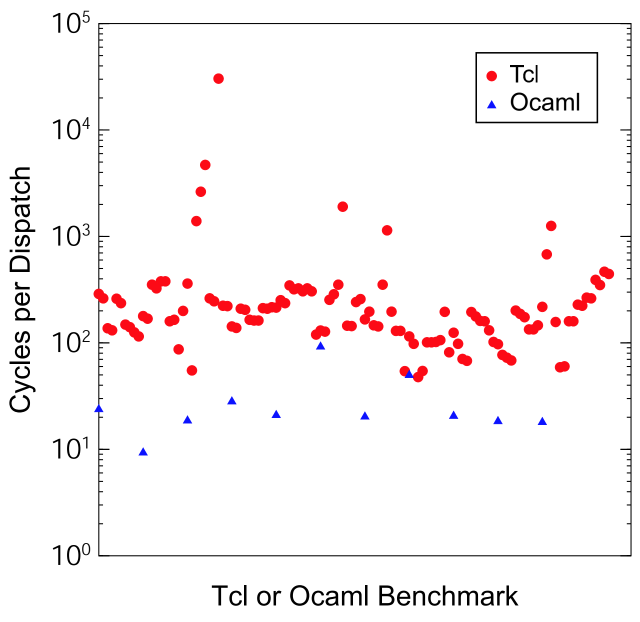


Figure 1. Tcl's mostly high-level opcodes have relatively large implementations, and require many cycles to execute. This chart plots a dot for each benchmark, showing the ratio of total execution cycles divided by total number of dispatches. Ocaml's bytecodes are lower level, thus dispatch optimizations help it more than Tcl.

proposed a technique based on subroutine threading, called *context threading*. It requires non-portable run-time synthesis of native code (`calls` to the opcode bodies), but significantly increased performance for Ocaml and Java VMs. In this paper, we apply context threading to Tcl.

In addition to cheaper dispatch, context threading also offers a flexible foundation for further incremental optimizations. This turns out to be important for Tcl, where the dispatch-only improvements have limited impact for many benchmarks.

We will also summarize another system we built, *catenation* [22], which attempts to eliminate *all* dispatch, but is more complicated and performs worse. We also try direct threading, and find, for Tcl on Sparc, it achieves roughly the same dispatch-oriented benefits as context threading. It is simple, portable, and avoids the pitfalls of our more exotic techniques, such as adding instruction cache load, which we find is already unusually heavy in Tcl. Tcl should favor direct over switch dispatch. However, direct threading cannot offer the same flexible optimization platform as context threading.

The main contributions of this work are:

- An implementation of context threading for Tcl.
- An implementation of context threading on Sparc, showing that it performs well on Ocaml and has some benefit for Tcl. Earlier work targeted only the x86 and PowerPC architectures.
- The observation that context threading, while making parsimonious use of the instruction cache, can still cause miss-related stalls if the cache is already filled by a VM's working set.
- Recommendations for Tcl VM development: use direct threading, and, longer-term, lower-level bytecodes.

2 Background

In this section, we review the structure and performance of some traditional dispatch techniques, and our own less portable but potentially higher performance versions.

2.1 Slow Switch Dispatch

Why does switch dispatch take 17 cycles to execute? The C compiler usually implements a `switch` statement as a bounds check, a table lookup to find the address of the code for the relevant `case` statement (in an interpreter, this chooses which virtual opcode to dispatch), and then an indirect branch. The bounds check wastes several cycles, and the table lookup requires a shift instruction, and possibly an extra load, because the base address of the table is rarely in a register. Finally, each opcode body ends with a `break` statement, which branches back to the dispatch logic, if the `switch` is in a `for` loop.

The indirect branch is a major hazard on modern out-of-order, deeply-pipelined CPUs (the Ultrasparc III has a 14 stage pipeline [20].) To avoid this hazard, most CPUs incorporate hardware to predict, based on the program location of the indirect branch, what its target will be. This is typically a *branch target address cache* (BTAC) or *branch target buffer* (BTB). Early Ultrasparcs associated one or two *next fetch addresses* (NFA) with each instruction cache line, containing another I-cache index, rather than a memory address [23].

Whichever technology is used, Ertl [6] points out that switch dispatch fares poorly, because only a single indirect branch, and thus a single prediction of the next virtual instruction in the program, is available. There are roughly 100 Tcl opcodes, so the predictor is wrong almost every time. We refer to the difficultly in predicting the next opcode as the *context problem* [3].

2.2 Direct Threaded Code

Bytecode represents the virtual program as an array of bytes, containing opcodes and operands. The virtual program counter (vpc) is a pointer into this array. Direct threaded code instead stores an array of machine *words*, and changes the opcodes to the native memory *addresses* of the opcode bodies. See Figure 4c. Dispatch then becomes essentially two machine instructions: load the native address at the vpc, and then indirect jump there. Typically, a separate copy of this dispatch code is appended to the bottom of *every* body. This saves many instructions, and, crucially, makes much better use of the branch target predictors, because now there are 100 predictions, one for the indirect branch at the end of each opcode.

Direct threading is a big improvement over switch dispatch, even for Tcl, where dispatch overhead is low. Unfortunately, even it consumes too many cycles on modern CPUs, because the predictors are still wrong about half the time in many VMs [6]. To see why direct threading still exhibits the context problem, consider the virtual instruction sequence Ipush, Ipush, Iadd. When dispatching from the first to second Ipush, the predictor will learn that Ipush typically follows itself. When dispatching from the second Ipush, it will guess that the next opcode body will again be Ipush, and begin fetching and speculatively executing instructions from that body. Since the actual destination is Iadd's body, the pipeline will have to discard the results, and stall while waiting to fetch the correct instructions.

2.3 Context Threading

To resolve the context problem, we proposed *context threading* [3]. It is based on an old interpreter technique known as *subroutine threading*, which compiles virtual program instructions into a series of native call instructions to the bodies of those virtual instructions. We refer to the sequence of calls as the context threading table, or CTT (see Figure 2). While executing in the CTT, the branch target predictor does very well, because each call goes to the same address every time each it executes.

Basic subroutine threading leaves two main challenges: operands and control flow. The CTT defines only the opcodes used by the instructions in a virtual program, not the operands. We retain the original direct threaded program, and it is an integral part of context threading's program representation. The opcode bodies are essentially identical to the bodies in a traditional switch-based or threaded interpreter, differing only in the final dispatch instruction, now a return, instead an indirect branch. These bodies fetch operands from the bytecode program using the same logic as the switch-based interpreter.

This requires that the virtual program counter be maintained at all times, which, of course, is already done by the traditional bodies. Furthermore, when virtual control flow is encountered, context threading executes corresponding native control flow. To implement this, we slightly alter the direct threaded code (which we call the DTT). Note that the opcode bodies only refer to the operands in the DTT. The opcode-address slots are unused. Given a virtual instruction, we rewrite the corresponding DTT opcode-address slot to instead contain the address of the CTT slot for that instruction. We use this information in the dispatch code after virtual branches. While most bodies end with a return, virtual branches bodies end with traditional direct threaded dispatch – loading from the DTT, offset by the newly adjusted virtual PC, and an indirect jump, to the CTT.

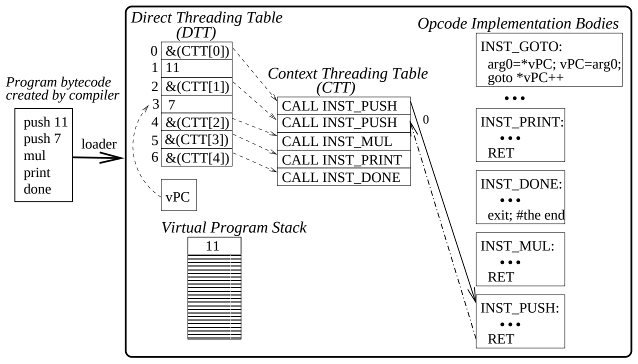


Figure 2. Context Threading

The key performance advantage of subroutine threading is that it usually avoids the costly indirect branch of dispatch. Instead, native calls to immediate addresses are perfectly predicted by the CPU's branch target buffer, because the address of the call and its target are perfectly correlated. The return instruction at the end of each body is perfectly predicted by the hardware *return address stack* (RAS) present in most modern CPUs.

Implementing virtual branches using an indirect branch dispatch exploits neither the BTB nor RAS. Thus the first optimization context threading makes over subroutine threading is to "inline" virtual branch code into the CTT. The actual virtual branch body is called or inlined, and then a native branch emitted inline. This avoids the indirect branch, the body correctly returns to its corresponding call, and the native branch is exclusively associated with the virtual branch, exposing it to the hardware conditional (taken/not-taken) branch predictors.

In addition to reducing dispatch overhead, another key feature of subroutine threading is its flexibility. For example, context threading makes one final optimization, which is to inline the bodies of very simple opcodes directly into the CTT. When applied to Tcl, this flexibility turns out to be even more important to performance than cheap dispatch, which, as discussed in Section 1, is of limited impact for Tcl.

2.4 Catenation

Another approach we tried earlier, *catenation*, eliminates *all* dispatch. We include it in our evaluation for the sake of comparison. The basic idea is to quickly generate native code from a bytecode program using copies of code *templates* derived from a switch-based or threaded interpreter compiled by GNU C. The original bytecode program is not consulted during execution. Much interpreter state remains, so we consider catenation to be an advanced dispatch technique, not a compiler. However, some aspects of interpretation *are* removed. Catenation changes virtual branches to native, specializes bytecode operands into the templates, and mostly eliminates the virtual program counter, rematerializing it only for exception handling. For full details of the implementation, consult the original paper [22]. We'll briefly present some details here.

We started with Tcl 8.4a3 and Miguel Sofer's experimental **s4** [19] performance enhancements. Among other things, these enhancements inline common cases for things like variable handling into opcode bodies, reducing calls to runtime functions.

The implementation is divided into three main parts: creating templates in C, analyzing and extracting the templates from compiled C, and finally using the templates to perform catenation. The last of these steps is shown in Figure 3. To create the templates, we made modifications to the function `TclExecuteByteCode()`, which implements the opcode bodies of the bytecode interpreter. Each template must be a self-contained piece of code. Most of the changes are to reduce the use of shared code between bodies, to remove updates of the virtual program counter, and to replace code which fetches virtual operands with code that instead loads a 'magic number' constant, used for operand specialization. As shown in Figure 3c, we also append to virtual branch bodies an assembly language compare instruction which sets the Sparc condition code register to indicate whether the branch is taken or not.

We surround each opcode body with C labels, and build an array of these labels using `gcc`'s labels-as-values extension, yielding an array of the starting and ending native addresses of each body. Still, when `TclExecuteByteCode()` is compiled by `gcc`, the output is not usable as templates. We post-process the assembly output using Tcl scripts to fix various problems. This is brittle; upgrading to a new `gcc` version can take a day of work, and we have not attempted a port to another architecture. Ertl [10] shows a slightly more portable way to find specialization slots.

When our Tcl interpreter starts up, it caches an analysis of the templates, including their start, length, and virtual and native type of each specialization slot. The location of each native slot is found by looking for three different Sparc instructions with the magic numbers as operands. This information is stored in a set of 'patches', which allow the final implementation stage to proceed quickly.

Each time the Tcl bytecode compiler emits a new bytecode object, it is compiled to native code using the templates. Catenation is a three pass process. The first pass computes the net length of templates for all virtual instructions, and constructs a two-way map from virtual pc location to native address. Next, all templates are copied into place.

The third pass is specialization, which applies patches to each template. This can be seen in Figure 3c, where the operand to `inst_push` is substituted into the Sparc `sethi/or` instructions (the Sparc idiom for setting a 32 bit constant in a register.) Most operands undergo only a sign extension, but in this case we lookup the address of the object to push in the literal table at catenation time, and specialize into the code, saving the lookup at execution time.

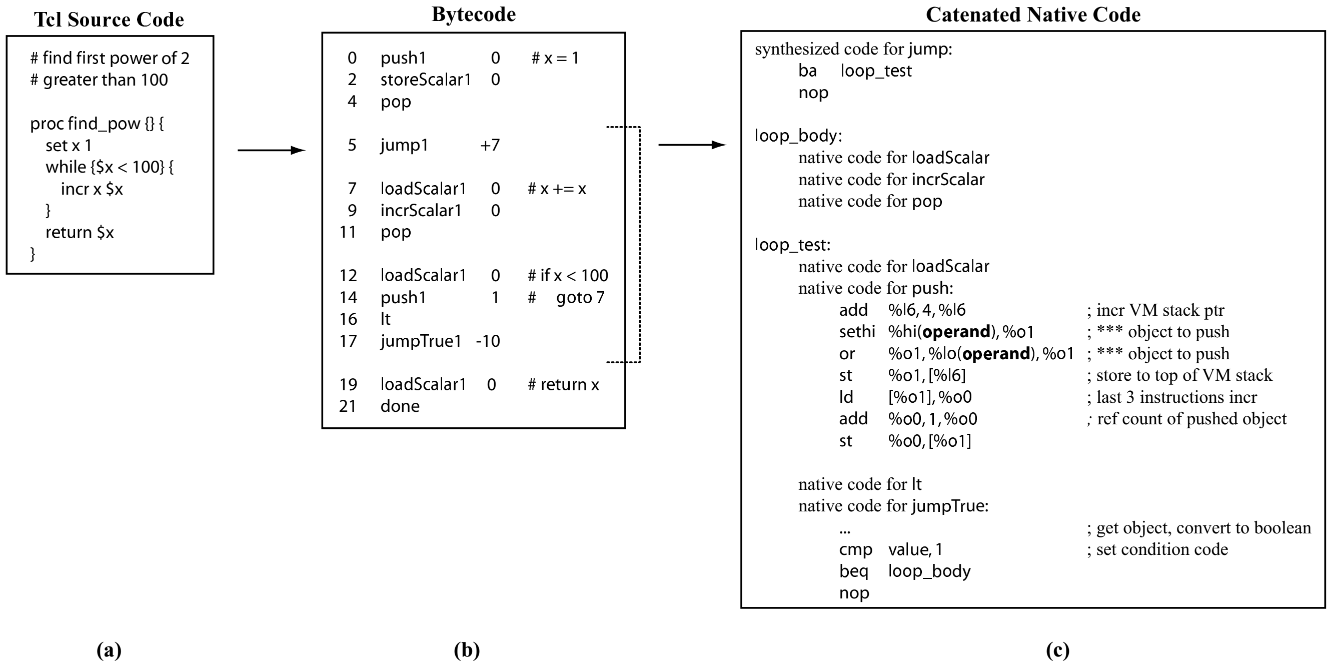
In addition to specializing operands, all pc-relative native instructions in a template whose target is outside the template (e.g. C function calls) must be adjusted based on their new location. Finally, after each virtual branch (`jump_true` or `jump_false`) template is copied, we select and emit the appropriate native branch instruction (`beq` or `bne`, respectively).

In the original Tcl interpreter, exception handling code shared by all opcodes unwinds the virtual stack, and uses a run-time lookup on the existing `vpc` to determine the new `vpc` at which to resume execution. But we haven't maintained `vpc`. Instead, we re-materialize it: when an opcode triggers an exception, it sets the `vpc` to a *constant*, then branches to the common exception code. The constant is known at compile time. For example, in Figure 3b, when compiling `storeScalar` at `vpc` 6 into native code, we can specialize the value 6 into the template.

In summary, catenation and context threading are two techniques to reduce dispatch overhead. Catenation removes it completely, but is complicated to implement, and introduces the problem of code growth. Context threading is much simpler, and by avoiding costly indirect branches, makes dispatches relatively cheap. Both techniques provide a flexible foundation for further optimizations, but because it is simpler, context threading is more amenable to incremental improvements. It causes very minimal code growth, whereas catenation causes much more.

3 Related Work

See our earlier papers [3, 22] for references to literature relevant to the present work. Here we briefly note newer work and the most important projects in this space.



```
# find first power of 2
# greater than 100

proc find_pow {} {
  set x 1
  while {$x < 100} {
    incr x $x
  }
  return $x
}
```

```
 0  push1          0    # x = 1
 2  storeScalar1   0
 4  pop

 5  jump1         +7

 7  loadScalar1    0    # x += x
 9  incrScalar1    0
11  pop

12  loadScalar1    0    # if x < 100
14  push1          1    #    goto 7
16  lt
17  jumpTrue1    -10

19  loadScalar1    0    # return x
21  done
```

```
synthesized code for jump:
      ba    loop_test
      nop

loop_body:
      native code for loadScalar
      native code for incrScalar
      native code for pop

loop_test:
      native code for loadScalar
      native code for push:
            add    %l6, 4, %l6                  ; incr VM stack ptr
            sethi  %hi(operand), %o1            ; *** object to push
            or     %o1, %lo(operand), %o1       ; *** object to push
            st     %o1, [%l6]                   ; store to top of VM stack
            ld     [%o1], %o0                   ; last 3 instructions incr
            add    %o0, 1, %o0                  ; ref count of pushed object
            st     %o0, [%o1]

      native code for lt
      native code for jumpTrue:
            ...                                 ; get object, convert to boolean
            cmp    value, 1                     ; set condition code
            beq    loop_body
            nop
```

Figure 3. Catenation copies templates from interpreter code in virtual program order. Native control flows naturally from one to the next, without dispatch. Virtual operands and branches are both converted to native.

3.1 VM Dispatch

Ertl and Gregg [7, 8] have thoroughly studied VM performance at the microarchitectural level, using detailed cycle-accurate simulations of many dispatch strategies in many different VMs. They find that indirect branch mispredicts are a major performance obstacle in their VMs of interest, and discuss various hardware and software techniques for dealing with this problem.

The basic idea we call catenation (but not template specialization) was surveyed early by Rossi and Sivalingam [15], who credit it to Kenneth Oksanen. Piumarta and Riccardi [14] rediscovered similar techniques, and show how to portably apply them in a real implementation, using the Ocaml VM. Their approach is elegant, but still uses conventional dispatch for opcodes which implement virtual flow opcodes or whose bodies contain C function calls, which can't be easily relocated on some architectures because they are pc-relative.

Ertl and Gregg [7] call this technique *dynamic superinstructions*. They improve it to handle virtual control flow. Later they show [9] how to easily detect bodies which are not trivially relocatable, by comparing two bodies compiled from separate copies of the same C source. All these techniques are more portable than catenation, and they do not make radical changes to bytecodes, such as eliminating the virtual program counter or specializing virtual operands into native.

Ertl and Gregg [10] later do implement make those changes, in a technique very similar to catenation, but more portable and refined. It performs well on Forth's low-level opcodes.

3.2 Tcl VM performance

Adam Sah [17] created a dynamically-typed caching Tcl object system to improve performance while preserving Tcl's "everything is a string" semantics. He also caches script parsing results, but does not compile. Lewis [11] incorporated all those ideas and added the bytecode compiler and VM for Tcl version 8.0, which has had perhaps the largest impact on Tcl performance in practice.

Rouse and Christopher [16] implemented the commercial ICE 2.0 static (ahead-of-time) compiler for Tcl, targeting C and, later, Lewis's Tcl 8 bytecodes. They report creating infrastructure for optimization using static type annotations, while preserving dynamic features such as variable traces. They achieved approximately 30% execution time reduction over Tcl 8.0's bytecode compiler.

Cuthbert's `kt2c` [4] system seems intended for intellectual property protection, rather than performance improvement. It is an incomplete prototype, but translates ahead-of-time Tcl 8 bytecodes to C source code in external files, which is then compiled by a C compiler. Whereas catenation uses native code templates made from the compiled opcode bodies, `kt2c`'s templates are the bodies in C source form. The system contains some type inference infrastructure, which appears to be unused.

The commercially available TclPro [18] (and now the ActiveState Tcl Dev Kit [1]) serializes and externalizes Tcl bytecodes, for intellectual property protection and also deployable packaging of Tcl software.

Miguel Sofer has made many improvements to the released Tcl VM, and experimented with other improvements to the opcode bodies (e.g. the s4 [19] extensions we use) and dispatch performance. The Tcl Wiki page [12] contains ideas about lower-level Tcl bytecodes which are crucial to making Tcl's VM more "efficient", as defined by Ertl and Gregg [8].

3.3 Context Threading

Zaleski et al. [24] explore the flexibility of context threading as a VM architecture, including potential for incremental development of features and performance in dynamic languages such as Java. Many of the ideas seem to be applicable to Tcl, with extremely late binding, dynamic typing and features such as variable traces.

4 Design and Implementation

We now build on the background in Section 2 to give details of our implementation of direct and context threading for Tcl.

4.1 Direct Threading for Tcl

For experimental purposes, we implement direct threading as a pass that translates the output of the Tcl ByteCode compiler, which is stored in the `ByteCode.codeStart` data structure. We allocate an array of words as large as the array of bytes, and, while copying the bytes to words, translate the opcodes to corresponding body addresses (using a table similar to the one described in Section 2.4). We also pack four-byte operands, which have been unpacked by the bytecode compiler into unaligned bytes, back into words. We leave empty words in place to preserve the invariant that the same virtual program counter can index into the same place in both the DTT and bytecode. We redefined the interpreter macros which access four-byte operands (e.g., `TclGetInt4AtPtr`), and changed the type of `vpc` to be `unsigned int *` instead of `unsigned char *`. Instead of all this, a release-quality implementation could simply change the bytecode compiler to output direct-threaded code.

With the DTT in place, dispatch from the end of each opcode body can then be implemented using the `gcc` computed-goto extension: `goto **pc`.

In addition to the savings from direct dispatch, using words instead of bytes means that dispatch and operand loads use aligned data. This helps the memory system compared to bytecode, which requires multiple loads, some of which are unaligned, and shifts and adds to re-assemble words from bytes. We have experimented with leaving the word operands unpacked, retaining multiple (aligned) loads and shifts. The cost of the multiple loads and shifts in overall Tcl performance is about 1%.

4.2 Context Threading for Tcl

Our context threading implementation for Tcl is slightly different than described in Section 2. The output is illustrated in Figure 4. The differences are consequences of two factors. First, we target the Sparc CPU, where all control transfer instructions (CTI), including `call`s, have a *delayed branch slot*, an instruction which executes after the CTI, even if the CTI is taken. Second, most of Tcl's opcode bodies contain C function calls. These clobber the link register which holds the return address, which must be saved to enable return to the CTT.

We handle both of these factors by emitting into the delay slot of the CTT `call` an instruction to save the link register, `%o7` in the Sparc ABI[1], to a register reserved for that purpose in the C stack frame (by declaring it as a local variable in the main interpreter function, `TclExecuteByteCode()`).

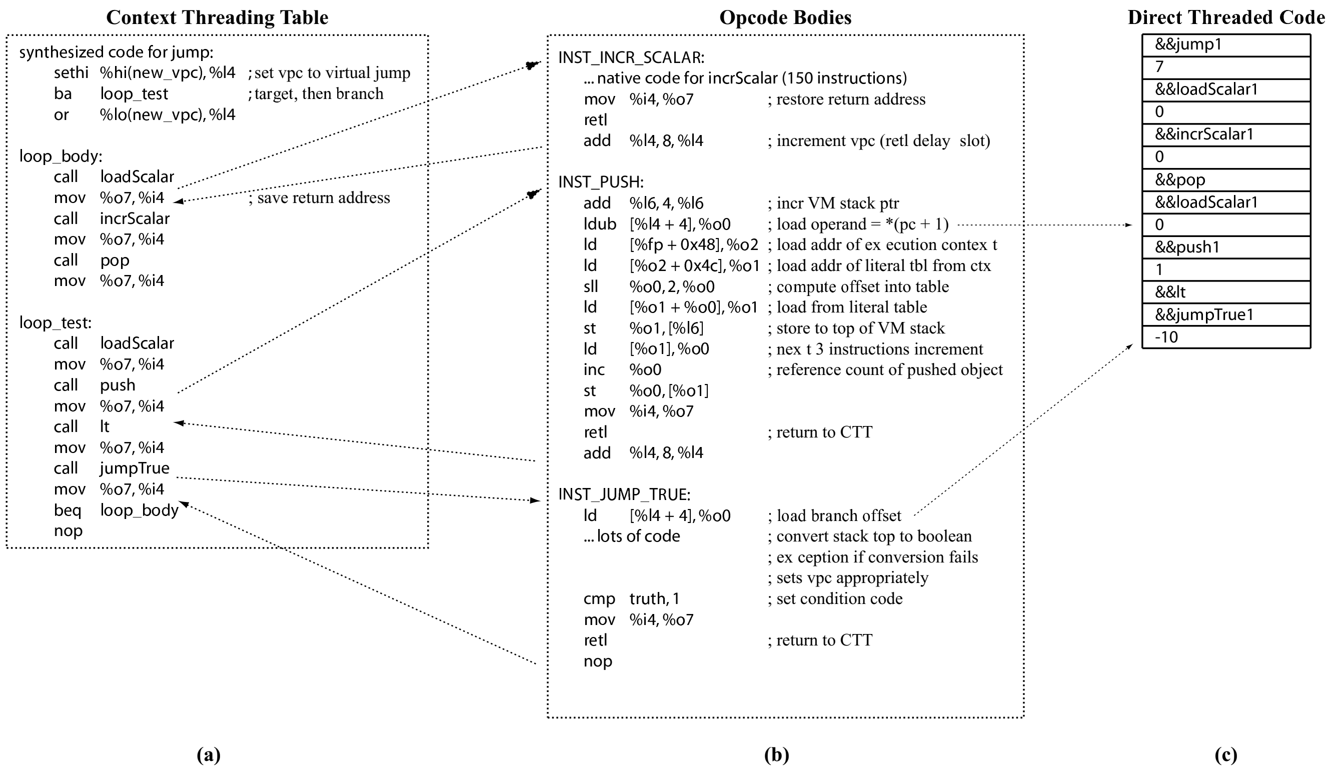
Only the CTT code is synthesized. The opcode bodies themselves require few changes. As in later versions of Tcl, we have moved all code that increments `vpc` to the end of the body, and combined it in a macro which also performs dispatch. For context threading, we redefine this macro to use a `gcc asm` to emit the code shown in Figure 4c: restore the saved link register, `retl` (return from leaf routine), and increment the `vpc`.

There are six cases in the main loop of our CTT-generating code, including the one above which simply calls a body. We describe the other five cases below.

4.2.1 Branch Inlining

Case 2 for the CTT handles the unconditional branch instructions. `jump1` and `jump4` become essentially a single unconditional native branch. Recall, however, that context threading maintains all interpreter state, including the `vpc`. Referring to the first three lines of Figure 4a, you will see a

[1]According to Sparc documentation [20], the value written by a call into the link register is visible to the instruction in the delay slot.



```
synthesized code for jump:
      sethi   %hi(new_vpc), %l4    ; set vpc to virtual jump
      ba      loop_test            ; target, then branch
      or      %lo(new_vpc), %l4

loop_body:
      call    loadScalar
      mov     %o7, %i4             ; save return address
      call    incrScalar
      mov     %o7, %i4
      call    pop
      mov     %o7, %i4

loop_test:
      call    loadScalar
      mov     %o7, %i4
      call    push
      mov     %o7, %i4
      call    lt
      mov     %o7, %i4
      call    jumpTrue
      mov     %o7, %i4
      beq     loop_body
      nop
```

```
INST_INCR_SCALAR:
    ... native code for incrScalar (150 instructions)
    mov    %i4, %o7                 ; restore return address
    retl
    add    %l4, 8, %l4              ; increment vpc (retl delay  slot)

INST_PUSH:
    add    %l6, 4, %l6              ; incr VM stack ptr
    ldub   [%l4 + 4], %o0           ; load operand = *(pc + 1)
    ld     [%fp + 0x48], %o2        ; load addr of ex ecution contex t
    ld     [%o2 + 0x4c], %o1        ; load addr of literal tbl from ctx
    sll    %o0, 2, %o0              ; compute offset into table
    ld     [%o1 + %o0], %o1         ; load from literal table
    st     %o1, [%l6]               ; store to top of VM stack
    ld     [%o1], %o0               ; nex t 3 instructions increment
    inc    %o0                      ; reference count of pushed object
    st     %o0, [%o1]
    mov    %i4, %o7
    retl                            ; return to CTT
    add    %l4, 8, %l4

INST_JUMP_TRUE:
    ld     [%l4 + 4], %o0           ; load branch offset
    ... lots of code                ; convert stack top to boolean
                                    ; ex ception if conversion fails
                                    ; sets vpc appropriately
    cmp    truth, 1                 ; set condition code
    mov    %i4, %o7
    retl                            ; return to CTT
    nop
```

&&jump1
7
&&loadScalar1
0
&&incrScalar1
0
&&pop
&&loadScalar1
0
&&push1
1
&&lt
&&jumpTrue1
-10

Figure 4. Output of our context threading implementation for Tcl. When reading Sparc assembly code, note that the instruction after each control transfer instruction (e.g. `call`, `beq`) is executed even if the branch is taken.

`sethi/or` pair (the Sparc idiom for setting a 32-bit constant in a register) which sets the `vpc`.

Conditional branches (case 3) are more complex. In addition to evaluating the condition, they must first coerce the input data to boolean type, potentially generating an exception. To see how we handle these, refer to the last four lines of Figure 4a. We first call variations of `jump_true` and `jump_false` bodies, amended with a `cmp` (compare) instruction to test the output of the virtual condition, and set the native condition code register, and finally return to the CTT. There, we emit a native conditional branch instruction. This native branch is now well-correlated with virtual program behavior, mobilizing hardware branch history ('way-ness') predictors.

4.2.2 Inlining Small Opcodes

Few of Tcl's opcode bodies are sufficiently simple to inline into the CTT. We profiled the benchmark suite to measure the dynamic frequency of each virtual opcode. `load_scalar1` (14%) and `push1` (12%) are most frequent. We also plotted the static length in native instructions of each opcode body. For case 4 of our CTT generation, we chose to inline `push1`, because it is one of the shortest (see Figure 4c.), with no internal control flow or function calls. An inlined copy is unique to the static virtual program instance, hence we can specialize as we did for catenation. While not shown, the outcome after inlining is, instead of a `call` to the generic `push` body, the CTT contains the seven instructions for `push` shown in Figure 3c, plus an `add` instruction to increment the `vpc`.

4.2.3 Optimizing Conditional Branches

The final and most effective optimization we apply is to completely inline certain virtual conditional branches, eliminating the need to call the body. These are critical low-level loop instructions, and yet the Tcl VM requires pushing a newly-allocated object on the VM stack, popping, and de-allocating. By studying traces of virtual execution, we found the most frequent "relational" opcodes to precede `jump_true` and `jump_false` were `gt`, `lt`, `foreach_step4`, and `try_cvt_to_numeric`. We created a variation of each. Instead of pushing the boolean result on the stack, the variation simply sets the CPU condition code. The inlined conditional branch can then be resolved in the CTT as a native conditional branch (similar to case 3, above), but without a call to the body. We must then emit three instructions to set the `vpc` for the taken and not-taken path.

This optimization can only be applied if the preceding virtual opcode and conditonal are in the same virtual basic block. Before generating the CTT, a pass over the bytecode finds and records all jump targets. This could be combined with the pass to compute the required length of the CTT for memory allocation purposes, but does not consume significant time. If a conditional jump does not meet this requirement, or is preceded by an opcode other than those above, we apply case 3.

We tried to use a similar transformation in the simple switched interpreter. We applied a peephole optimization to the same "relational" instructions to detect, execute and skip a successor conditional jump. (A similar change is in Tcl 8.5.) This change did not yield significant performance gains in our `s4` VM.

4.3 Porting Effort

Clearly any technique that involves synthesizing native code is not portable, so instead we think in terms of porting effort. Our implementation for Ocaml uses the `ccg` runtime assembler [13], and we had already ported it to x86 and PowerPC architectures. Porting basic context threading to the Sparc required four programmer hours. It is simpler than the Tcl implementation described here, requiring emitting only `call`, `nop`, `retl` and `add` instructions. Branch inlining and, especially, inlining other small opcodes, requires more instructions. It also requires interfacing with more interpreter state; e.g. it must emit code to set the `vpc`.

We use some unwarranted chumminess with the compiler, shown in Figure 5, to determine which register `gcc` has allocated for `vpc`, and the virtual stack pointer, as well as the variable we declare for saving the link register. Another approach would be to use `gcc` extensions to force the allocation of variables to specific registers. But this constrains the compiler's optimizer. We can also try to crack (disassemble and decode) the Sparc instructions which increment `vpc`. The decoding is perhaps no uglier than our approach, but locating the instruction to decode is difficult, and raises the kind of robustness issues we experienced with our catenation implementation.

The direct threading translation is implemented in 87 lines (36 semicolons.) Code to generate the CTT is 698 lines (293 semicolons), including debugging and instrumentation code, and several experimental variations. Simple CTT generation for Ocaml requires less than 60 lines, but doesn't perform any inlining. Catenation is 1352 lines (720 semicolons.)

```
enum { INST_ADD /* ... */};

void interpret (unsigned int *prog)
{
  unsigned int *vpc = prog;

  for (;;)
    switch (*vpc++) {
      case INST_ADD:
        asm volatile ("\
          .data                \n\
          .global vpc_location \n\
          .align 8             \n\
          vpc_location:        \n\
            .asciz \"%[vpc]\"  \n\
            .text" :: [vpc] "g" (vpc));
        /* actual work goes here */
        break;
    }
}

int main (int argc, char *argv [])
{
  extern char vpc_location [];

  printf ("gcc allocated vpc reg or stack slot: %s\n",
          vpc_location);
}
```

Figure 5. Programmers of mixed-mode systems must emit native code which interfaces with interpreter state, when most of the interpreter is coded in C. GNU C has several extensions useful for this kind of programming. Its `asm` statements normally output instructions, but here we output assembly to put a string in a global variable in the data segment. The string describes the storage assigned by gcc for the virtual program counter (technically valid only at this point in the program). In our implementation (not shown), we lookup this string in a table of register names, or parse it to obtain a stack frame offset for a C local variable. Then we use this to emit native instructions *at run time*.

5 Experimental Evaluation

In this section we evaluate the performance of our dispatch techniques in Tcl, with particular emphasis on context threading. After describing the setup, we report increasingly detalied statistics on a large set of benchmarks. We also introduce a simple context threading implementation for Ocaml on Sparc. This hints at the technique's potential, not fully realized for Tcl, and helps us compare with our earlier work (Ocaml on x86 and PowerPC).

5.1 Methodology

Our experimental platform is a SunFire V440 running Solaris 8 on four 1281 MHz UltraSPARC IIIi CPUs. Each has a 32 KB I-cache, 64 KB D-cache, and 1 MB on-chip L2 cache. All caches are four-way set associative. Prefetch and store buffers are 2 KB each. The pipeline is 14 stages and non-blocking. The chip is four-way superscalar, with two integer, two FP, one load/store and one branch unit. There is a 16k entry branch prediction table. We run on an unloaded machine.

We created a new Tcl command, nbench, which runs an arbitrary Tcl script for a given number of iterations (defaults to one) and collects data from the given Sparc performance counters by calling the Solaris cpc performance counter API:

```
tclsh% nbench {fact 5} results 1000
tclsh% parray results
results(exec_pic0) = 4206
results(exec_pic1) = 3516
results(overflow) = 0
```

The API and operating system virtualize the counters so other system activity is not counted. By default, the counters collect cycle and instruction counts, and include kernel activity, but via switches can also collect many micro-architectural statistics, such as pipeline stalls due to instruction cache misses. Unfortunately, we were unable to get reliable data from the counter which measures stalls due to indirect branch misprediction. The CPU can only collect two statistics at a time. We make several runs of each benchmark, each collecting cycles and one other statistic. We use the mean cycles value for analysis.

The canonical performance benchmark suite for Tcl is tclbench [21]. It contains over 500 benchmarks, mostly very small. We generally report only benchmarks with more than 10,000 bytecodes dispatched, because experimental noise is otherwise high. We exclude some benchmarks where the standard deviation is more than 2% of the mean. These tend to be system-only workloads, which are unlikely to be affected by our optimizations. Benchmarks such as WORDCOUNT, which contain system and significant user computation, are included.

We added instrumentation to measure bytecode compilation time, and time for our various native compilations, and also to collect dynamic execution counts for each virtual opcode. Because this instrumentation can perturb results, we run each set of benchmarks twice, with and without it. We include run-time re-compilation time (which occurs, for example, when expr's argument is unbraced), and add in initial compilation to the time for a *single* benchmark iteration. This is much tougher than tclbench, which amortizes compile time over all iterations. In real-world applications, performance sensitive code is typically executed more than once.

5.2 Results

We discuss three main aspects of performance. First, the running time of each benchmark is reported as time, in cycles. Cycles only count while executing the Tcl process or in the kernel on behalf of the process – other system processes are excluded. Second, we describe the micro-architectural behavior of a few interesting examples. Finally, we will discuss compilation costs.

5.2.1 Run time

Table 2 shows the overall change in performance, relative to switch threading. For instance, the first row of this table indicates that direct threading speeds up the average benchmark by a geometric mean of 4.3%. Furthermore, 88% (53 of 60 benchmarks) speed up. The remaining seven slow down. Overall, we see that catenation, direct and context threading all improve performance. While not shown, we have less formally measured the separate results of the various optimizations we tried. Direct threading gains about 1% overall as a result of modifying all immediate parameters to be words rather than bytes. Context threading uses the DTT for operands, and thus also benefits 1%. It also wins about 1 to 1.5% from inlining the push bytecode. Simple subroutine threading, without any branch or other inlining, performs roughly only as well as direct, gaining about 5%. Fully inlining certain conditional branch bytecodes, as described in Section 4.2.3, wins another 2 - 9%, depending on how loopy the workload is (as measured by how many dispatches it performs). All three dispatch techniques seem to win about 4% due to dispatch, and the rest due to other optimizations.

5.2.2 Overall tclbench performance

Figure 6 shows the elapsed time required for each benchmark summarized by Table 2. A bar is plotted for each of the 60 tclbench programs that dispatch more than 10,000 virtual instructions. To show the pattern of speedup across

Dispatch type	geom mean speedup	% of benchmarks improved
direct	4.3%	88.%
catenation	4.0	73.
context	5.4	88.
context + inline cond	12.0	97.

Table 2. Percentage geometric mean speedup of various dispatch techniques on UltraSPARC III CPU for the 60 programs in tclbench that dispatch 10,000 bytecodes or more.

all benchmarks, the bars on each plot are sorted by performance, so we cannot use these plots to compare specific benchmarks. For example, the upper left hand plot, Figure 6a, shows that Direct Threading slows seven benchmarks, the worst by about 5%, and speeds up the rest. Note that several bars are clipped in Figure 6b.

Direct threading loses on few benchmarks, but makes more modest performance improvements than the other techniques. Catenation produces the biggest wins, as well as the biggest losses. Context Threading wins more than Direct Threading, and loses roughly the same amount. Context Threading with inlined virtual conditional branches wins the most, and loses the least.

5.2.3 Selected Microarchitectural Detail

To investigate what causes these trends, Figure 7 reports the micro-architectural behavior of a few hand-picked benchmarks. Each cluster of bars indicates the performance of a benchmark. Each bar in a cluster indicates the performance of a modified version of the Tcl interpreter, relative to the performance of the same VM using switch dispatch. Consequently, the height of the bar corresponding to switch is 1.0. From left to right, the bars represent switch dispatch, direct threading, catenation, context threading, and finally context threading with inlined virtual conditional branches.

Each bar is made up of up to five stacks, which attribute the execution time to various factors. The bottom stack shows cycles when the CPU was stalled waiting for instruction cache lines to be read from memory. The second stack, 2nd_br, reports stalls when the Ultrasparc's pipeline refetches a branch when "two branch instructions line up in one 4-instruction group" [20]. It is relevant to the performance of context threading, because our insertion of returns into the bodies is not subject to a compiler's careful instruction scheduling, potentially very important for RISC CPUs. The third from top stack measures "Return stack stall", which indicates the extent to which the machine stalls because the return address stack mispredicted. We include this statistic to demonstrate that context threading, which relies on and places load on the RAS, does not overwhelm it. Finally, the top stack measures all other execution cycles, mainly real work. We measured *some* other statistics, such as data dependencies between instructions in the pipeline, and found they did not significantly differ between techniques.

The leftmost two benchmarks, MATRIX-mult and FACT, use Tcl as a general purpose programming language, calculating a 20 x 20 matrix multiplication, and a non-recursive calculation of the factorial of 16 (we run FACT 20000 times per benchmark iteration.) These benchmarks make relatively few calls to Tcl's large runtime system, and the inner loops are made up of relatively small bytecode bodies. These programs have a relatively high proportion of dispatch to real work, so we expect improvements from our dispatch oriented techniques. Direct threading shortens the dispatch path and shows good speedup. We inspected the virtual instructions in the inner loop of FACT and MATRIX, and observe that the same opcode appear more than once followed by different opcodes. This suggests that they will suffer from the context problem. We thus expect Context Threading to outperform Direct Threading – see Section 5.2.5 for discussion of why it does not. Catenation removes all dispatch, and shows a good speedup. Context threading improves the dispatch, but not enough to do as well as catenation (which has removed the dispatch code altogether). Context threading with inlined virtual branches does the best because it optimizes the inner loop closing branch more than the catenation.

The next two benchmarks, LOOP_foreach and LOOP_while, are small microbenchmarks that execute empty loops. The inner loop of LOOP_while includes a conditional branch which enables Context Threading with inlined branches to beat catenation. LOOP_while shows some of our best speedups, about 40% for Context Threading with inlining.

The MD5 benchmark illustrates a problem with all our techniques when the code footprint of a program becomes large. MD5 has a huge inner loop, with 118 lines of Tcl source to compute the md5 hash function. This compiles to 1518 virtual instructions, and recursively enters the interpreter to handle called Tcl procs. Catenation's copies yield many native instructions, inducing many instruction cache misses, and slowing down by 40%. Context threading, which generates much less code per virtual instruction, still causes enough misses to balance out any improvements to dispatch.

The two BASE64 benchmarks illustrate how I-cache stalls can be low enough that dispatch improvements compensate for them. Generally, however, catenation's performance is too unstable for Tcl, with its large opcodes.

Finally, we show MATRIX-transposition because it has the highest proportion of initial compile time of any benchmark with over 1,000 dispatches, even with the normal switched interpreter. Most benchmarks spend less than 1% of time compiling, but in this case our techniques' increased compile time hurt performance, although only catenation becomes slower than switch.

5.2.4 Ocaml Performance

For the sake of comparison, Figure 8 shows how simple subroutine-threaded Ocaml performs on Ultrasparc. Unlike our earlier work with Pentium 4 and PowerPC, we haven't yet implemented branch- and other inlining. As Figure 1 showed, Ocaml bytecodes are much lighter than

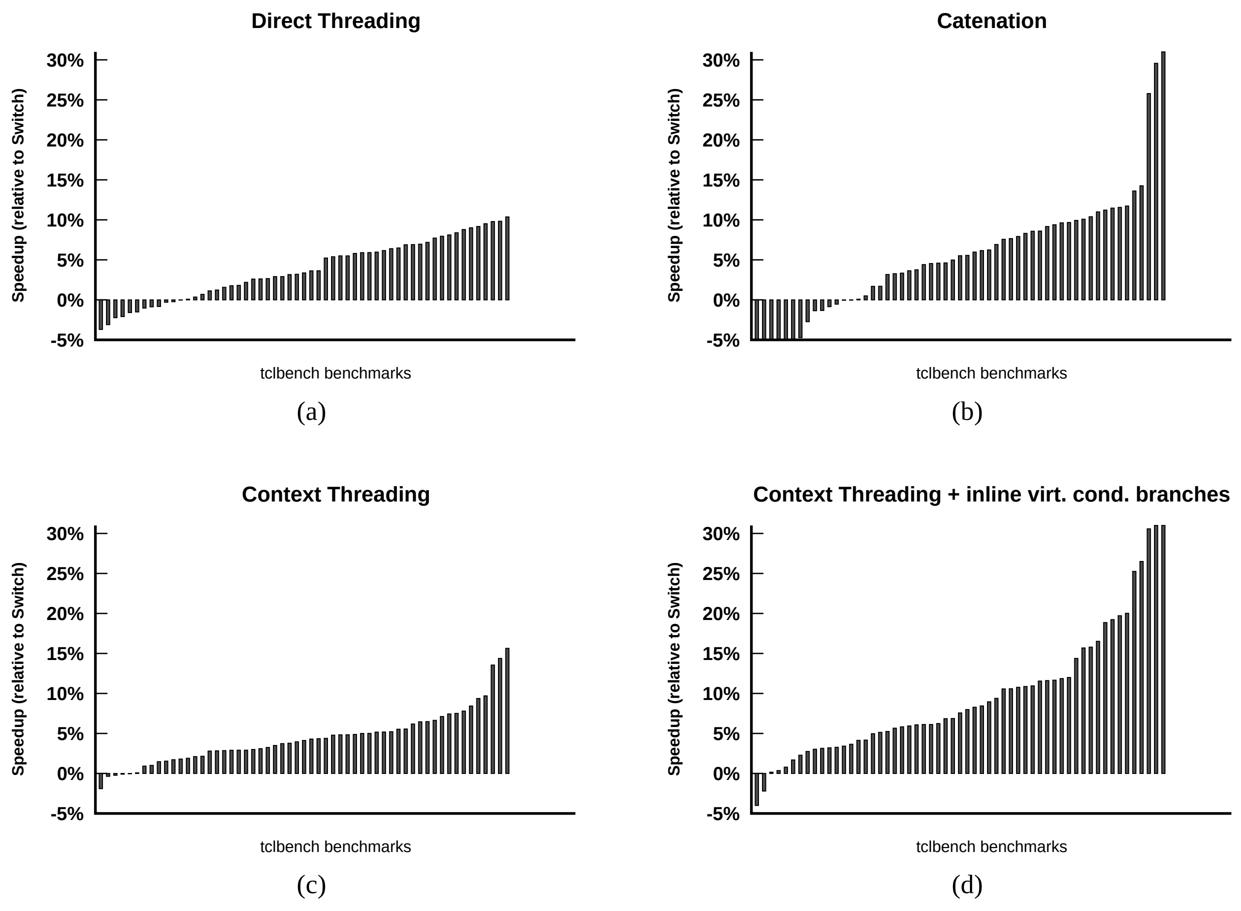


Figure 6. Performance of various Tcl dispatch techniques, compared to Switch, for benchmarks that execute more than 10000 virtual instructions. Higher is faster. The clipped values in (b) are -130, -44 (analyzed in detail in Figure 7), -43, -11, -10, and -5.45.

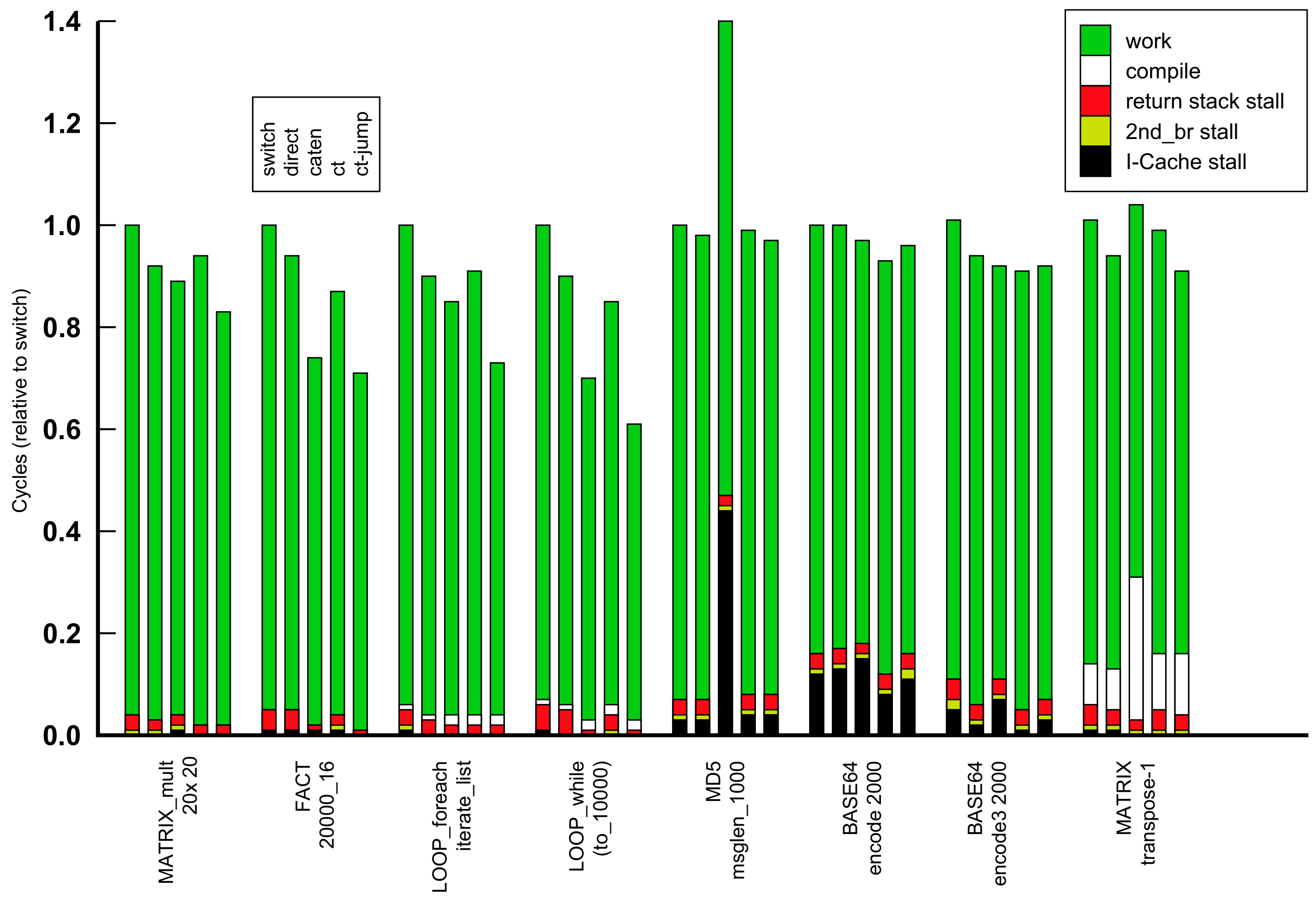


Figure 7. Comparison of some microarchitectural statistics for each dispatch technique on selected benchmarks. Lower is faster. The five stacks in each cluster, from left to right, are switch, direct, catenation, context threading, and context threading with inlined virtual conditional branches.

Tcl's. This implies that there is much more opportunity to improve Ocaml performance by optimizing dispatch, and this is confirmed by the speedups obtained. Context threading improves Ocaml performance, as reported in Figure 8, much more than it speeds up Tcl. Because we were unable to make stable measurements of stalls due to mispredicted branches, we can not conclusively report that these speedups are caused by a reduction in mispredicted branches. Nevertheless, we speculate that reducing these branches causes the speedup on Ultrasparc, as it did on the PowerPC and Pentium 4, where we used performance counters to measure branch target stalls and events.

5.2.5 Context Threading improves less than expected

Above we pointed out that we expect Direct Threading to suffer many branch mispredictions in the inner loop of the FACTORIAL and MATRIX-mult benchmarks. Hence, we expected that context threading should win over direct threading, as it did for Ocaml. For Tcl, it does not, and we do not conclusively understand why. We believe it is related to saving the link register after calls from the CTT, and restoring before returning, necessary because of C function calls in bodies. We have experimented with removing these saves, instead re-loading the return address from the DTT only after calls. With the `s4` VM we are using, these calls are rare on critical paths. But this did not improve performance. More investigation is required.

5.2.6 Compilation Time

As a baseline for compilation speed, we use the time for compiling Tcl source to bytecode. Translating bytecode to direct threaded code adds approximately 6% to this time. Catenation adds an average of 44%. Compilation time for all our techniques is sensitive to the benchmark, particularly

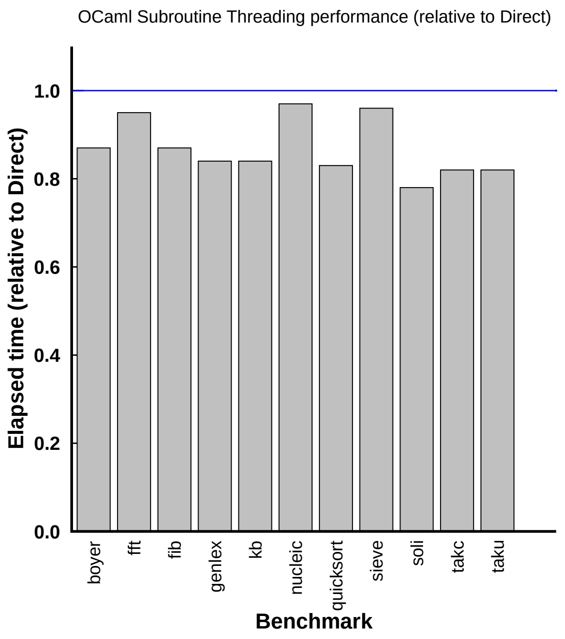


Figure 8. Subroutine threading speeds up Ocaml's virtual machine on all benchmarks on the UltraSPARC III, even compared to the already-efficient default technique, direct threading.

for catenation, which can require as little as 10% or as much 150% more time than the bytecode compiler. On average, and including the time for the translation to direct threaded code, context threading adds about 35%. With the inlined conditional branch optimization it adds 38%.

Only four benchmarks in the entire suite use run-time recompilation. None are shown in Figure 7. EXPR-unbraced is so short (64 dispatches) that it is dominated by compilation time, even for switch. Context threading's additional compile time slows it down by 80%, but this amounts to $40\mu s$, which we consider insignificant. The benchmark GC-Cont_expr::cGCC_5000, also uses run-time recompilation, and all our techniques increase this time, but all have fewer total cycles, because the compiled code executes faster.

6 Conclusions and Future Work

We have implemented context threading for Tcl on Sparc, speeding up 97% of benchmarks by an average of 12%. Compilation time increases only modestly, by 30%.

Our Ocaml experiment shows that context threading can perform well on Sparc, even without fancy optimizations. The Sparc delayed branch slot slightly complicates implementation, but does not hamper performance. On the other hand, Tcl (like some other languages with dynamic types) has high-level opcodes, which creates several problems. Dispatch overhead is low, limiting the efficacy of our dispatch-oriented techniques. Direct and basic context threading improve overall performance some, but the real gains of our work come from the optimizations made possible by context threading's flexibility.

Our earlier technique, catenation, is hampered by poor portability, high compilation time, and unstable performance for Tcl. Tcl's large opcode bodies tax the instruction cache, even in a simple switch-based interpreter. This is unusual for a virtual machine. Part of the problem may be that most of the opcodes contain the same code to do things like coerce types and handle errors. Catenation's copying of these large opcodes amplifies the problem. On the other hand, Ertl reports that catenation [7] (and specialization [10]) can work well for Forth's small bodies.

When used as a general purpose language, e.g., to compute the factorial function, Tcl is 10 - 20 times slower than compiled C. In this context, 10% speedup does not seem very significant. We believe a new VM architecture is warranted, with lower-level bytecodes.

Dynamic typing requires polymorphic bytecodes, but these should be split into separate pieces for detecting types, loading fields, performing arithmetic, etc.. A type inference [2] system should be able to overcome most of the overhead of dynamic types. The VM needs an intrinsic integer type, because the overhead of 'boxing' and 'unboxing' `Tcl_Obj`s wastes time and obscures program semantics, limiting value re-use. Machine integers are still important – in loop induction variables and list indices, for example – even if Tcl's superior (well-defined) integer semantics are preferred for most program variables. Lisp and Smalltalk use tagged pointers to efficiently implement integers alongside dynamically typed objects.

Perhaps only after these changes is it appropriate to apply efforts such as our CTT inlining optimizations.

Lower level bytecodes will obviously result in *more* bytecodes required to represent a virtual program. They will be more amenable to analysis and optimizing transformations, and also more suitable as input to a JIT's intermediate program representation. The best JITs apply powerful time-consuming optimizations to hot code, while interpreting cold code. Lower-level bytecodes will actually *increase* interpreter dispatch counts, but we believe that the efficient dispatch techniques we have presented will limit increases in overhead.

Even with the limited dispatch overhead, we believe context threading for Tcl should perform better. We would like to port to x86 (possibly after integrating the `ccg` [13] runtime assembler, which eased our Ocaml ports) and study behavior there. We'd also like to learn more about the Ultrasparc's pipeline, try to get reliable statistics from its counter to measure branch target predictor performance, and use the interpreter-oriented 'prepare-to-branch' instruction referred to in Sun marketing material, but about which we've been unable to find any technical data. Finally, we'd like to inline more opcodes, such as the critical path of `load_scalar`, and try to manage I-cache misses induced by the CTT by inlining less as its size grows.

7 Acnowledgements

We thank the Tcl Community for excellent software, and our colleagues Angela Demke Brown, Marc Berndl, Tarek Abdelrahman, Norman Wilson, and Dan Astoorian.

References

[1] ActiveState Corporation. Tcl Dev Kit [online]. Available from: `http://www.activestate.com/Products/Tcl_Dev_Kit/`.

[2] O. Agesen, J. Palsberg, and M. I. Schwartzbach. Type inference of SELF: Analysis of objects with dynamic and multiple inheritance. *Lecture Notes in Computer Science*, 707:247–68, 1993. Available from: `http://citeseer.nj.nec.com/agesen93type.html`.

[3] M. Berndl, B. Vitale, M. Zaleski, and A. D. Brown. Context threading: A flexible and efficient dispatch technique for virtual machine interpreters. In *Proc. Third Annual Intl. Symposium on Code Generation and Optimization (CGO)*, March 2005.

[4] D. Cuthbert. The Kanga Tcl to C converter [online]. 2000. Available from: `http://sourceforge.net/projects/kt2c/`.

[5] M. A. Ertl. Speed of various threading techniques on several processors [online]. 1998. Available from: `http://www.complang.tuwien.ac.at/forth/threading/`.

[6] M. A. Ertl and D. Gregg. The behavior of efficient virtual machine interpreters on modern architectures. *Lecture Notes in Computer Science*, 2150, 2001.

[7] M. A. Ertl and D. Gregg. Optimizing Indirect Branch Prediction Accuracy in Virtual Machine Interpreters. In *Proc. of PLDI*, 2003.

[8] M. A. Ertl and D. Gregg. The Structure and Performance of *Efficient* Interpreters. *Journal of Instruction-Level Parallelism*, 5:1–25, 2003.

[9] M. A. Ertl and D. Gregg. Combining stack caching with dynamic superinstructions. In *Proc. 2nd IVME*, pages 7–14, 2004.

[10] M. A. Ertl and D. Gregg. Retargeting JIT compilers using C-compiler generated executable code. In *Proc. Intl. Conference on Parallel Architectures and Compilation Techniques (PACT 04)*, 2004.

[11] B. Lewis. An on-the-fly bytecode compiler for Tcl. In *Proc. of the 4th Annual Tcl/Tk Workshop*, 1996.

[12] Miguel Sofer et al. MS's bytecode engine ideas [online]. Available from: `http://wiki.tcl.tk/1683/`.

[13] I. Piumarta. CCG: Dynamic code generation for C and C++ [online]. 1999. Available from: `http://regal.lip6.fr/projects/vvm/realizations/ccg/ccg.html`.

[14] I. Piumarta and F. Riccardi. Optimizing direct-threaded code by selective inlining. In *Proc. of PLDI*, pages 291–300, 1998.

[15] M. Rossi and K. Sivalingam. A survey of instruction dispatch techniques for byte-code interpreters. Technical Report TKO-C79, Helsinki University Faculty of Information Technology, May 1996.

[16] F. Rouse and W. Christopher. A Typing System for an Optimizing Multiple-Backend Tcl Compiler. In *Proc. of the 5th Annual Tcl/Tk Workshop*, 1997.

[17] A. Sah. TC: An efficient implementation of the Tcl language. Master's thesis, UC Berkeley, 1994. Available from: `http://sunsite.berkeley.edu/TR/UCB:CSD-94-812`.

[18] Scriptics Corporation. The TclPro Development Kit [online]. September 1998. Available from: `http://www.tcl.tk/software/tclpro/`.

[19] M. Sofer. Tcl Engines. Available from: `http://sourceforge.net/projects/tclengine/`.

[20] Sun Microelectronics. *UltraSPARC III Cu User's Manual*. 2004.

[21] Tcl Core Team. TclLib benchmarks [online]. 2003. Available from: `http://www.tcl.tk/software/tcllib/`.

[22] B. Vitale and T. S. Abdelrahman. Catenation and operand specialization for Tcl VM performance. In *Proc. 2nd IVME*, pages 42–50, 2004.

[23] R. Yung. Design of the UltraSPARC instruction fetch unit. Technical Report SMLI TR-96-59, Sun Microsystems, Dec 1996.

[24] M. Zaleski, M. Berndl, and A. D. Brown. Mixed mode execution with context threading. In *Proc. of IBM CASCON 2005*.