Catenation and Specialization for Tcl Virtual Machine Performance

Benjamin Vitale
bv @ cs.toronto.edu
Department of Computer Science

Tarek S. Abdelrahman
tsa @ eecg.toronto.edu
Edward S. Rogers Sr. Department of Electrical and Computer Engineering

University of Toronto
Toronto, M5S 3G4 Canada

ABSTRACT

We present techniques for eliminating dispatch overhead in a virtual machine interpreter using a lightweight just-in-time native-code compilation. In the context of the Tcl VM, we convert bytecodes to native Sparc code, by concatenating the native instructions used by the VM to implement each bytecode instruction. We thus eliminate the dispatch loop. Furthermore, immediate arguments of bytecode instructions are substituted into the native code using runtime specialization. Native code output from the C compiler is not amenable to relocation by copying; fix-up of the code is required for correct execution. The dynamic instruction count improvement from eliding dispatch depends on the length in native instructions of each bytecode opcode implementation. These are relatively long in Tcl, but dispatch is still a significant overhead. However, their length also causes our technique to overflow the instruction cache. Furthermore, our native compilation consumes runtime. Some benchmarks run up to three times faster, but roughly half slow down, or exhibit little change.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Interpreters*

General Terms

bytecode interpreters

Keywords

virtual machines, just-in-time compilation, Tcl

1. INTRODUCTION

Many portable high level languages are implemented using virtual machines. Examples include traditional programming languages such as Java and scripting languages such as Tcl, Perl, and Python. The virtual machine may provide portability and intimate linkage with a high-level runtime system that is more sophisticated than a typical general purpose hardware machine — for example, it may provide garbage collection, object systems, and other services.

One penalty of the virtual machine approach is lost performance. Because the virtual machine code is interpreted, it runs slower than native code. Scripting systems are usually designed for portability, flexibility, extensibility, and expressiveness, not high performance. However, they are powerful enough that they are used to implement entire software systems, and thus their performance is important. The efforts to address the VM performance problem can be divided into two main camps. Clever interpreter designs such as threaded code can make the process of interpretation faster. Alternatively, compilers can translate the virtual machine code into native code, and execute that directly. Just-in-time (JIT) compilers make this translation at runtime. Avoiding a separate, slow, static compilation step means that a JIT compiler can try to be a drop-in replacement for an interpreter, preserving the interactive use of a virtual machine system, especially important for scripting. Unfortunately, a JIT compiler is typically large, and complex to develop and portably deploy. Furthermore, the delay for compilation at runtime may interfere with interactive use.

In this paper, we present an implementation of a relatively new point in the design space between interpretation and JIT compilation. Our technique, which we call *catenation*, eliminates the overhead of instruction dispatch in the virtual machine. Catenation creates a native code version of a bytecode program by making copies of the native code used by the interpreter to execute each of the virtual instructions in the original program. The resulting native code has certain properties that enable several optimizations, in particular, the removal of operand fetch code from the bytecode implementations ("bodies".)

We have implemented this technique for the Tcl scripting language. Originally, Tcl interpreted scripts directly from source, but in 1995 evolved to compile scripts to bytecodes on the fly, and then interpret the bytecodes [11]. While the bytecode compiler improves performance significantly, many Tcl applications still demand more speed. Our system

extends the bytecode compilation with an additional native compilation step, targeting the Sparc microprocessor.

The programming effort required in our approach is substantially less than a full-blown JIT compiler, but can still improve performance. It eschews expensive optimizations, and reuses code from the interpreter, which also reduces semantic errors. The compiler uses fixed-size native code templates, and precomputes as much information as possible, and thus runs quickly.

Catenation builds on techniques used by Piumarta and Riccardi to implement their *selective inlining* [15]. This involves making relocated copies of the native code used by the interpreter to implement each virtual opcode. The original technique placed several constraints on which virtual opcodes could be compiled, because not all code (e.g., `call` instructions to pc-relative targets) is trivially relocatable. These constraints do not apply to most of the short, low-level opcodes of Forth, or the VM used to evaluate selective inlining, Ocaml. However, the constraints present a severe problem for a virtual machine such as Tcl's, which has much higher level opcodes. We remove the constraints, so that all Tcl opcodes code can be moved, albeit at the expense of portability.

Catenation allows compilation of every instruction in a bytecode program into a *separate* unit of native code. This property, along with the infrastructure we built to relocate all code, allows us to perform several optimizations. We:

1. specialize the operands of bytecode instructions into the operands of the native instructions in the template, and employ some constant propagation,
2. convert virtual branches into native branches, and
3. elide maintenance of the virtual program counter, and, if necessary, rematerialize it.

In evaluating its performance, we found that our technique improves some benchmarks significantly, but slows others down. We attribute the slowdown to more instruction cache misses, resulting from the code growth caused by catenation. As we discuss in Section 7, we feel this result is interesting in light of recent work by Ertl and Gregg [8]. They find that a replication technique similar to our catenation also causes code growth, but that the code growth rarely causes enough I-cache misses to hurt performance. The key distinction, perhaps predicted by Ertl and Gregg, is that Tcl and many other popular VMs are not "efficient interpreters." The large opcode bodies in such VMs suffer less dispatch overhead, and thus gain less from its removal. More importantly, the large bodies, when replicated, cause excessive code growth.

The rest of this paper is organized as follows. Section 2 reviews the structure and performance of some common techniques for interpreter dispatch. Section 3 presents catenation, our approach to eliminating dispatch overhead. Section 4 introduces operand specialization, which removes the overhead of bytecode operand fetch. Section 5 provides some details of our implementation, including coaxing the C compiler into generating templates, and subsequent object-code processing. We evaluate the performance of these techniques in Section 6. Finally, in Sections 7 and 8 we discuss related and future work, and conclude.

```
enum { INST_ADD, INST_SUB, ... };

void interpret (unsigned char *program)
{
  unsigned char opcode, *pc = &program [0];
  int sum;

  for (;;) {
    opcode = *pc;
    switch (opcode) {
      case INST_ADD:
        sum = POP() + POP();
        PUSH (sum);
        pc += 1;
        break;
      case INST_SUB:
        /* ... */
        break;
      /* ... other instruction cases ... */
    }
  }
}
```

Figure 1: Simple interpreter dispatch using C `for` and `switch` statements

2. TRADITIONAL INTERPRETERS

The most fundamental job of the virtual machine is to dispatch virtual instructions in sequence. In this section, we compare some of the techniques used in practice. A typical dispatch loop is shown in Figure 1. Here dispatch is implemented as a C `switch` statement, and kept separate from the implementation of opcode semantics. A C compiler will likely translate this to a table lookup. While efficient in principle, the compiler often includes extra instructions for a bounds check. Furthermore, there is only one copy of this table lookup code, and each opcode body must end with a jump back to it. The two loads (get opcode from program, then lookup opcode in switch table) and indirect branch required are pipeline hazards on typical microprocessors. Worse, there is only one static copy of the indirect branch to dispatch to many possible opcode targets, and thus the processor's branch predictor does poorly, because it uses the address of the branch instruction as the main key when predicting the target [8].

Interpreters can employ many other dispatch techniques [3, 6, 7]. In the "Function table" approach, each opcode body is a separate function. Dispatch consists of a for loop, looking up the address of each function in a table, and an indirect function call. By making the table lookup explicit in the code, instead of letting the compiler generate it (as with `switch`), it can sometimes be more efficient, avoiding the spurious bounds check, for example. However, the function call, with associated stack frame setup, register saving, etc. is more expensive than the simple jump required in *switch*.

```
# compute n!, i.e. factorial function

proc factorial {n} {
    set fact 1
    while {$n > 1} {
        set fact [expr {$fact * $n}]
        incr n -1
    }
    return $fact
}
```

Figure 2: Tcl code to compute factorial function

Another approach is to remove the dispatch from the external `for/switch` loop, and instead place it at the end of *each* opcode body. The interpreter community refers to this as *shared* dispatch. This saves the jump from the end of each body to the single dispatch code. More importantly, it gives the branch predictor many static indirect jumps to work with, dramatically improving its chances of success. If switch-like table lookup dispatch is placed in each opcode, this is known as *token threading*. Alternatively, in *direct threaded* code, the bytecode program represents opcodes by the native addresses of their implementations, rather than one-byte opcode numbers, saving the table lookup. This dispatch can be expressed in roughly three machine instructions. Its execution time depends largely on the time to handle the indirect branch to the next opcode, but on typical workloads it averages about 12 cycles on a modern CPU such as the Ultrasparc-III we use.

On our Sparc, switch dispatch requires roughly 12 dynamic instructions, or 19 cycles. We found token threading takes about 14 cycles. If the bodies (the "real work") of the typical instruction are small (as with Forth), this overhead can dominate interpreter performance. Even with larger bodies, such as Tcl's, the overhead is significant. For example, the Tcl code to compute the factorial function shown in Figure 2 uses, on average, about 100 cycles per virtual opcode. Thus, indirect threaded dispatch accounts for 16% overhead. Even direct threading, at 12 cycles, takes significant time. We thus pose the question: Can *all* dispatch be eliminated? And, if so, is this profitable?

3. CATENATION

To remove all dispatch, we must instead execute native code. The typical approach is a full-blown compiler, including an intermediate representation and code generator, which can do much more than simply avoid dispatch. In this section, we describe a simpler approach to just avoid dispatch.

Consider the problem of interpreting the bytecode program in Figure 3a. The interpreter uses the (imaginary) native instructions in Figure 3b to implement the `push` and `add` opcodes, and instruction dispatch. In Figure 3c, we show in bold the dynamic sequence of *useful* (i.e., non-dispatch) native instructions executed when interpreting this bytecode program. Now we are set to understand the idea of "catenation," which is our technique for improving interpreter performance. If we simply copy into executable memory the sequence of useful work instructions — those shown in bold — we have "compiled" a native code program with exactly

```
push  2
push  3
add
```

(a) Sample bytecode program. Note, opcode `push` is used twice, with different operands.

push:	$p_1 p_2 p_3$
add:	$a_1 a_2 a_3 a_4 a_5$
dispatch:	$o_1 o_2 o_3$

(b) Definitions of virtual instruction bodies in native code. Each p_i, a_j, etc. represents one native instruction.

$o_1 o_2 o_3$ $\mathbf{p_1 p_2 p_3}$ $o_1 o_2 o_3$ $\mathbf{p_1 p_2 p_3}$ $o_1 o_2 o_3$ $\mathbf{a_1 a_2 a_3 a_4 a_5}$

(c) Dynamic sequence of native instructions executed when interpreting program in (a).

$p_1 p_2 p_3$ $p_1 p_2 p_3$ $a_1 a_2 a_3 a_4 a_5$

(d) Static sequence of native instructions emitted by catenating compiler for program in (a).

Figure 3: *Compiling* bytecode objects into catenated copies of native code from the interpreter avoids dispatch overhead

the same semantics as the interpreted version. This new program has no dispatch overhead.

Of course, most programs contain branches and loops, so dynamic execution paths do not look, as they do in this case, exactly like static code in memory. Catenation handles control flow by changing virtual machine jumps into native code jumps. In Figure 3d, the native code for the second virtual instruction (`push 3`) starts at native address 4. Thus a virtual jump to virtual pc 2 would be compiled into a native jump to address 4.

Catenation is based on the use of fixed-size *templates* of native code for each virtual opcode. Each template is self-contained, and does not depend on the context of the opcode being compiled. The separate cases of an interpreter largely meet this description, and indeed we generate our templates directly from the interpreter. Following Piumarta and Riccardi [15], we leverage interpreter-oriented extensions in the GNU C compiler to treat C language labels as values, delineating the boundaries of native code for each virtual opcode, as shown in Figure 4.

Catenating a unit[1] of Tcl bytecodes is a two pass process. The first pass computes the total size of the resulting native code, by adding up the sizes of the templates for each bytecode instruction in the unit. It then allocates a native code buffer of the appropriate size. The size of a template may be slightly larger than the interpreter case, to allow room for instructions we synthesize and emit at the end of a template. These are used for jumps, debugging, and profiling instrumentation. The pass also builds a map of the native code

[1]By *unit* of code, we mean an object in the Tcl object system whose type is `code`. This typically corresponds to a `proc` (procedure), but might also be an anonymous block of code, such as a command typed interactively at the interpreter prompt, or passed to the `eval` command, etc.

```
typedef void *ntv_addr;

struct { ntv_addr start; ntv_addr end } inst_table = {
  /* Labels as Values */
  { &&inst_incr_immed_begin, &&inst_incr_immed_end },
  /* ... */
}

#define NEXT_INSTR goto *inst_table [*vpc].start

/* increment integer object at top of stack by arg */

inst_incr_immed_begin:        /* Label */
{
  int incr_amount;
  Tcl_Obj *object = *tosPtr; /* tos is Top Of Stack */

  /* skip opcode; extract arg */
  incr_amount = *(uchar *) ++vpc;
  object->int_val += incr_amount;
  ++vpc; /* advance vpc over argument */

inst_incr_immed_end:          /* Label */
  NEXT_INSTR;
}
```

Figure 4: Using `gcc`'s labels-as-values extension to delineate the interpreter case for an opcode

offset for each bytecode. The offset map is used to determine the native destination address for jump instructions, and for exception handling, described below.

The second pass then copies the native code for each template. For example, to compile the `incr_immed` opcode shown in Figure 4, the essential operation consists of looking up the starting and ending addresses of the native code for this opcode in the interpreter, by referring to the `inst_table`, and then `memcpy()`'ing the code into the native code buffer.

There are a few things to observe about the code in Figure 4. First, the compiled code still has to fetch operands from the bytecode stream. It does this using the virtual program counter, `vpc`. The code uses the entire interpreter runtime, interfacing with it at the register level.

Also note that while the instruction template ends with the label `inst_incr_immed_end`, we still include the `NEXT_INSTR` code *after* the template proper. This code consists of a traditional token threaded dispatch. Its purpose is to force the C optimizer to build a control flow graph which reflects the fact that any opcode case may be preceded or followed by any other opcode case, which is precisely the situation after catenation (and during normal interpretation.)

However, even position independent code output by the C compiler is not intended for this sort of relocation; indeed, considerable transformation of the output is required to make catenation work. We undertake some of these transformations at interpreter build time, and others during catenation. We describe this in some detail in Section 5.

By itself, catenation is not a true compilation. Among other things, the resulting code still refers to operands in the bytecode instruction stream. We address this problem in the next section.

4. SPECIALIZATION

We would like to eliminate the fetching of operands from the bytecode stream. Among other things, this will reduce the need to maintain the virtual program counter. Catenated code provides the foundation for another a technique that removes the fetches. Next, we describe this technique, which we call *Operand Specialization*.

The implementation of a virtual opcode in the original interpreter is generic, in the sense that it must handle any instance of that opcode, in any location in any bytecode program, and with any valid values for operands. After catenation, on the other hand, there is a separate instance of native code implementing each static bytecode instruction in the program. During code generation, we know the operands of each instruction, and can treat them as runtime constants. We specialize a template with the values of these constants. Essentially, we are compiling bytecode instruction operands into native instruction operands.

Now, we will describe how to enhance the templates so they are appropriate for specialization. Given a virtual opcode with operands, one of the first tasks of the interpreter body implementing that opcode is to fetch and decode the operands. They are usually placed in variables. We remove the interpreter code that implements this fetch, and replace the variable with a magic number of an appropriate size. At specialization time, we substitute the magic number with the real operands from the bytecode program. We include

```
#ifdef INTERPRET

 #define MAGIC_OP1_U1_LITERAL \
   codePtr->objArrayPtr [TclGetUInt1AtPtr (pc + 1)]
 #define PC_OP(x)          pc ## x
 #define NEXT_INSTR        break

#elseif SPECIALIZE

 #define MAGIC_OP1_U1_LITERAL \
   (Tcl_Obj *) 0x7bc5c5c1
 #define NEXT_INSTR        goto *jump_table [*pc].start
 #define PC_OP(x)          /* unnecessary */

#endif
...
case INST_PUSH1:
  Tcl_Obj *objectPtr;

  objectPtr = MAGIC_OP1_U1_LITERAL;
  *++tosPtr = objectPtr;
  Tcl_IncrRefCount (objectPtr);
  PC_OP (+= 2);
  NEXT_INSTR;
```

Figure 5: Source for interpreter implementation of `push` opcode, with variation to generate template suitable for specialization.

```
add    %l6, 4, %l6          ; incr VM stack ptr
add    %l5, 1, %l5          ; inc vpc past opcode.
ldub   [%l5], %o0           ; load operand
ld     [%fp + 0x48], %o2    ; load addr of execution context
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           ; next 3 instructions increment
inc    %o0                  ; reference count of pushed object
st     %o0, [%o1]
inc    %l5                  ; increment vpc

sethi  %hi(0x800), %o0      ; rest is dispatch to next instr
or     %o0, 0x2f0, %o0
ld     [%l7 + %o0], %o1
ldub   [%l5], %o0
sll    %o0, 2, %o0
ld     [%o1 + %o0], %o0
jmp    %o0
nop                         ; branch delay slot unusable
```

Figure 6: C compiler's assembly language translation of code in Figure 5, using the INTERPRET variation.

several assertions in the compiler initialization code to ensure this results in a suitable template, as described in more detail below. Refer to Figure 5 to see the interpreter C code for the `push` opcode, with our conditionally-compiled variation to produce a template for specialization. Figures 6 and 7 show the assembly language output by the C compiler for each variation.

In addition to removing the load of the operand from the bytecode stream, Figure 7 shows that several related instructions were also removed. In the Tcl VM, the operand of the `push` opcode is not the address of the object to be pushed. Rather, the operand is an unsigned integer index into the *literal table*, a table of objects stored along with the bytecode. We treat the index, and the table, as runtime constants at specialization time, then perform a simple constant propagation. The result is that `push` is reduced from twelve Sparc instructions (twenty including dispatch) to seven, including one load instruction, instead of four. This is an extreme example, because `push` is much shorter than most Tcl opcodes, whose bodies can average hundreds of instructions. However, it is significant, because `push` is an extremely common opcode, both statically and dynamically. We also employ this table-lookup constant propagation on the `builtin_math_func` opcode, which takes a small unsigned integer argument to indicate which of several math functions should be invoked (e.g. `sin`, `sqrt`, etc.)

Catenation runs very fast, because it requires only copying native code, followed by a few fixups and linking. Specialization consumes more time, running for every operand of every bytecode instruction. It still can run quickly, because our templates are fixed size, and we can pre-compute the offsets where bytecode operands can be specialized into native code operands. In the next section, we give details of this and other aspects of our implementation.

```
add    %l6, 4, %l6              ; incr VM stack pointer
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               ; next 3 instructions incr
add    %o0, 1, %o0              ; ref count of pushed object
st     %o0, [%o1]
```

Figure 7: Template for `push` opcode, compiled from Figure 5 using the SPECIALIZE variation. Note that it is much shorter than Figure 6. The native operands of the instructions marked *** are points for operand specialization. The `sethi`/`or` instruction pair is the Sparc idiom for setting a 32 bit constant in a register.

5. PREPARING AND USING TEMPLATES

We build our templates using the output of the GNU C compiling a modified version of the Tcl interpreter. This assembly language output itself requires two small build-time transformations, and is then conventionally linked into the Tcl virtual machine. Finally, at runtime, our catenating compiler is divided into several phases, which include transformations on the templates. In this section, we describe each of these steps.

5.1 Building Templates

To each opcode case, we append a macro for token threaded dispatch, so that the optimizer is constrained to produce templates suitable for catenation, as discussed in Section 3. We redefine operand fetch macros to various magic constants, instead of code to perform actual fetches from the bytecode stream. We use a constant of the appropriate size and type so that the template is efficient. For example, if a virtual opcode takes a one byte integer operand, we use a magic constant less than 256. For a four byte operand, we use a constant that forces the compiler to generate code general enough to load any four byte operand. On the Sparc, this is a two instruction sequence. Occasionally, we had to manually introduce uses of the C `volatile` keyword, or other tricks, to force the optimizer to not propagate the magic constants too deeply into the opcode implementation. Other times, our code-rewriting system, discussed below, handles the optimizer's output.

At build time, the bodies are assembled into a single C function, with each delineated by labels as shown in Figure 4. We generate C code to store all these labels in a table, indexed by opcode number. Then, the function is compiled to assembly, with `gcc` set to optimize. The resulting assembly is largely suitable for templates, but we must post-process it to undo the work of the C optimizer in two situations. The first occurs when a native branch instruction in the middle of a body branches to the exit of the body. The normal target of the branch is one native instruction beyond the end of the case, which is precisely what is required for catenation. However, the optimizer exploits the Sparc delayed-branch slot [18], schedules an instruction of the (extraneous) dispatch code after the branch, and changes the branch target to *two* instructions beyond the end. Suppressing the optimizer delayed-branch scheduler using a command-line switch to the C compiler was not a reasonable option, because this optimization is important to the performance of

Sparc code, and we want to leave most such transformations intact.

The second problem occurs when the compiler applies a *tail-merging* [14] optimization to two or more opcode cases. This results in the cases sharing part of their implementation, but catenation requires that each case be self-contained, because later passes may make changes to the code, and these must be separate for each opcode.

Using text-processing techniques on the assembly output, we "deoptimize" the delayed branch and tail-merging transformations, if they have been applied in the places described. Together, they affect 14 of 97 opcodes. All build-time manipulation and assembly post-processing for deoptimization is performed with Tcl scripts. These scripts (after bootstrapping) are executed by a Tcl "interpreter" which incorporates our native code compiler.

After post-processing, the templates are assembled, and the entire Tcl virtual machine is linked together in the traditional fashion. In the rest of this section, we describe the runtime manipulations of the templates to accomplish compilation of Tcl bytecode.

5.2 Compiler Initialization

During and after catenation, we apply many small changes to the fixed-length templates to complete the process of compilation. These changes fix the code so that it executes correctly after it is relocated, and also handle operand specialization. We call these fixups *patches*, and refer to the overall process as *patching*. For example, when a native `call` instruction targeting a C library subroutine (e.g. `malloc`) is relocated during catenation, the target will be wrong, because Sparc `call`s are pc-relative – that is, the target depends on the address of the `call` instruction itself. A patch is used to correct this problem, by adjusting the target address in the relocated instruction. This patch type actually applies to any pc-relative instruction whose target lies outside the template being moved. A few other patch types are described below.

To accelerate catenation in general, and patching in particular, we analyze the native code once at initialization time. We locate the starting and ending points of the template for each opcode, and find and store the native code offsets, types, and sizes of each patch required by each opcode. Using this information, a patch can be applied very quickly, essentially in two steps. First, some information is extracted from the bytecode stream – for example, a one byte unsigned integer operand. Then, it may be filtered or manipulated in some way. Finally, it is applied to the output native code template. Thus, we store for each patch an input and output type, along with its relative offset from the beginning of the template. Input types essentially select the kind of patch necessary, and include, for example, plain operands of various size and signedness, destination addresses of virtual jumps, and operands which need translation through the literal table or builtin math function table.

After a patch has fetched and transformed the input data, the data must be placed into the template at the appropriate location. This means changing the operands of one or more *native* machine instructions. On a RISC processor like the Sparc, there are only a few formats, and indeed only a few instructions, to recognize for patching: loading a small immediate integer constant (one which fits in 13 bits), loading larger constants, calls, and a few branches. In the case of virtual branch opcodes, the patch may also synthesize (rather than rewrite) the native instruction (branch false, branch true, or branch always.) The analysis expects certain instruction patterns to appear a certain number of times (usually once for each operand of a given opcode.) The magic constant is used to locate the pattern, and confirm it appears the correct number of times. If these assertions fail, the interpreter code must be retooled. This happened during development with a handful of opcodes, because our pattern-matcher did not yet recognize all variations of compiler output.

The input and output types for each patch can largely be determined from the type of virtual opcode (e.g. virtual branch) or operands, or during the analysis phase (e.g. native pc-relative `call`s.) There is a small table of exceptions, coded by hand to handle special cases. Our code contains a large number of assertions. During development, we identified the exceptions when a small number of assertions failed.

5.3 Compilation

A code unit is compiled to native code only the first time it is executed. The process runs quickly, because most of the work has been done in the initialization pass. The compilation is simply "interpretation" of the patches for the catenated program.

Each patch can be interpreted very quickly, because it requires only a load from the bytecode stream, possibly another to index through constant tables, sometimes one or two adds to handle pc-relative instructions, possibly a lookup in the virtual-to-native pc map, a few operations for masking, shifting the data into destination native instruction, then store. On average, this requires about 120 μs per patch. The initialization pass requires about 4 ms. An initial version of our system did not perform the separate initialization pass. While it was still profitable to compile code that was executed many times, the faster technique vastly broadens the applicability of catenation.

The final patch type is `UPDATE_PC`. While we don't maintain the `vpc` during execution, the nature of catenated code makes it is easy to rematerialize. `UPDATE_PC` is used on the right-hand-side of assignments to the interpreter variable `vpc`, and is patched with the constant value `vpc` would have had during execution of the original interpreter. In catenated code there is a separate body of native code for every static bytecode instruction, and so the value of `vpc` is implicitly known. For example, an `UPDATE_PC` in the native code emitted for the virtual instruction at `vpc` 5 is patched to the constant 5. We set the value only in exception handling paths in the interpreter code. To conserve space, we will not describe Tcl exception handling here, except to say that stack unwinding and handler resolution is done at runtime, using a map from `vpc` of exception to `vpc` of handler. Another map allows us to report source line number diagnostics on uncaught errors. We could have translated all these maps to native addresses, entirely eliminating the `vpc`

in our case, but the current implementation works well, and it's useful to demonstrate rematerialization of `vpc`, because one can imagine other interpreters or dynamic recompilation systems requiring it.

5.4 Implementation Notes

The implementation is divided into several modules, following the structure described above. The pre-computation of templates and patches is implemented in 771 lines of C, containing 462 semicolons. The code generator, which uses the templates to implement catenation and operand specialization, is 581 lines long, or 258 semicolons. While these modules include some profiling instrumentation, an additional 535 lines (240 semicolons) of Tcl extension commands to collect detailed statistics when running on real machines. Finally, for the simulation experiments described below, we created 1181 lines (513 semicolons) of statistics extensions to both Tcl and our simulator.

As described above, we also made small (typically one- or two-line) changes to several Tcl VM opcode implementations. Excluding these changes and the instrumentation code, roughly two weeks of effort were required to code the template and code generators. However, we spent considerably more time finding and resolving subtle bugs, such as the "deoptimizations" (requiring 250 lines of Tcl) described in Section 5.1.

6. PERFORMANCE EVALUATION

To measure the impact of catenation and specialization in our implementation, we constructed two sets of experiments. The first measures execution time on a small number of benchmarks from the Tcl performance benchmark suite [21], to determine if our modified interpreter actually improves performance. We capture detailed micro-architectural statistics. The second experiment tries to answer questions raised by the first, by running a larger set of benchmarks on both our catenating Tcl interpreter, and the original, while varying only the size of the instruction cache. This hypothetical scenario requires a simulation infrastructure, because of the lack of variety in I-cache sizes within generations of the Sparc CPUs. Using CPUs from different generations, which have substantial architectural differences, would confound the results.

6.1 Cycle counts and I-Cache Misses

The highest precision clock available for our first experiment is the CPU's cycle counter, available using the Sparc performance counters [20], which are present in the `sparcv8` instruction set on the Ultrasparc-II and later CPUs. Two 64-bit hardware registers can each be programmed to collect any one of a wide variety of events, such as cycles, retired instructions, cache misses, branch mis-predicts, etc. To facilitate the experiment, we implemented a Tcl command to collect performance statistics while running arbitrary Tcl code, and separately track bytecode compilation time, native compilation time, and execution time. We can also choose whether to include events during execution of system code on behalf of the application. We ran our benchmarks on an otherwise unloaded machine, and exclude events incurred while the operating system was executing other programs. The machine is a Sun Microsystems SunBlade 100

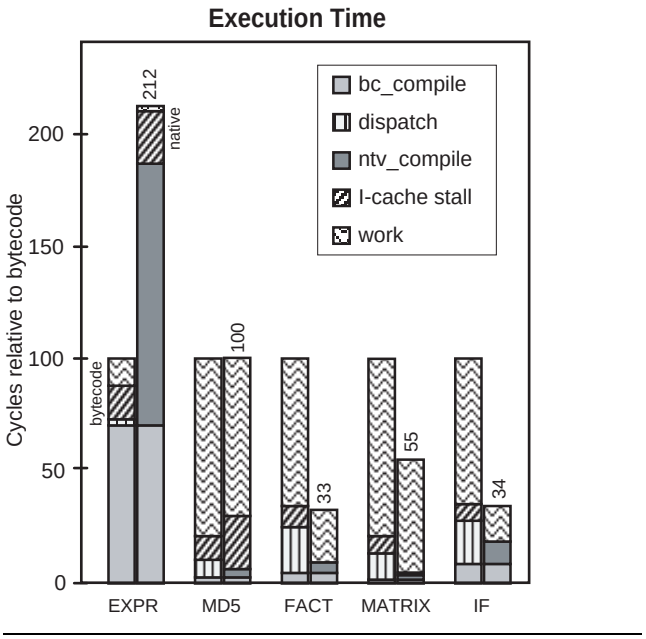

Figure 8: Performance counter benchmarks.

with a 502 MHz Ultrasparc IIe, 640 MB RAM, a 16 KB 2-way set-associative instruction cache, and a 16 KB direct mapped data cache. The 4-way unified Level 2 cache is 256 KB. It runs the Solaris 8 operating system. The Tcl benchmarks are: runtime evaluation of a simple arithmetic expression of constants and variables, a hash function involving many bitwise operations, the factorial function, multiplication of two 15x15 matrices, and a long if-then-else tree.

The results are depicted in Figure 8. For each benchmark, the bar on the left shows the performance of the benchmark with the original bytecode interpreter. The bar on the right shows the same, but with our catenating compiler. The performance of each benchmark is measured by the total number of execution cycles, normalized with respect to the original bytecode interpreter. Cycles are broken down into those spent on useful work, those devoted to dispatch, those devoted to bytecode compilation, and to native compilation, and those wasted in instruction cache stalls.

As intended, catenation removes all dispatch overhead in all cases. Furthermore, the three optimizations it enables – operand specialization, virtual to native branch conversion, and elimination of the virtual program counter – substantially reduce the amount of work cycles in all cases. For three cases, FACT, MATRIX, and IF, the result is a significant improvement in total execution time.

Our techniques do not always reduce execution time. Sometimes it is mostly unchanged, or actually increased. There are two main problems, one of which shows up in the EXPR benchmark, and the other in MD5. While not shown, we measured dynamic *instruction* counts in addition to cycles. In every case besides EXPR, catenation reduces instruction counts, because it saves dispatch, and benefits from the subsequent optimizations. EXPR-unbraced, however, requires a large amount of compilation time. In fact, this benchmark

is a contrived idiom to force continuous late recompilation due to Tcl semantics. This idiom is well known by experienced programmers, and generally avoided. We include it here to underscore that any workload which spends lots of time recompiling will do poorly in any JIT compiler, including ours. The JIT must regenerate native code each time bytecode is regenerated. Our JIT is relatively fast, but typically requires between 100% and 150% as much time as the bytecode compilation.

The MD5 benchmark exhibits a more serious problem. The catenated code does slightly less work than the interpreted version, but the overall execution time stays the same. As the chart shows, increased I-cache misses defeat the advantages of removing dispatch and improving work efficiency. Now, in many cases, catenated code exhibits *better* I-cache performance, because useful code is tightly packed in memory, without intervening unused opcode implementations. However, catenation causes major code growth – on average, we measure a factor of 30. The expanded code's working set can easily exceed a typical 32 KB I-cache, and consequently I-cache stall cycles can overwhelm the savings from removing dispatch and operand fetch.

6.2 Varying I-cache Size

To further explore the effect of I-cache misses induced by code growth, we ran the entire set of 520 Tcl benchmarks, with the original and catenating interpreters, under the Simics [12] full machine simulator configured in separate experiments with I-caches of 32, 128, and 1024 KB. A fourth experiment simulated an infinite I-cache, by running with no latency to the external L2-cache. The benchmarks are all quite short, but some solve interesting problems such as gathering statistics on DNA sequences.

At the realistic 32 KB I-cache size, 54% of benchmarks actually run slower using catenated code. The larger 128 and 1024 KB caches slowed 45% and 34% of benchmarks, respectively. Even with the infinite I-cache, 18% of benchmarks slow down. This is because there is not enough execution time to amortize the cost of native code compilation, except in four (less than 1% of) benchmarks, due to continuous recompilation, as described earlier.

7. RELATED WORK

Ertl and Gregg [8] evaluate the performance of a large number of advanced interpretation techniques. Their focus is the high proportion of indirect branch instructions in an interpreter (Forth) with short opcode bodies, and the predictability of those branches by a pipelined micro-architecture. Their techniques include forming superinstructions and making replicas of code. Their *dynamic superinstructions across basic blocks with replication* technique is very similar to our catenation, except that they leave some dispatch in place to handle virtual branches, whereas we remove all dispatch. Instead, their key goal is to have more indirect branch instructions – one for each static bytecode instruction – whose behavior and context precisely match the behavior of the bytecode program. This yields much better branch prediction. They also find that I-cache stalls induced by code growth due to replication are not a major problem for their "efficient" Forth VM, which has short opcode bodies. On the other hand, our VM, with large opcode bodies, encounters major I-cache stalls on many benchmarks.

QEMU [4] is a CPU emulator which employs techniques very similar to our catenation and operand specialization. It is more portable than our system, supporting a plurality of host and target architectures, some including full-system emulation. Where we use magic numbers to identify points for specialization in our templates, QEMU stores operands in global variables, and exploits the ELF relocation metadata generated by the linker. Some unwarranted chumminess is still required to elide function prologues, etc.. CPU opcode instruction bodies tend to be small, with complex implementations "outlined" to called functions, and thus catenation performs well.

Brouhaha [13] is a portable Smalltalk VM that executes bytecode using a direct threaded interpreter built from native code templates. The templates are written in a style similar to function-table dispatch, but then, like our system, after compilation the assembly is post-processed to remove prologues, epilogues, and make other transformations. Where we post-process using Tcl, Brouhaha uses `sed`, and does significantly more rewriting. Little runtime rewriting seems required, although neither superinstructions, replication, nor operand specialization are implemented.

DyC [10] is a dynamic compilation system based on programmer annotations of runtime constants, which are used in dynamic code generation to exploit specialization by partial evaluation. The authors motivate one of their papers using the example of a bytecode interpreter – in this case, `m88ksim`, a simulation of a Motorola 88000 CPU. The input data for this benchmark is a bytecode program. DyC treats the entire bytecode program as a constant, and, using an optimization they call *complete loop unrolling*, accomplishes essentially the same effect as our catenation. The system applies traditional optimizations after partial evaluation. This process is static and quite expensive, and thus might not be appropriate for a dynamic scripting language which frequently compiles and re-compiles. At static compile time, they specialize their runtime specializer, so it is pre-loaded with most of the analysis for optimization and code generation. This is more general than, but similar to, our system of patches, which pre-computes the necessary fix-ups. They report speedups of 1.8 on `m88ksim`, but do not discuss the complexities of I-cache and code explosion.

Trace-based dynamic re-compilation techniques [2] also have the promise to automatically accomplish effects similar to catenation (and many other optimizations.) Sullivan et al. [19] show how to extend Dynamo's infrastructure, by telling it about the virtual program counter, so that it is able to perform well while executing virtual machine interpreters, whereas it had previously done poorly.

There have been several efforts to improve Tcl performance. The `kt2c` system [5], while unfinished, uses the bytecode for a given function to build a C file containing a huge "superinstruction" implementing all the bytecode instructions in the function. It converts virtual jumps into C goto statements. It performs some analysis of the types and locations of objects pushed onto the stack, but no use is made of this

information. For Python, the similar "211" system [1] compiles extended basic blocks of Python bytecode into superinstructions by concatenating the C code for the interpreter cases, performing some peephole optimization, and then invoking the C compiler in a separate, offline step. We are sympathetic to Aycock's suggestions that a VM based on registers and lower-level bytecodes would be more amenable to compilation.

The `s4` [17] project has experimented with improved dispatch techniques for the Tcl VM, such as token threading, and many of its results have been folded back into the production Tcl release, improving performance.

The ICE 2.0 Tcl Compiler project [16] created a commercial stand alone static (ahead-of-time) Tcl version 7 to C compiler in 1995, and then a later version in 1997 that also targeted bytecode and included a conservative type-inference system, setting the stage for classical compiler optimizations. The compiler offered an approximately 30% improvement in execution time over the Tcl 8.0 bytecode compiler. Both the ICE compilers were static, that is, required a separate compile step. This precludes using the compiler as a drop-in replacement for the original interactive Tcl interpreter, an important modality for scripting languages. The source code of the ICE Compilers was never released to the research community, and is no longer actively developed.

8. CONCLUSION AND FUTURE WORK

In this paper, we presented techniques that allow us to "compile away" the bytecode program, acting as a very naive JIT compiler. However, almost all the runtime infrastructure of the interpreter remains, so it may be better to think of this system as an advanced interpretation technique, rather than a true compiler. In practice, there is a range of techniques from simple interpretation to advanced dynamic compilation, which trade off implementation size and complexity, start-up performance, and portability. We offer a new point on this spectrum, and implement it in a non-portable but transparent and complete Tcl interpreter system.

Our experimental evaluation indicates that catenation and the optimizations it enables improve the performance of several benchmarks, often significantly. On the other hand, for some benchmarks, the I-cache stalls often induced by code growth from catenation degrade performance. The overall effect, on a typical microprocessor, is that about half our benchmarks speed up, by as much as a factor of 3, but the other half slow down.

Ertl and Gregg [9] suggest that dispatch is more of a problem for small opcode bodies, and that the architecture of "inefficient" popular VMs should be improved before advanced interpretation techniques are applied. On the other hand, we find that dispatch *is* a source of overhead in Tcl's large opcode bodies, and that its removal via catenation significantly improves performance for some benchmarks. Thus, we believe our techniques are applicable to VMs with large opcode bodies.

Furthermore, Ertl and Gregg found [8] found that I-cache stalls induced by code growth due to replication are not a major problem. In contrast, in our system, we find that I-cache stalls are a problem. This is perhaps not surprising given the larger opcodes in the Tcl VM.

Exploiting the C compiler to build templates for code-copying is a clever and largely portable technique. We have extended it, allowing all opcode bodies to be moved, but sacrificed portability. Furthermore, our experience leaves us with the opinion that the approach is too brittle for general experimentation, depending excessively on the compiler and machine architecture. A more explicit code generation infrastructure would be more flexible for exploring the interesting issues surrounding native compilation and optimization of dynamic languages such as Tcl.

Catenation implicates the classic inlining tradeoff, and a more selective technique is required, perhaps preferring code expansion to dispatch overhead only where profitable. Mixed-mode execution might facilitate this, and we would like to explore this in future work. A key related question is deciding when to compile and when to interpret. A useful heuristic might be based on the potential correlation between opcode body size and I-cache performance. To study this, we would like to apply our technique to other interpreters, including some with shorter, lower-level bodies than Tcl. Finally, we would like to explore *outlining* of large instruction bodies, perhaps by moving them to separate functions, and calling these from the body.

9. ACKNOWLEDGMENTS

We thank Angela Demke Brown, and Michael Stumm for their ideas and encouragement. Colleagues Mathew Zaleski and Marc Berndl offered stimulating brainstorms. Virtutech Inc. provided an academic license for its excellent Simics full system simulator. Finally, the anonymous reviewers' feedback was invaluable.

10. REFERENCES

[1] J. Aycock. Converting Python Virtual Machine Code to C. In *Proc. of 7th Intl. Python Conf.*, 1998.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proc. of PLDI*, 2000.

[3] J. R. Bell. Threaded code. *Communications of the ACM*, 16:370–372, 1973.

[4] F. Bellard. Qemu x86 cpu emulator [online]. 2004. Available from: `http://fabrice.bellard.free.fr/qemu/`.

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

[6] R. B. Dewar. Indirect threaded code. *Communications of the ACM*, 18:330–331, 1973.

[7] M. A. Ertl. Threaded code [online]. 1998. Available from: `http://www.complang.tuwien.ac.at/forth/threaded-code.html/`.

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

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

[10] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. DyC: an expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 248(1–2):147–199, 2000.

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

[12] P. S. Magnusson and F. L. et al. SimICS/sun4m: A Virtual Workstation. In *Proc. of the Usenix Annual Technical Conference*, 1998.

[13] E. Miranda. BrouHaHa - A Portable Smalltalk Interpreter. In *Proc. of OOPSLA'87*, pages 354–365.

[14] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

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

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

[18] SPARC International Inc.A. *The SPARC Architecture Manual, Version 8*. 1992.

[19] G. T. Sullivan, D. L. Bruening, I. Baron, T. Garnett, and S. Amarasinghe. Dynamic native optimization of interpreters. In *Proc. of the 2003 workshop on Interpreters, Virtual Machines and Emulators*.

[20] Sun Microelectronics. *UltraSPARC IIi User's Manual*. 1997.

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

[22] B. Vitale. Catenation and Operand Specialization for Tcl Virtual Machine Performance. Master's thesis, University of Toronto, 2004.