Catenation and Operand Specialization

For Tcl VM Performance

Benjamin Vitale\textsuperscript{1}, Tarek Abdelrahman
U. Toronto

\textsuperscript{1}Now with OANDA Corp.
Preview

• Techniques for fast interpretation (of Tcl)
• Or slower, too!
• Lightweight compilation; a point between interpreters and JITs
• Unwarranted chumminess with Sparc assembly language
Implementation

- Runtime Initialization
  - Template extraction
  - Patch synthesis

- JIT Compiler
  - Catenation
  - Specialization
  - Linking

- Native machine code

Tcl source → Tcl bytecode compiler → JIT Compiler → Native machine code
Outline

- VM dispatch overhead
- Techniques for removing overhead, and consequences
- Evaluation
- Conclusions
Interpreter Speed

• What makes interpreters slow?

• One problem is dispatch overhead
  – Interpreter core is a dispatch loop
  – Probably much smaller than run-time system
  – Yet, focus of considerable interest
  – Simple, elegant, fertile
Typical Dispatch Loop

for (;;)
    opcode = *vpc++
    switch (opcode)

        case INST_DUP
            obj = *stack_top
            *++stack_top = obj
            break

        case INST_INCR
            arg = *vpc++
            *stack_top += arg

    Fetch opcode
    Dispatch

    Real work

Fetch operand
Dispatch Overhead

- Execution time of Tcl INST_PUSH

<table>
<thead>
<tr>
<th></th>
<th>Cycles</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real work</td>
<td>~4</td>
<td>5</td>
</tr>
<tr>
<td>Operand fetch</td>
<td>~6</td>
<td>6</td>
</tr>
<tr>
<td>Dispatch</td>
<td>19</td>
<td>10</td>
</tr>
</tbody>
</table>
Goal: Reduce Dispatch

<table>
<thead>
<tr>
<th>Dispatch Technique</th>
<th>SPARC Cycle Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>for/switch</td>
<td>19</td>
</tr>
<tr>
<td>token threaded, decentralized next</td>
<td>14</td>
</tr>
<tr>
<td>direct threaded, decentralized next</td>
<td>10</td>
</tr>
<tr>
<td>selective inlining (average) Piumarta &amp; Riccardi PLDI’98</td>
<td>&lt;&lt;10</td>
</tr>
<tr>
<td>?</td>
<td>0</td>
</tr>
</tbody>
</table>
Native Code the Easy Way

• To eliminate *all* dispatch, we must execute native code

• But, we’re lazy hackers

• Instead of a writing a real code generator, use interpreter as source of templates
Interpreter Structure

- Dispatch loop sequences bodies according to virtual program
Catenation

Virtual Program

- No dispatch required
- Control falls-through naturally

```
push 0
push 1
add
```

```
copy of code for inst_push

<table>
<thead>
<tr>
<th>copy of code for inst_push</th>
</tr>
</thead>
<tbody>
<tr>
<td>copy of code for inst_add</td>
</tr>
</tbody>
</table>
```

“Compiled”
Native Code
Opportunities

- Catenated code has a nice feature
- A normal interpreter has one generic implementation for each opcode
- Catenated code has separate copies
- This yields opportunities for further optimization. However…
Challenges

• Code is not meant to be moved after linking

• For example, some pc-relative instructions are hard to move, including some branches and function calls

• But first, the good news
Exploiting Catenated Code

- Separate bodies for each opcode yield three nice opportunities
  1. Convert virtual branches to native
  2. Remove virtual program counter
  3. Reduce operand fetch code to runtime constants
Virtual branches become Native

bytecode

L1: dec_var x
    push 0
    cmp
    bz$_{\text{virtual}}$ L1
done

native code

L2: dec_var body
    push body
    cmp body
    bz$_{\text{native}}$ L2
done body

- Arrange for \texttt{cmp} body to set condition code
- Emit synthesized code for \texttt{bz}; don’t memcpy
Eliminating Virtual PC

• vpc is used by dispatch, operand fetch, virtual branches – and exception handling
• Remove code to maintain vpc, and free up register
• For exceptions, we rematerialize vpc
Rematerializing vpc

- Separate copies
- In copy for vpc 1, set vpc = 1

1: inc_var 1
3: push 0
5: inc_var 2

Bytecode

code for inc_var
  if (err)
    vpc = 1
  br exception

code for push

code for inc_var
  if (err)
    vpc = 5
  br exception

Native Code
Moving Immovable Code

• pc-relative instructions can break:

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Relative Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7000:</td>
<td>call +2000</td>
<td>9000</td>
<td>&lt;printf&gt;</td>
</tr>
<tr>
<td>3000:</td>
<td>call +2000</td>
<td>5000</td>
<td>&lt;?????&gt;</td>
</tr>
</tbody>
</table>
Patching Relocated Code

- *Patch* pc-relative instructions so they work:

```
3000: call +2000 (5000 <?????>)
```

```
3000: call +6000 (9000 <printf>)
```
Patches

- Objects describing change to code
- Input: **Type**, **position**, and **size** of operand in bytecode instruction
- Output: **Type** and **offset** of instruction in template
- Only 4 output types on Sparc!

**Input Types**

<table>
<thead>
<tr>
<th>ARG</th>
</tr>
</thead>
<tbody>
<tr>
<td>LITERAL</td>
</tr>
<tr>
<td>BUILTIN_FUNC</td>
</tr>
<tr>
<td>JUMP</td>
</tr>
<tr>
<td>PC</td>
</tr>
<tr>
<td>NONE</td>
</tr>
</tbody>
</table>

**Output Types**

<table>
<thead>
<tr>
<th>SIMM13</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETHI/OR</td>
</tr>
<tr>
<td>JUMP</td>
</tr>
<tr>
<td>CALL</td>
</tr>
</tbody>
</table>
Interpreter Operand Fetch

<table>
<thead>
<tr>
<th>Push command</th>
<th>Bytecode Instruction</th>
<th>Literal Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>push 1 00 01</td>
<td>increment vpc to operand</td>
<td>0 0xf81d4</td>
</tr>
<tr>
<td></td>
<td>load operand from bytecode stream</td>
<td>1 0xfa008 “foo”</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>add l5, 1, l5</td>
<td>increment vpc to operand</td>
</tr>
<tr>
<td>ldub [l5], o0</td>
<td>load operand from bytecode stream</td>
</tr>
<tr>
<td>ld [fp+48], o2</td>
<td>get bytecode object addr from C stack</td>
</tr>
<tr>
<td>ld [o2+4c], o1</td>
<td>get literal tbl addr from bytecode obj</td>
</tr>
<tr>
<td>sll o0, 2, o0</td>
<td>compute offset into literal table</td>
</tr>
<tr>
<td>ld [o1+ o0], o1</td>
<td>load from literal table</td>
</tr>
</tbody>
</table>
Operand Specialization

- Array load becomes a constant
- Patch **input**: one-byte integer literal at offset 1
- Patch **output**: `sethi/or` at offset 0
Net Improvement

- Interpreter:
  11 instructions + 8 dispatch

- Catenated:
  6 instructions + 0 dispatch

- **push** is shorter than most, but very common

```
sethi o1, hi(obj_addr)
or o1, lo(obj_addr)
st o1, [l6]
ld [o1], o0
inc o0
st o0, [o1]
```

Final Template for **push**
Evaluation

- Performance
- Ideas
Compilation Time

- Templates are fixed size, fast
- Two catenation passes
  - compute total length
  - memcpy, apply patches (very fast)
- adds 30 - 100% to bytecode compile time
Varying I-Cache Size

- Four hypothetical I-cache sizes
- Simics Full Machine Simulator
- 520 Tcl Benchmarks
- Run both interpreted and catenating VM
Varying I-Cache Size

![Bar Chart]

- I-Cache Size (KB): 32, 128, 1024, infinite
- # of Benchmarks Improved: 45, 54, 66, 82

Percentage scale on the y-axis ranges from 0% to 100%.
Branch Prediction - Catenation

- No dispatch branches
- Virtual branches become native
- Similar CFG for native and virtual program
- BTB knows what to do
- Prediction rate similar to statically compiled code: excellent for many programs
Implementation Retrospective

- Getting templates from interpreter is fun
- Too brittle for portability, research
- Need explicit control over code gen
- Write own code generator, or
- Make compiler scriptable?
Related Work

- Ertl & Gregg, PLDI 2003
  - Efficient Interpreters (Forth, OCaml)
  - Smaller bytecodes, more dispatch overhead
  - Code growth, but little I-cache overflow

- DyC: m88ksim

- Qemu x86 simulator (F. Bellard)

- Many others; see paper
Conclusions

• Many ways to speed up interpreters
• Catenation is a good idea, but like all inlining needs selective application
• Not very applicable to Tcl’s large bytecodes
• Ever-changing micro-architectural issues
Future Work

- Investigate correlation between opcode body size, I-cache misses
- Selective outlining, other adaptation
- Port: another architecture; an efficient VM
- Study benefit of each optimization separately
- Type inference
JIT emitting

- Interpret patches
- A few loads, shifts, adds, and stores
Token Threading in GNU C

#define NEXT goto *(instr_table[*vpc++])

Enum {INST_ADD, INST_PUSH, ...};
char prog [] = {INST_PUSH, 2, INST_PUSH, 3, INST_MUL, ...};
void *instr_table [] = { &INST_ADD, &INST_PUSH, ...};

INST_PUSH:
    /* ... implementation of PUSH ... */
    NEXT;

INST_ADD:
    /* ... implementation of ADD ... */
    NEXT;
Virtual Machines are Everywhere

• Perl, Tcl, Java, Smalltalk. grep?
• Why so popular?
• Software layering strategy
• Portability, Deploy-ability, Manageability
• Very late binding
• Security (e.g., sandbox)
Software layering strategy

- Software getting more complex
- Use expressive higher level languages
- Raise level of abstraction
Problem: Performance

- Interpreters are slow: 1000 – 10 times slower than native code
- One possible solution: JITs
Just-In-Time Compilation

• Compile to native inside VM, at runtime
• But, JITs are complex and non-portable – would be most complex and least portable part of, e.g. Tcl
• Many JIT VMs interpret sometimes
Reducing Dispatch Count

• In addition to reducing cost of each dispatch, we can reduce the *number* of dispatches

• Superinstructions: static, or dynamic, e.g.:

• *Selective Inlining*

  Piumarta & Riccardi, PLDI’98
Switch Dispatch Assembly

for_loop:
  ldub [i0], o0
  fetch opcode

switch:
  cmp o0, 19
  bounds check
  bgu for_loop

  add i0, 1, i0
  increment vpc

  sethi hi(inst_tab), r0
  lookup addr
  or r0, lo(inst_tab), r0
  sll o0, 2, o0
  ld [r0 + o0], o2

  jmp o2
  dispatch

  nop
Push Opcode Implementation

```assembly
add  l6, 4,  l6  ; increment VM stack pointer
add  l5, 1,  l5  ; increment vpc past opcode. Now at operand
ldub [l5],  o0  ; load operand from bytecode stream
ld  [fp + 48],  o2  ; get bytecode object addr from C stack
ld  [o2 + 4c],  o1  ; get literal tbl addr from bytecode obj
sll  o0, 2,  o0  ; compute array offset into literal 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 ref count
inc  o0
st  o0,  [o1]
```

• 11 instructions
Indirect (Token) Threading

push 11
push 7
mul
print
done

Bytecode "asm"

<table>
<thead>
<tr>
<th>Bytecode</th>
<th>Instruction Table</th>
<th>Interpreter Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>push 0x80001000</td>
<td>inst_push: ...</td>
</tr>
<tr>
<td>1</td>
<td>pop 0x80001020</td>
<td>... next</td>
</tr>
<tr>
<td>2</td>
<td>add 0x80001034</td>
<td>inst_pop: ...</td>
</tr>
<tr>
<td>3</td>
<td>sub 0x80001058</td>
<td>... next</td>
</tr>
<tr>
<td>4</td>
<td>mul 0x80001072</td>
<td>inst_add: ...</td>
</tr>
<tr>
<td>5</td>
<td>print 0x8000111c</td>
<td>... next</td>
</tr>
<tr>
<td>6</td>
<td>done 0x80001024</td>
<td>...</td>
</tr>
</tbody>
</table>
Token Threading Example

```c
#define TclGetUInt1AtPtr(p) ((unsigned int) *(p))
#define Tcl_IncrRefCount(objPtr) ++(objPtr)->refCount
#define NEXT goto *jumpTable [*pc];

case INST_PUSH:
    Tcl_Obj *objectPtr;

    objectPtr = codePtr->objArrayPtr [TclGetUInt1AtPtr (pc + 1)];
    *++tosPtr = objectPtr;     /* top of stack */
    Tcl_IncrRefCount (objectPtr);
    pc += 2;
    NEXT;
```
Token Threaded Dispatch

- 8 instructions
- 14 cycles

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sethi</td>
<td>hi(800), o0</td>
</tr>
<tr>
<td>or</td>
<td>o0, 2f0, o0</td>
</tr>
<tr>
<td>ld</td>
<td>[l7 + o0], o1</td>
</tr>
<tr>
<td>ldub</td>
<td>[15], o0</td>
</tr>
<tr>
<td>sll</td>
<td>o0, 2, o0</td>
</tr>
<tr>
<td>ld</td>
<td>[o1 + o0], o0</td>
</tr>
<tr>
<td>jmp</td>
<td>o0</td>
</tr>
<tr>
<td>nop</td>
<td></td>
</tr>
</tbody>
</table>
Direct Threading

- Represent program instructions with address of their interpreter implementations
Direct Threaded Dispatch

- 4 instructions
- 10 cycles

```
ld    r1 = [vpc]
add   vpc = vpc + 4
jmp   *r1
nop
```
Direct Threading in GNU C

#define NEXT goto *(*vpc++)

int prog [] =
    {&&INST_PUSH, 2, &&INST.Push, 3, &&INST_MUL, ...};

INST_PUSH:
    /* … implementation of PUSH … */
    NEXT;

INST_ADD:
    /* … implementation of ADD … */
    NEXT;
Superinstructions

• Note repeated opcode sequence

• Create new synthetic opcode

\[
\text{iload\_bipush\_if\_icmplt}
\]

• takes 3 parms
Copying Native Code

- `inst_push`: ld, st, next
- `inst_mul`: ld, mul, st, next
- `inst_print`: ...

- `inst_push`: ld, st
- `inst_mul`: ld, mul, st
- `inst_print`: ...

- `inst_push_mul`: ld, st, next
- `inst_mul`: ld
- `inst_print`: ...
inst_add Assembly

inst_table:
.word inst_add
.switch table
 .word inst_push
 .word inst_print
 .word inst_done

inst_add:
  ld [i1], o1
  add i1, -4, i1
  arg = *stack_top--
  ld [i1], o0
  add o0, o1, o0
  *stack_top += arg
  st o0, [i1]
  b for_loop
  dispatch
Copying Native Code

uint push_len = &&inst_push_end - &&inst_push_start;
uint mul_len = &&inst_mul_end - &&inst_mul_start;

void *codebuf = malloc (push_len + mul_len + 4);
mmap (codebuf, MAP_EXEC);

memcpy (codebuf, &&inst_push_start, push_len);
memcpy (codebuf + push_len, &&inst_mul_start, mul_len);
/* … memcpy (dispatch code) */
Limitations of Selective Inlining

- Code is not meant to be memcpy’d
- Can’t move function calls, some branches
- Can’t jump into middle of superinstruction
- Can’t jump out of middle (actually you can)
- Thus, only usable at virtual basic block boundaries
- Some dispatch remains
Catenation

- Essentially a template compiler
- Extract templates from interpreter
Catenation - Branches

<table>
<thead>
<tr>
<th>bytecode</th>
<th>native code</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1: inc_var 1</td>
<td>L1: code for inc_var</td>
</tr>
<tr>
<td>push 2</td>
<td>code for push</td>
</tr>
<tr>
<td>cmp</td>
<td>code for cmp</td>
</tr>
<tr>
<td>beq L1</td>
<td>code for beq-test</td>
</tr>
<tr>
<td>...</td>
<td>beq L1</td>
</tr>
</tbody>
</table>

- Virtual branches become native branches
- Emit synthesized code; don’t memcpy
Operand Fetch

• In interpreter, one generic copy of `push` for all virtual instructions, with any operands
• Java, Smalltalk, etc. have `push_1`, `push_2`
• But, only 256 bytecodes

• Catenated code has `separate` copy of push for each instruction

push 1
push 2
inc_var 1

sample code
Threaded Code, Decentralized Dispatch

• Eliminate bounds check by avoiding `switch`
• Make dispatch explicit
• Eliminate extra branch by not using `for`

• James Bell, 1973 CACM
• Charles Moore, 1970, Forth

• Give each instruction its own copy of dispatch
Why Real Work Cycles Decrease

• We do not separately show improvements from branch conversion, vpc elimination, and operand specialization
Why I-cache Improves

• Useful bodies packed tightly in instruction memory (in interpreter, unused bodies pollute I-cache)
Operand Specialization

• **push** not typical; most instructions much longer (for Tcl)
• But, **push** is very common
Micro Architectural Issues

- Operand fetch includes 1 - 3 loads
- Dispatch includes 1 load, 1 indirect jump
- Branch prediction
Branch Prediction

Control Flow Graph - Switch Dispatch

- 85 - 100% mispredictions [Ertl 2003]
Better Branch Prediction

CFG - Threaded Code

- Approx. 60% mispredict

BTB

<table>
<thead>
<tr>
<th>B_1</th>
<th>Last Succ</th>
</tr>
</thead>
<tbody>
<tr>
<td>B_2</td>
<td>Last Succ</td>
</tr>
<tr>
<td>B_3</td>
<td>Last Succ</td>
</tr>
<tr>
<td>B_4</td>
<td>Last Succ</td>
</tr>
<tr>
<td>B_5</td>
<td>Last Succ</td>
</tr>
<tr>
<td>B_6</td>
<td>Last Succ</td>
</tr>
<tr>
<td>B_7</td>
<td>Last Succ</td>
</tr>
<tr>
<td>B_8</td>
<td>Last Succ</td>
</tr>
<tr>
<td>B_9</td>
<td>Last Succ</td>
</tr>
<tr>
<td>B_{10}</td>
<td>Last Succ</td>
</tr>
</tbody>
</table>
How Catenation Works

• Scan templates for patterns at VM startup
  – Operand specialization points
  – vpc rematerialization points
  – pc-relative instruction fixups
• Cache results in a “compiled” form
• Adds 4 ms to startup time
From Interpreter to Templates

• Programming effort:
  – Decompose interpreter into 1 instruction case per file
  – Replace operand fetch code with magic numbers
From Interpreter to Templates 2

• Software build time (**make**):
  – compile C to assembly (PIC)
  – selectively *de-optimize* assembly
  – Conventional link
```c
#ifdef INTERPRET
    #define MAGIC_OP1_U1_LITERAL codePtr->objArray[GetUInt1AtPtr(pc + 1)]
    #define PC_OP(x) pc ## x
    #define NEXT_INSTR break
#elseif COMPILE
    #define MAGIC_OP1_U1_LITERAL (Tcl_Obj *) 0x7bc5c5c1
    #define NEXT_INSTR goto *jump_range_table[*pc].start
    #define PC_OP(x) /* unnecessary */
#endif case INST_PUSH1:
    Tcl_Obj *objectPtr;
    objectPtr = MAGIC_OP1_U1_LITERAL;
    *++tosPtr = objectPtr; /* top of stack */
    Tcl_IncrRefCount(objectPtr);
    PC_OP(=, 2);
    NEXT_INSTR; /* dispatch */
```

**Magic Numbers**
Dynamic Execution Frequency
Instruction Body Length

Body length (ntv insns).