Alternative Dispatch Techniques for the Tcl VM

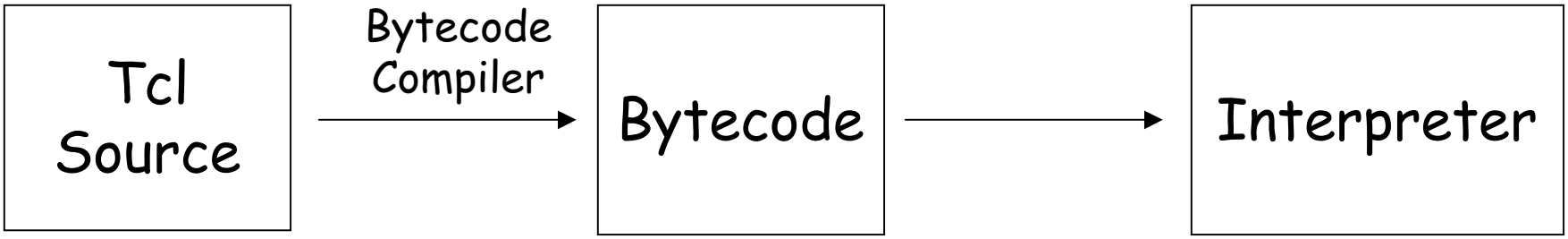
Benjamin Vitale

Mathew Zaleski

Outline

- How the VM Interprets Bytecode
- Dispatch speed on pipelined CPUs
- The Context Problem
- Context Threading
- Results

Running a Tcl Program

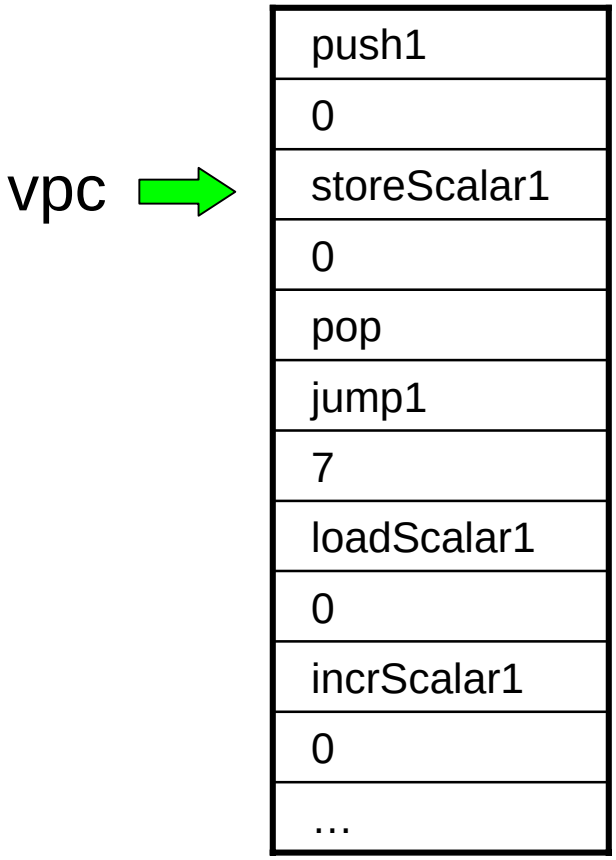


Compiling to Bytecode

```
# 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
```

Interpreter



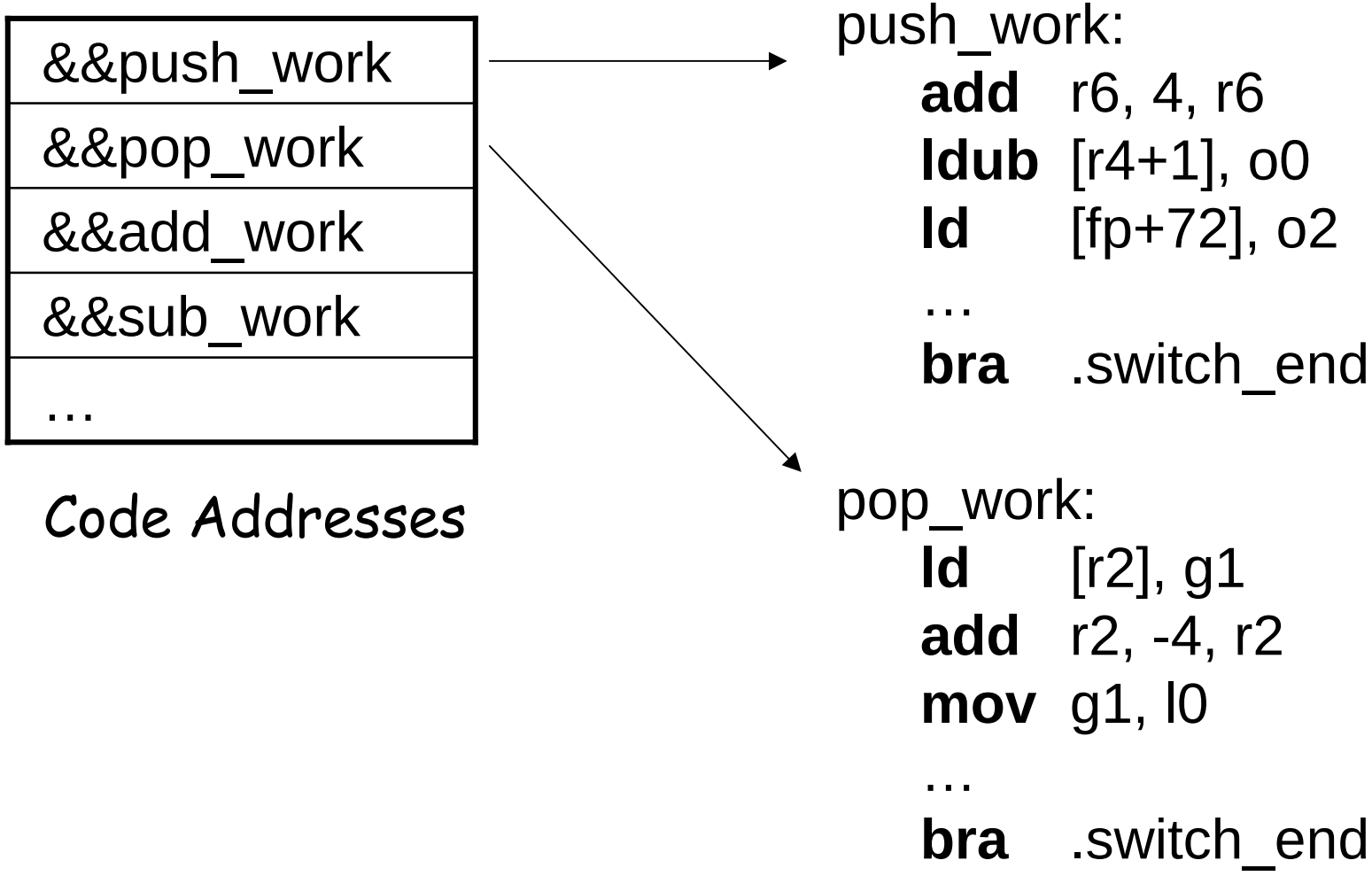
Bytecode Representation

```
for (;;) {
    opcode = *vpc;
    switch (opcode) {
    case PUSH1:
        // real work…
        vpc += 2;
        break;
    case POP:
        …
```

Performance Problem

- Interpreting bytecode is faster than interpreting source
- But still slow
- One problem for some VMs is **high dispatch overhead**
- How does **switch()** dispatch work?

How C compiles `switch()`



Executing `switch()`

```
ldub    opc = [vpc]              // Opcode load (unaligned)

cmp     opc, max_opc             // Bounds check (useless)
bg      switch_default

set     r5 = switch_table        // Table lookup (avoidable)
mul     r1 = r4 * 4
ld      [r5 + r1], r1

jmp     r1 + r5                  // Indirectly jump to work
```

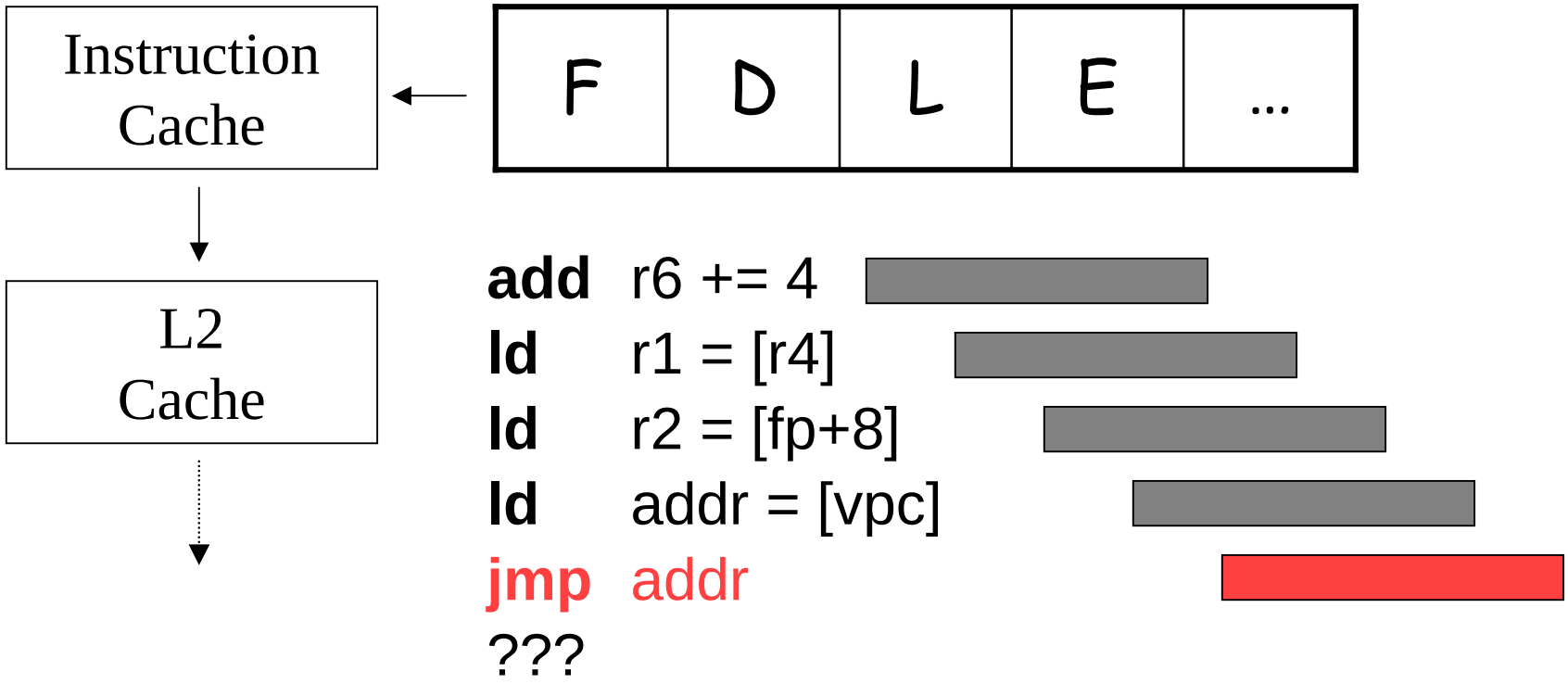
- **17 cycles**

Direct Threading

```
ld    address = [vpc]     // Opcode load (aligned)
jmp   address             // Indirect jump
```

- 12 Cycles
- portably expressed in Gnu C
  - we should consider this for Tcl
- 2 insns in 12 cycles. What is CPU doing?

CPU Pipeline



- Keeping pipeline full requires pre-fetching. But which instructions?

Branch Target Predictor

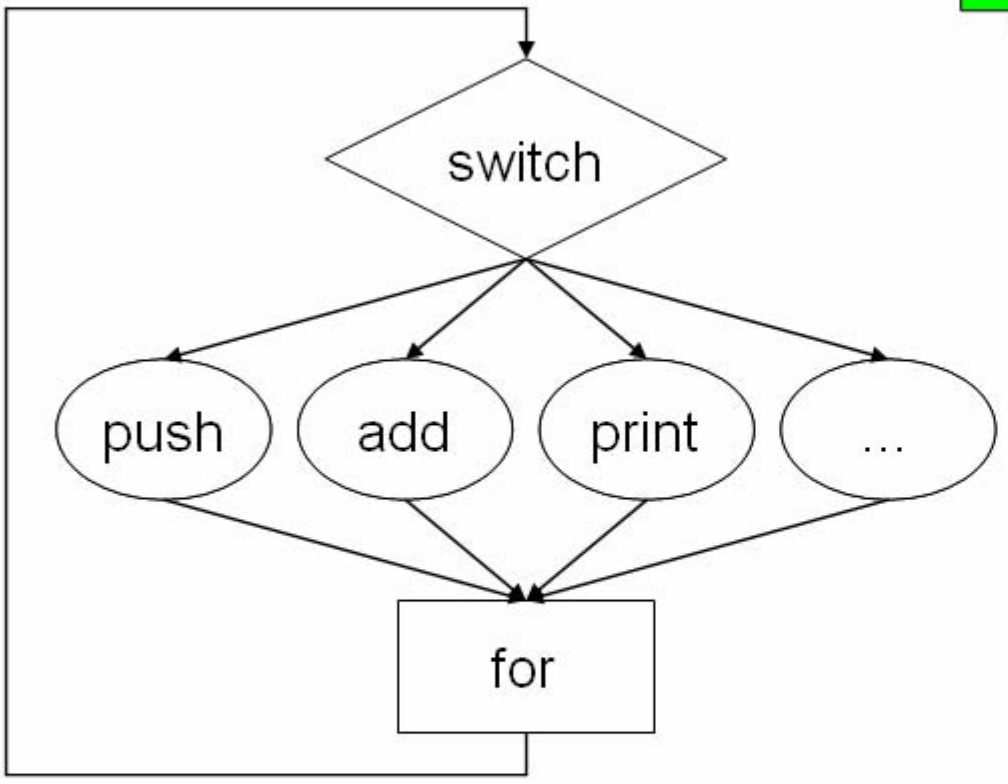
```
 0   add   r6 += 4
 4   ld    addr = [r1]
 8   cmp   r6, 12
12   bg    6
16   jmp   addr
20   ld    r2 = [r3]
24   sll   r2 = r2, 2
28   jmp   r2
```

pc_{jmp}	pc_{target}
16	42
28	1000

Branch Target Address Cache

- Predict branch target from past behavior

Context Problem Example



Interpreter

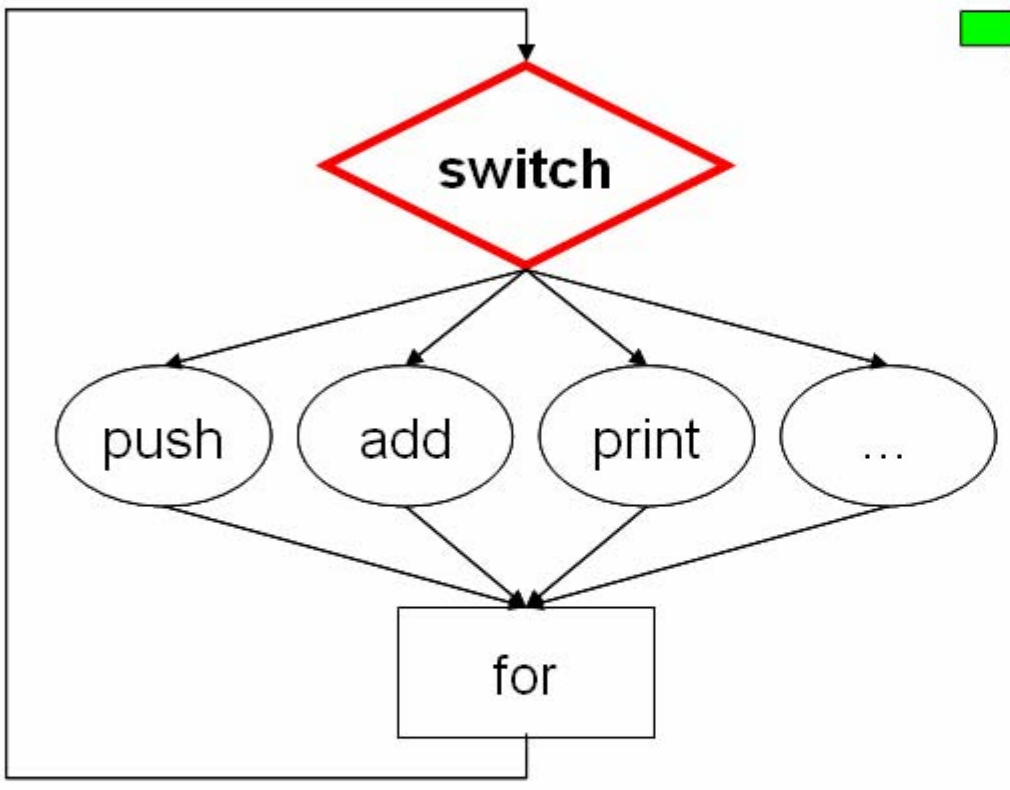
```
push 2
push 3
add
print
```

Bytecode Program

pc_{jmp}	target
...	

BTAC

Context Problem Example



Interpreter

```
push 2
push 3
add
print
```

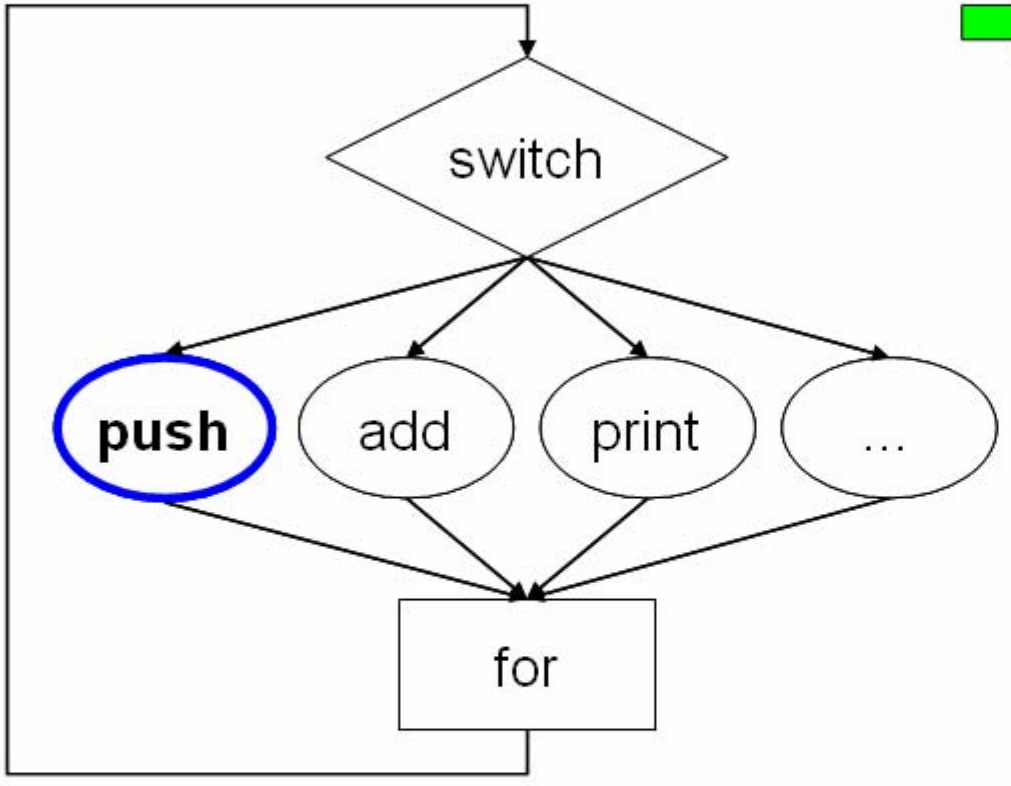
Bytecode Program

pc_{jmp}	target
...	

?

BTAC

Context Problem Example



Interpreter

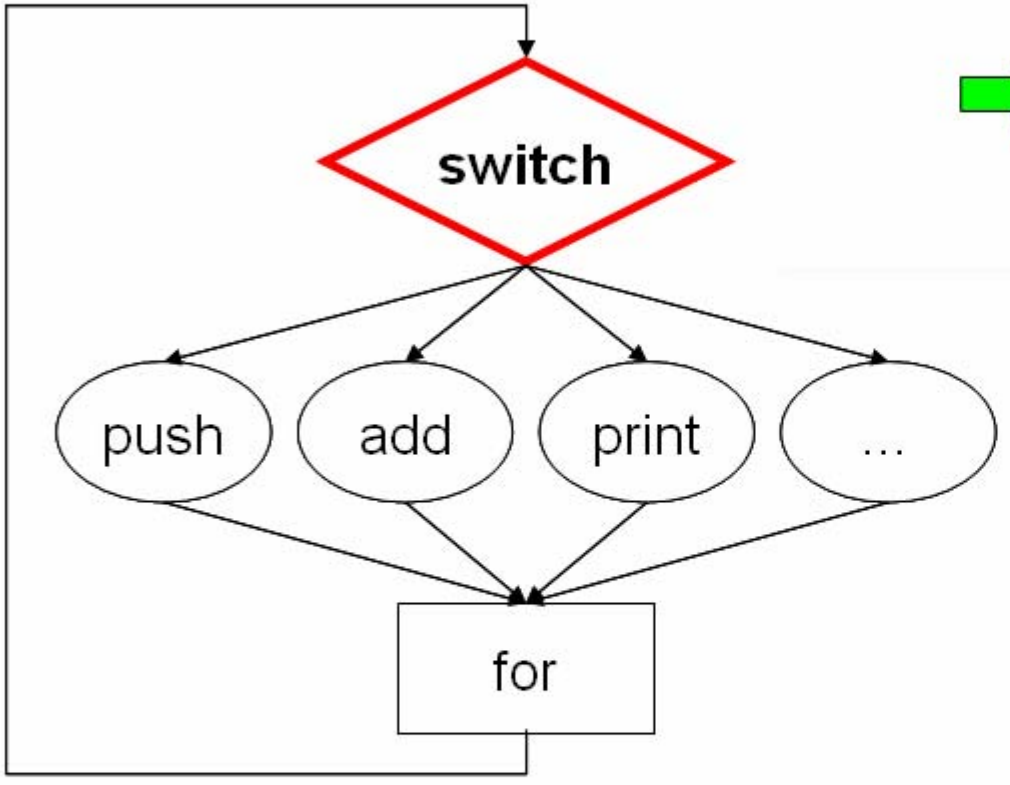
push 2
push 3
add
print

Bytecode Program

pc_{jmp}	target
switch	push
...	

BTAC

Context Problem Example



Interpreter

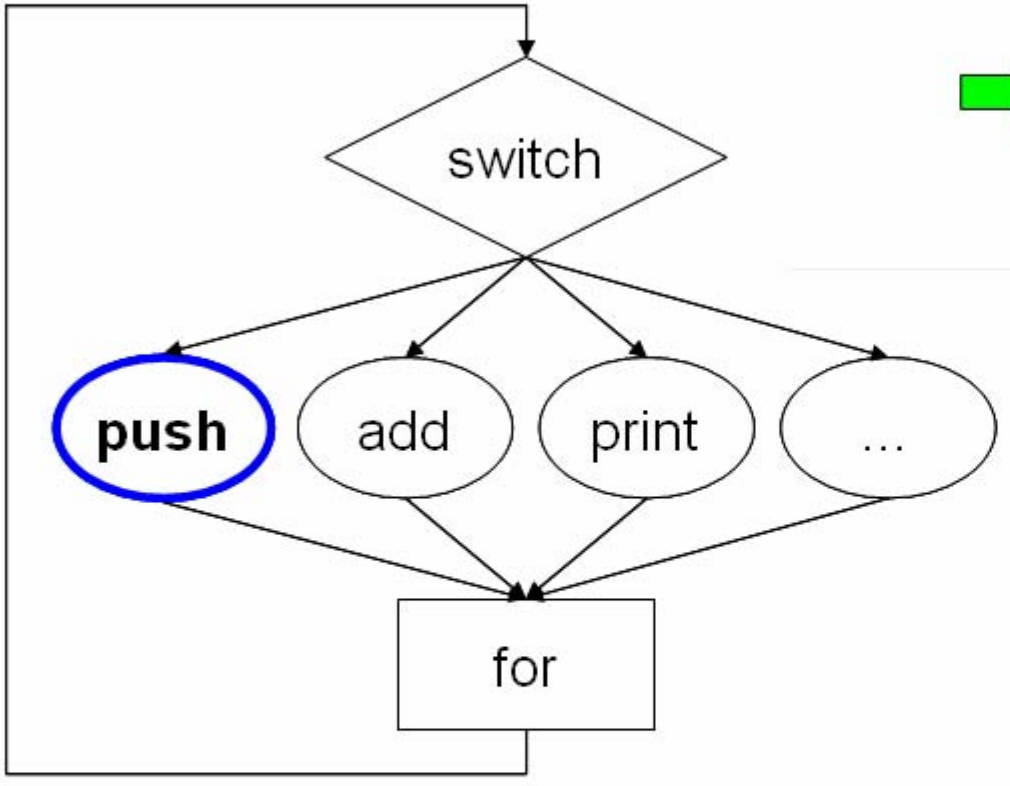
```
push  2
push  3
add
print
```

Bytecode Program

pc_{jmp}	target
switch	push
...	

BTAC

Context Problem Example



Interpreter

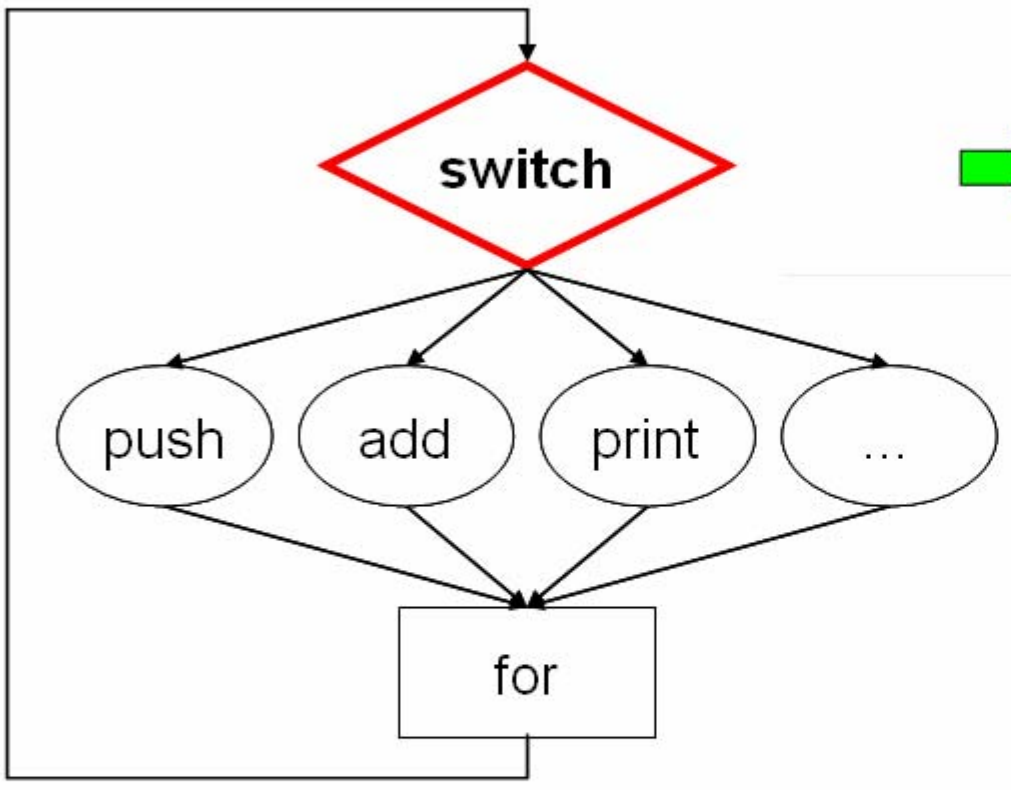
```
push 2
push 3
add
print
```

Bytecode Program

pc_{jmp}	target
switch	push
...	

BTAC

Context Problem Example



Interpreter

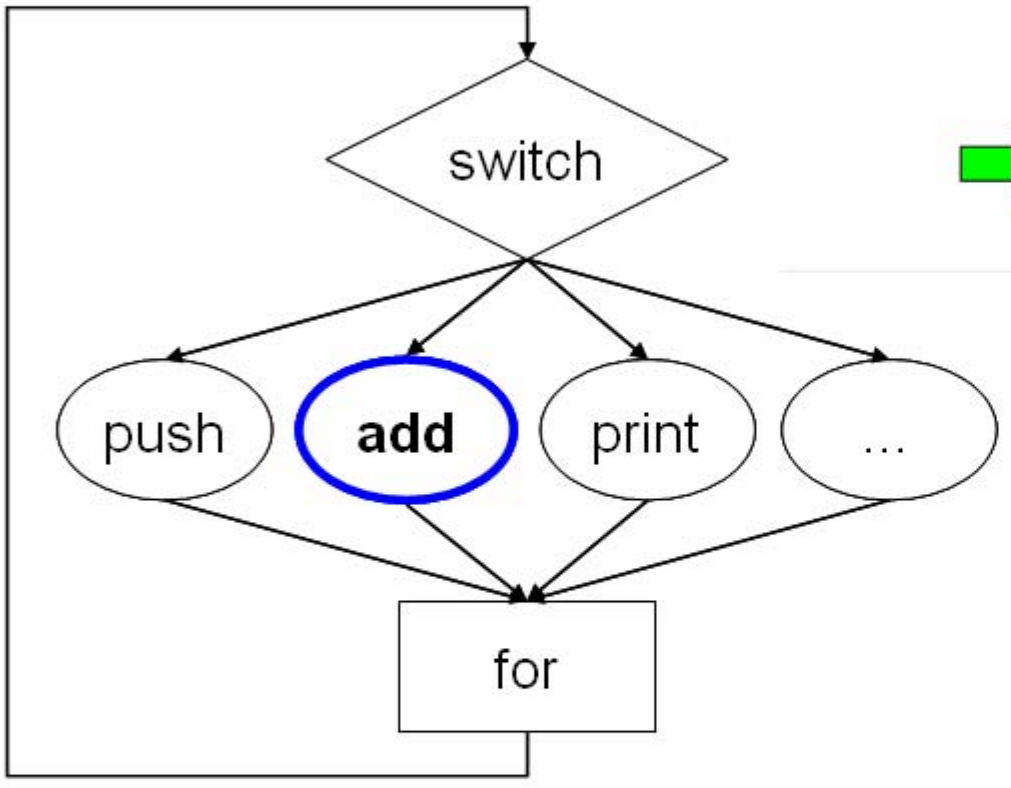
```
push 2
push 3
add
print
```

Bytecode Program

pc_{jmp}	target
switch	push
...	

BTAC

Context Problem Example




```
push 2
push 3
add
print
```

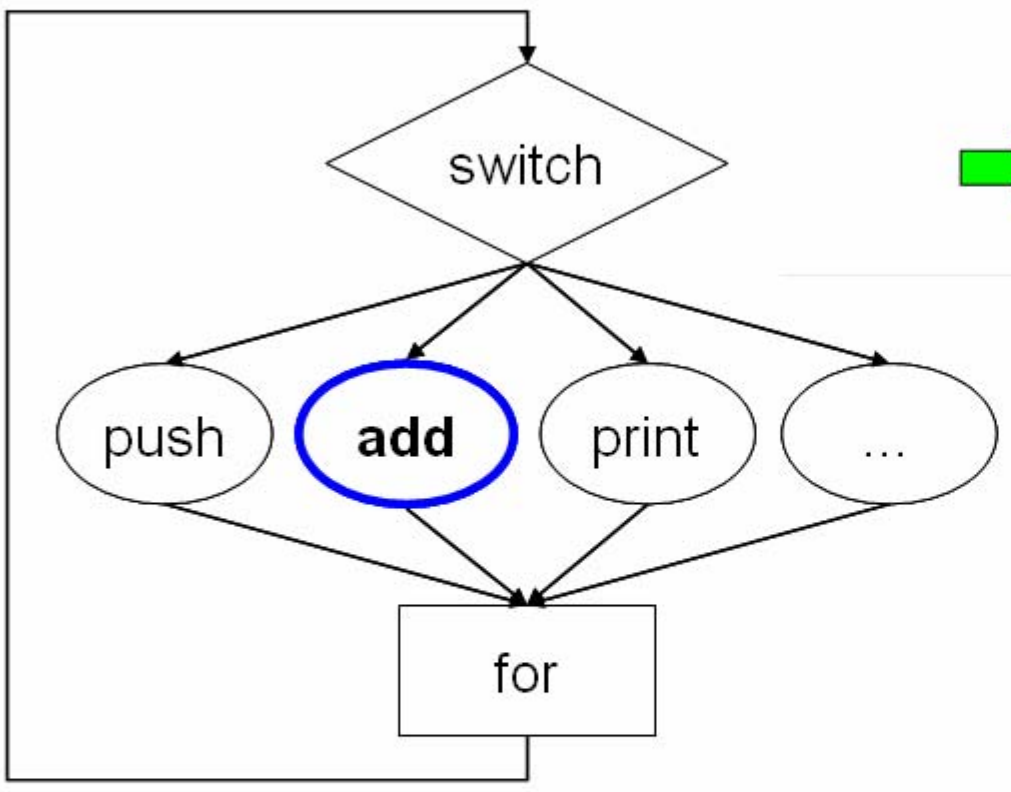
pc_{jmp}	target
switch	push
...	

Interpreter

Bytecode Program

BTAC

Context Problem Example




```
push 2
push 3
add
print
```

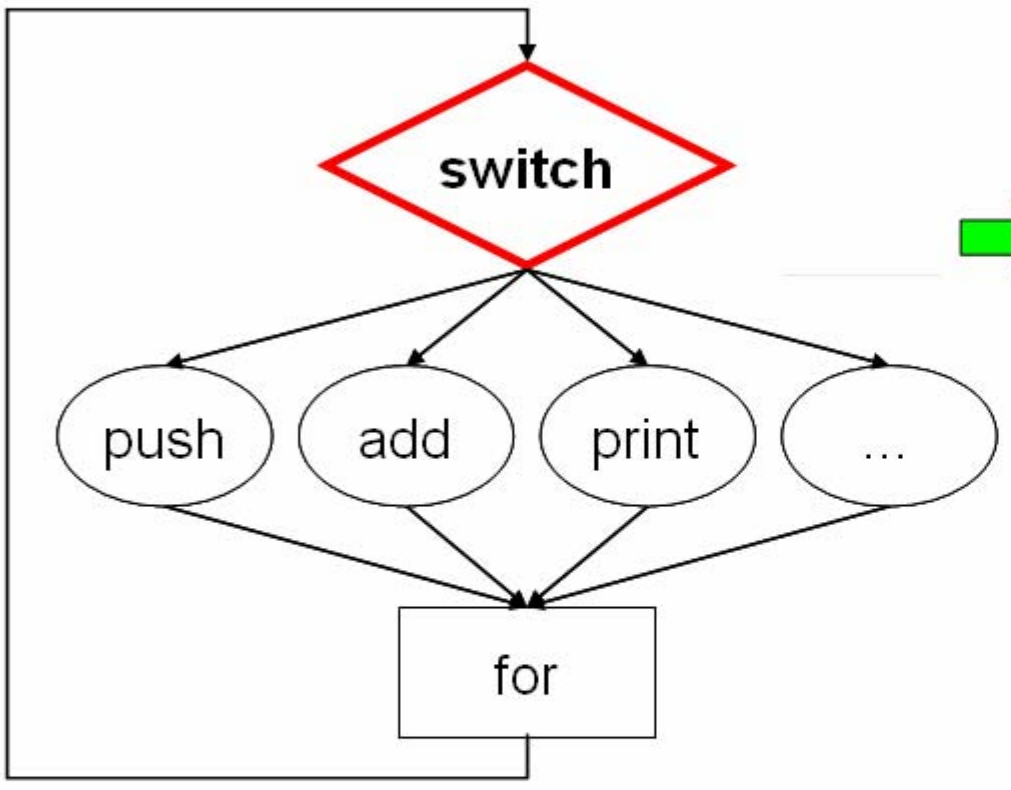
pc_{jmp}	target
switch	add
...	

Interpreter

Bytecode Program

BTAC

Context Problem Example



Interpreter

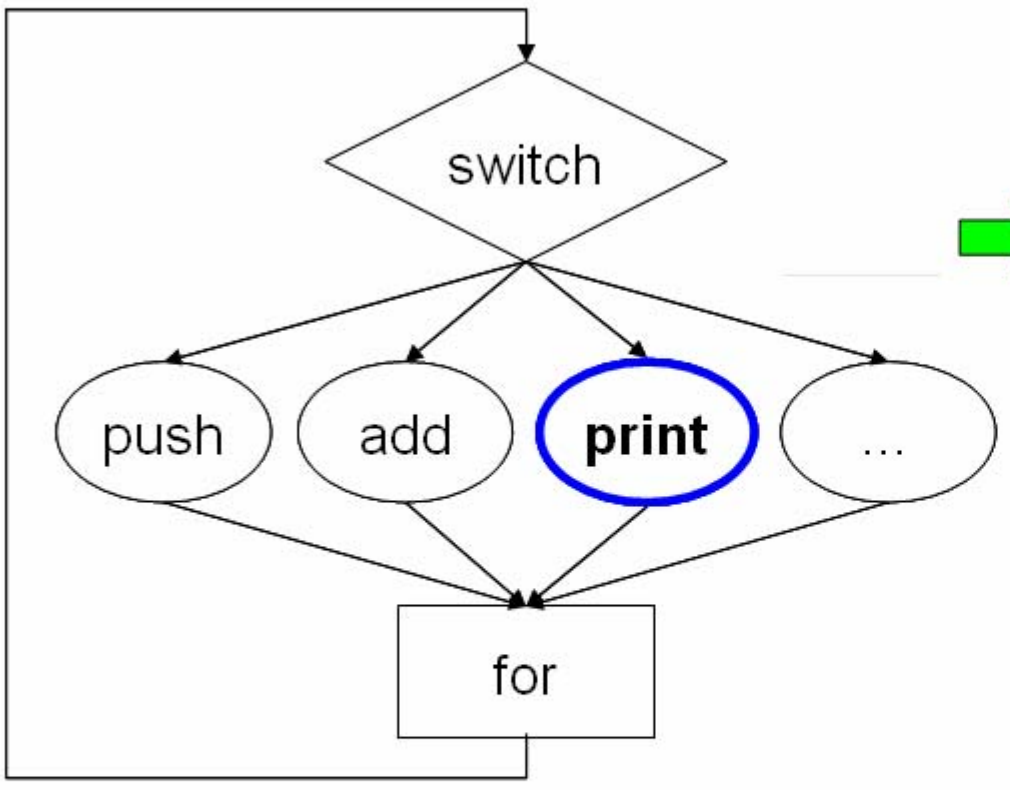
```
push 2
push 3
add
print
```

Bytecode Program

pc_{jmp}	target
switch	add
...	

BTAC

Context Problem Example




```
push 2
push 3
add
print
```

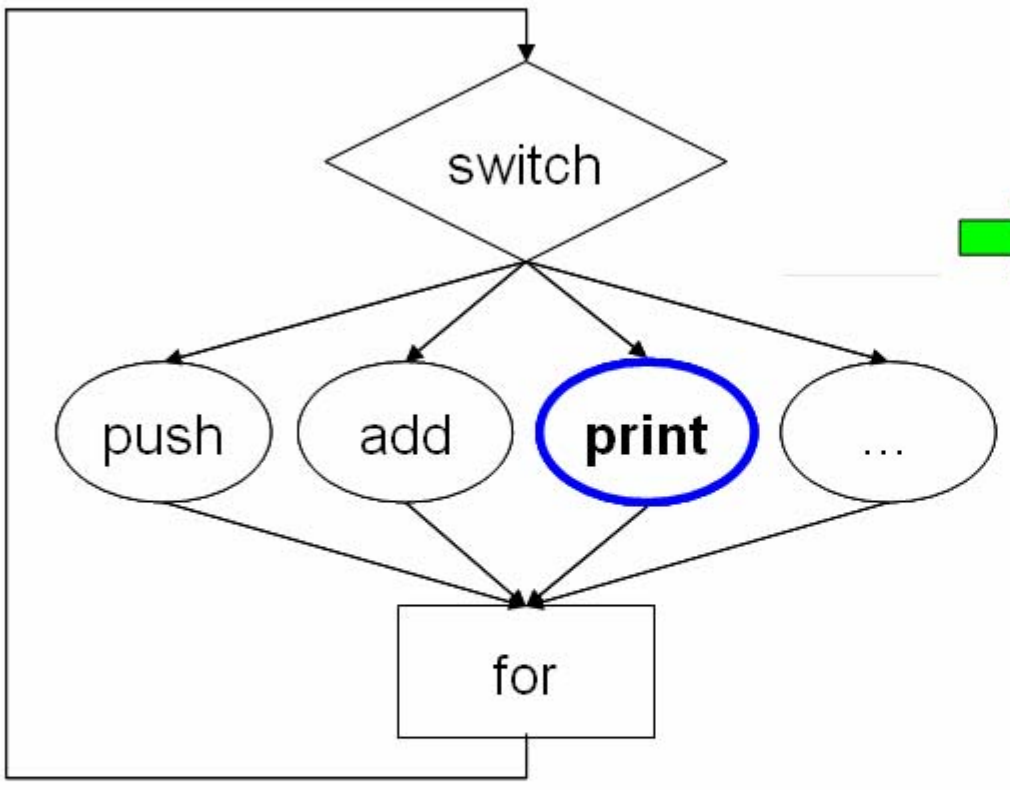
pc_{jmp}	target
switch	add
...	

Interpreter

Bytecode Program

BTAC

Context Problem Example




```
push 2
push 3
add
print
```

pc_{jmp}	target
switch	print
...	

Interpreter

Bytecode Program

BTAC

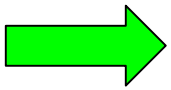
Context Problem

- Hardware is using PC for prediction
  - Only one branch means one BTAC entry
- VM is using vpc
  - **branch depends on vpc, has many targets**
  - [Ertl03] 85% mispredicts, costs 10+ cycles
- How can we avoid misprediction?

Subroutine Threading

- Old idea. Great for modern CPUs
- Correlates native pc with virtual pc
- 6 cycle dispatch

```
0  push1          0
2  storeScalar1   0
4  pop
5  loadScalar1    0
7  incrScalar1    0
9  pop
```



```
call  push1
call  storeScalar1
call  pop
call  loadScalar1
call  incrScalar1
call  pop
```

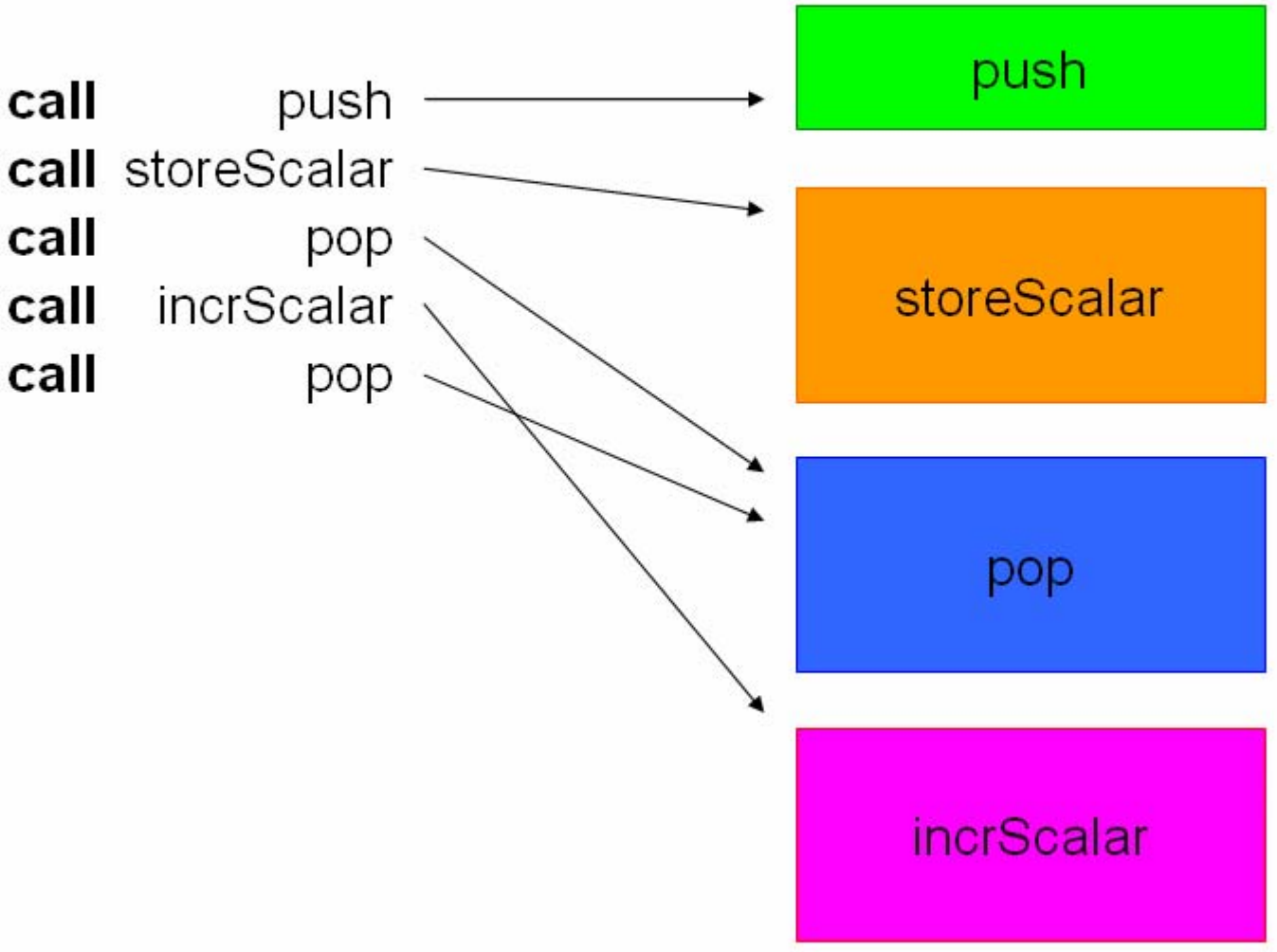
Bytecode

Native Code ("CTT")

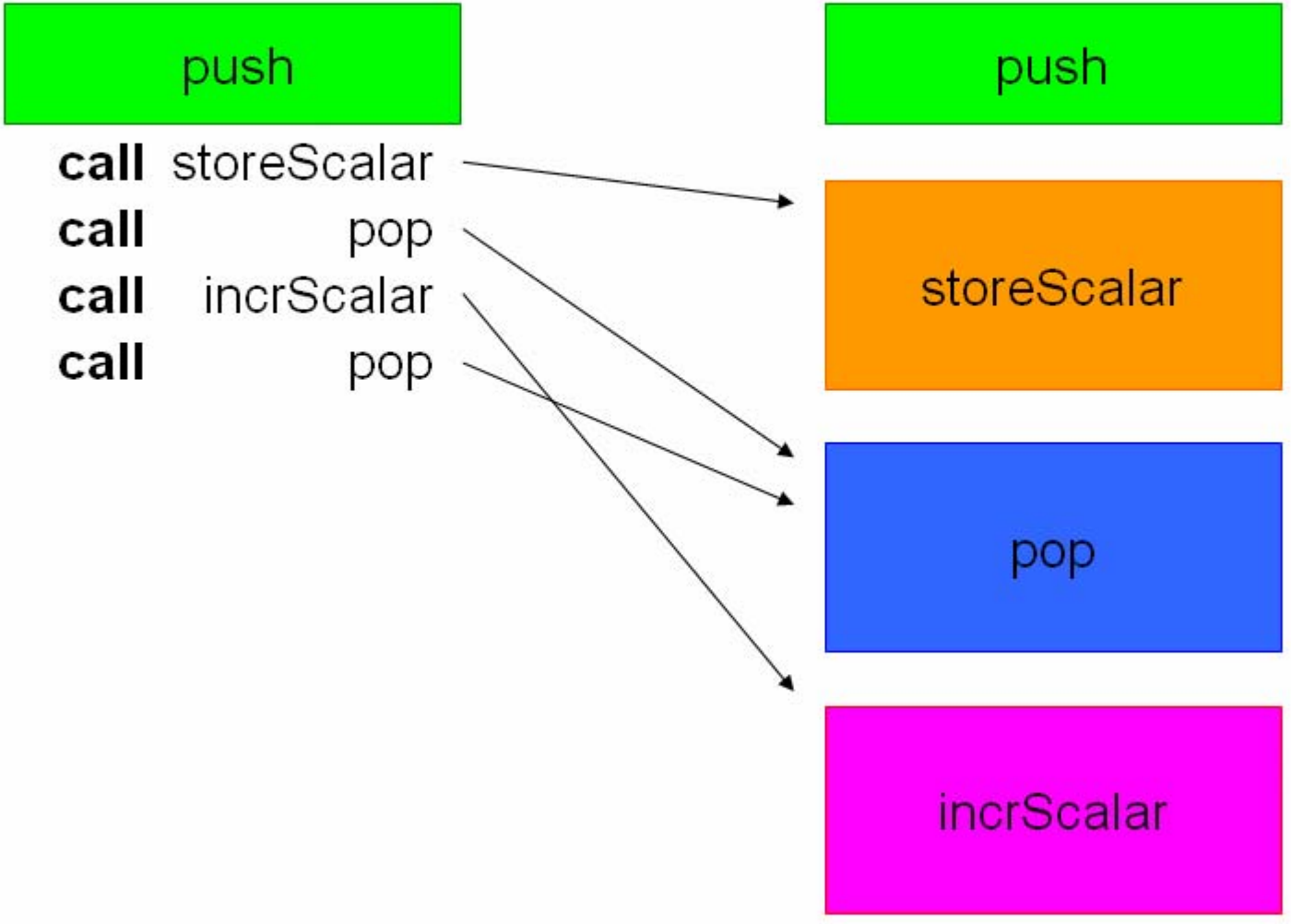
Context Threading

- Our implementation of subroutine threading
- *CGO'05*
- Keep bytecode around for operands, etc.
- Optimizations exploit CTT's flexibility

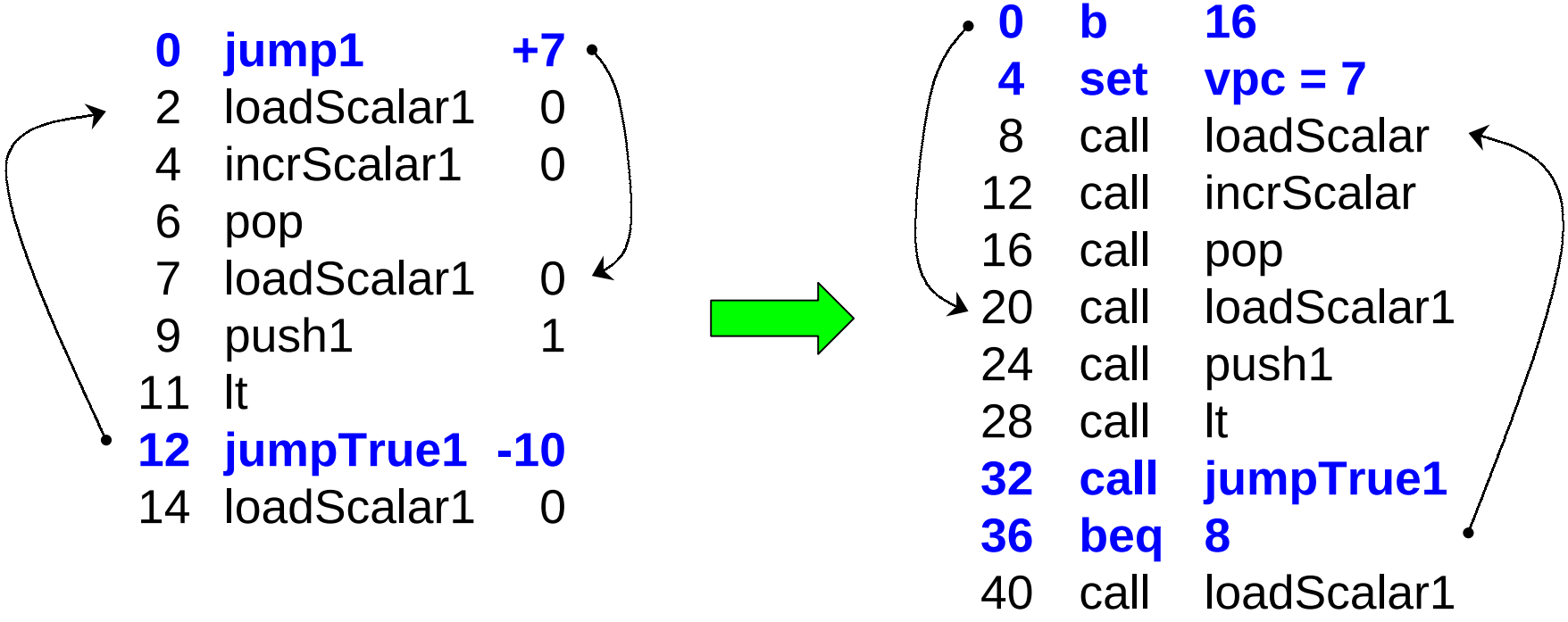
Inlining Small Opcodes



Inlining Small Opcodes



Virtual Branches become Native



- jump becomes two native instructions
- jumpTrue uses native branch, but also calls

Conditional Branch Peephole Opt

- We can eliminate the call to jumpTrue
  - Profile: what precedes cond. branches?
    - gt, lt, tryConvertNumeric, foreachStep4
  - **move the branch code into CTT and** gt
- Tcl 8.5 has a similar optimization, but bigger payoff for native
- **Loops go faster**

Cond. Branch Peephole Opt Demo

```
call   gt
call   jumpTrue
beq    target_n
```

gt:

```
c = do_compare...
o = new_bool (c)
push (o)
vpc++
return
```

jump_true: *91 insns*

```
o = pop ()
coerce_bool (o)
if (o.bool)
    vpc = target_v
else
    vpc = fall_thru_v
asm ("cmp o.bool, 0")
return
```

Cond. Branch Peephole Opt Demo

```
call   gt
call   jumpTrue
beq    target_n
```

gt:

```
c = do_compare...
o = new_bool (c)
push (o)
vpc++
return
```

jump_true: *91 insns*

```
o = pop ()
coerce_bool (o)
if (o.bool)
    vpc = target_v
else
    vpc = fall_thru_v
asm ("cmp o.bool, 0")
return
```

Cond. Branch Peephole Opt Demo

```
call   gt
call   jumpTrue
beq    target_n
```

gt:

```
c = do_compare...
o = new_bool (c)
push (o)
vpc++
return
```

jump_true: *91 insns*

```
o = pop ()
coerce_bool (o)
if (o.bool)
    vpc = target_v
else
    vpc = fall_thru_v
asm ("cmp o.bool, 0")
return
```

```
call   gt
call   jumpTrue
beq    target_n
```

Cond. Branch Peephole Opt Demo

```
call   gt
call   jumpTrue
beq    target_n
```

```
gt:
    c = do_compare...
    o = new_bool (c)
    push (o)
    vpc++
    return

jump_true:        91 insns
    o = pop ()
    coerce_bool (o)
    if (o.bool)
        vpc = target_v
    else
        vpc = fall_thru_v
    asm ("cmp o.bool, 0")
    return
```

```
call   gt_jump

beq    target_n
```

Cond. Branch Peephole Opt Demo

```
call   gt
call   jumpTrue
beq    target_n
```

```
gt:
    c = do_compare...
    o = new_bool (c)
    push (o)
    vpc++
    return

jump_true:          91 insns
    o = pop ()
    coerce_bool (o)
    if (o.bool)
        vpc = target_v
    else
        vpc = fall_thru_v
    asm ("cmp o.bool, 0")
    return
```

```
call   gt_jump
set    vpc = target_v
beq    target_n
set    vpc = fall_thru_v
```

Cond. Branch Peephole Opt Demo

```
call   gt
call   jumpTrue
beq    target_n
```

gt:

```
c = do_compare...
o = new_bool (c)
push (o)
vpc++
return
```

jump_true: *91 insns*

```
o = pop ()
coerce_bool (o)
if (o.bool)
    vpc = target_v
else
    vpc = fall_thru_v
asm ("cmp o.bool, 0")
return
```

```
call   gt_jump
set    vpc = target_v
beq    target_n
set    vpc = fall_thru_v
```

gt_jump:

```
c = do_compare...

vpc++
return
```

Cond. Branch Peephole Opt Demo

```
call   gt
call   jumpTrue
beq    target_n
```

gt:

```
c = do_compare...
o = new_bool (c)
push (o)
vpc++
return
```

jump_true: *91 insns*

```
o = pop ()
coerce_bool (o)
if (o.bool)
    vpc = target_v
else
    vpc = fall_thru_v
asm ("cmp o.bool, 0")
return
```

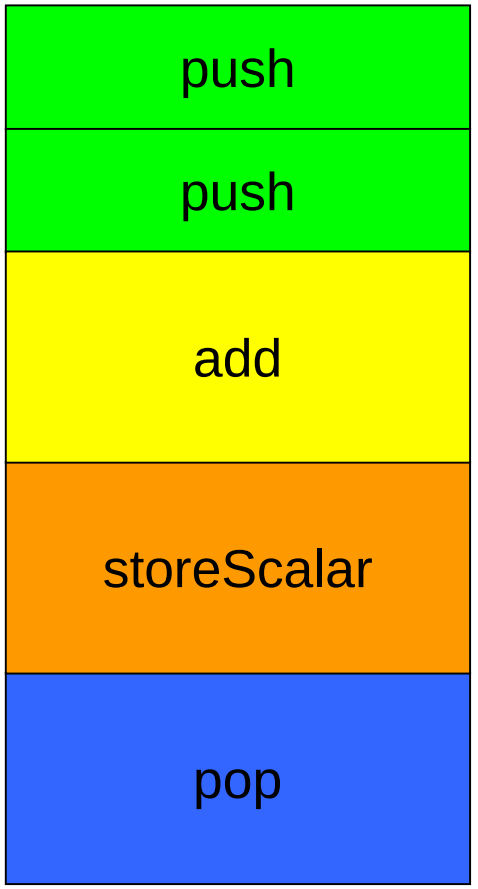
```
call   gt_jump
set    vpc = target_v
beq    target_n
set    vpc = fall_thru_v
```

gt_jump:

```
c = do_compare...
asm ("cmp o.bool, 0")
vpc++
return
```

Catenation

- IVME '04
- Inline *everything*
  - Specialize operands
  - Eliminate `vpc`
- Complicated
- 0 cycle dispatch



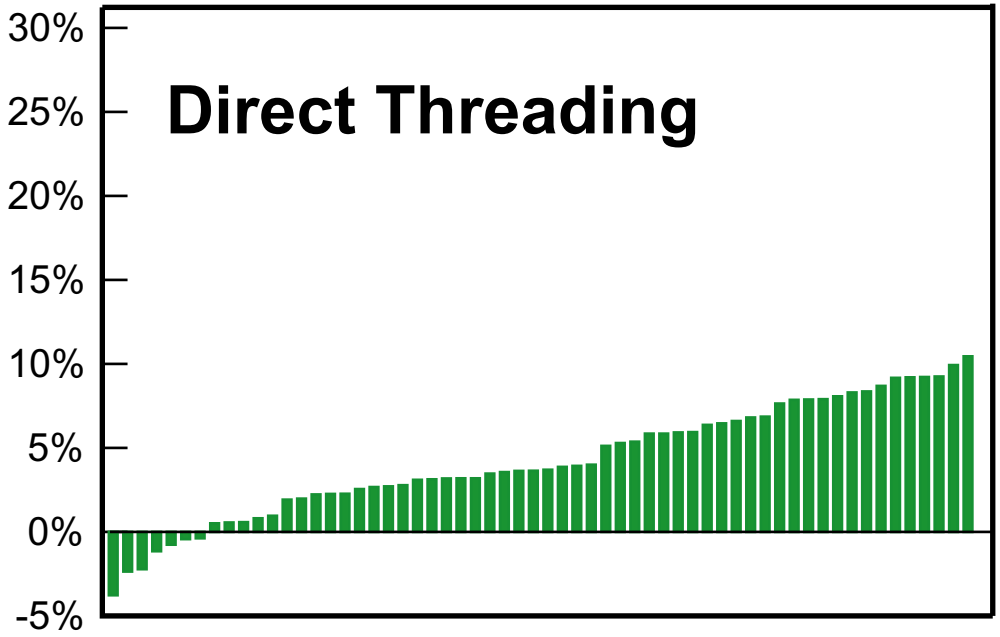
Results

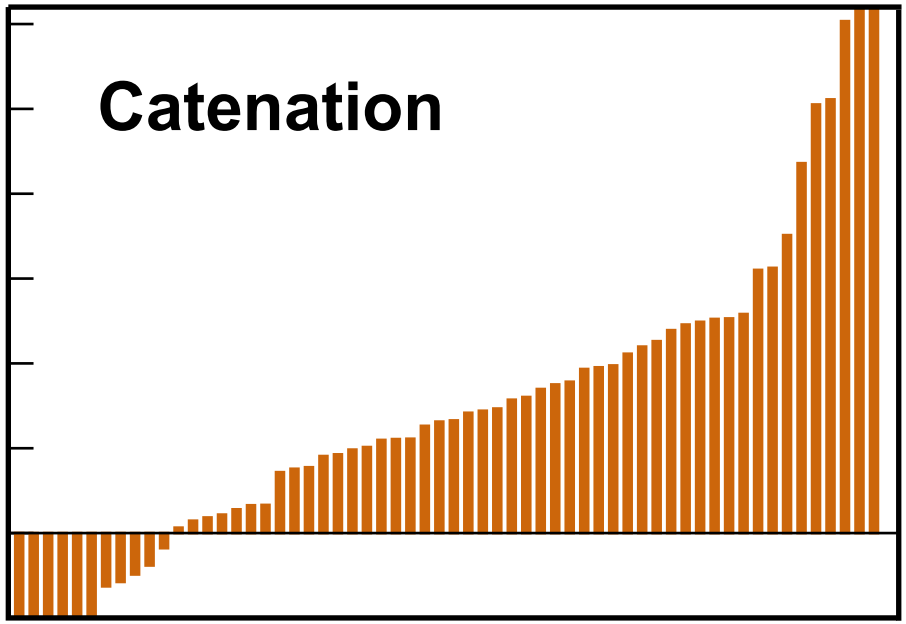
- Tclbench
  - microbenchmarks, only 12 with more than 100,000 dispatches
  - de-facto standard
  - focus on 60 with > 10,000 dispatches
- UltraSPARC III
- Use switch interpreter as baseline

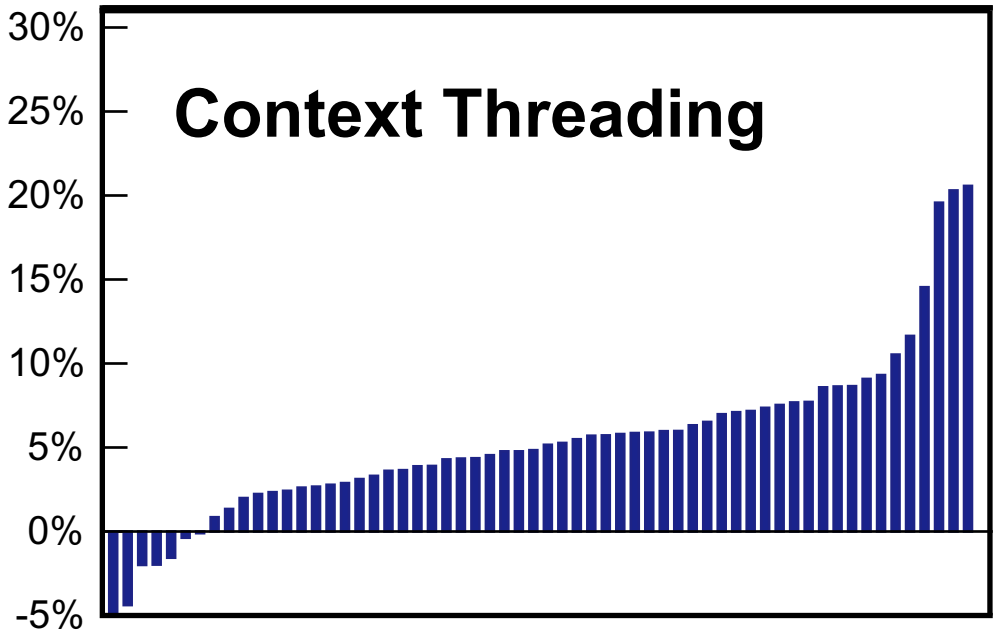
Performance Summary

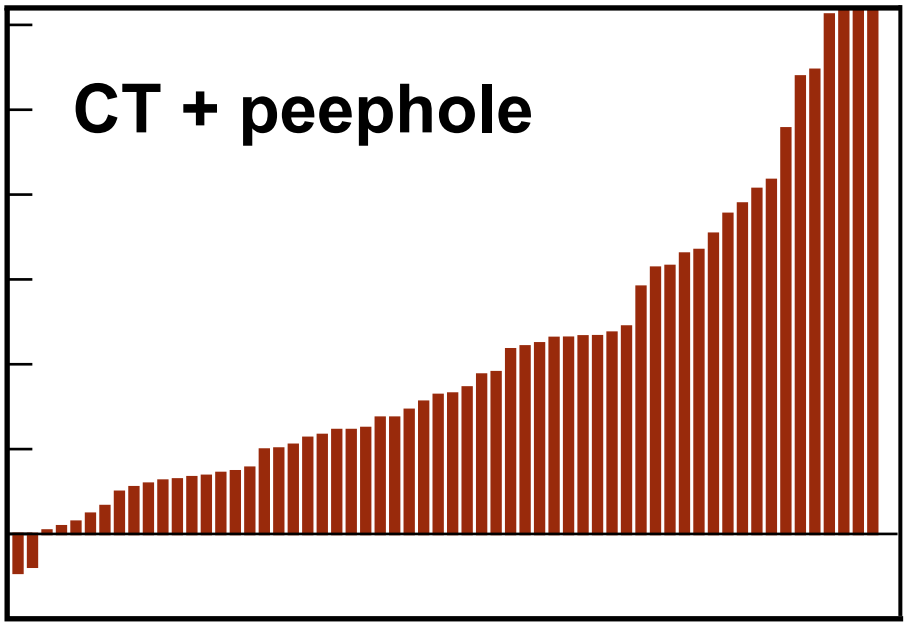
Dispatch type	Geo. mean speedup	Number of benchmarks improved
Direct Threading	4.3%	88%
Catenation	4.0	73
Context Threading	5.4	88
CT + peephole	**12.0**	**97**

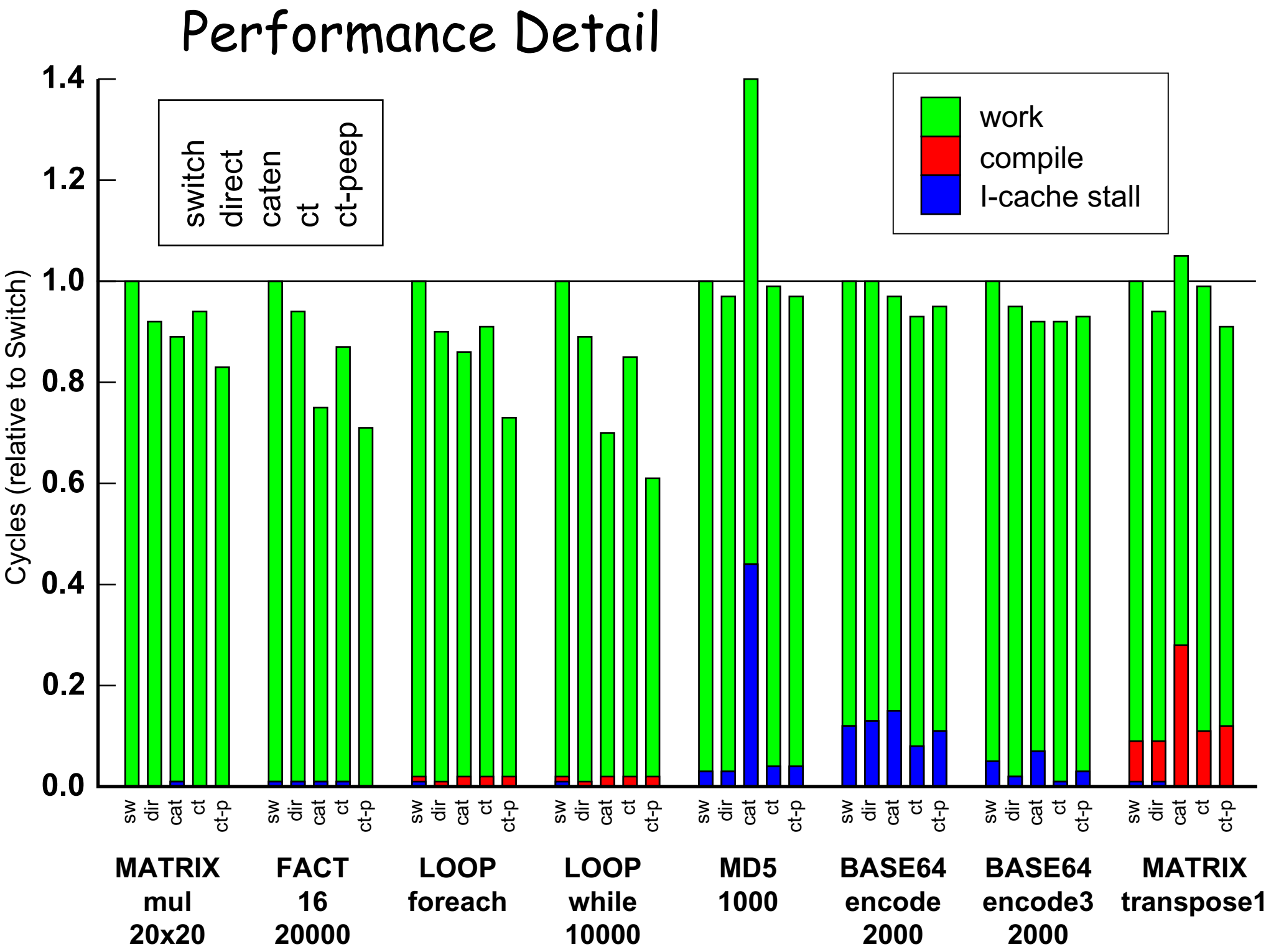
Tclbench Speedup versus Switch



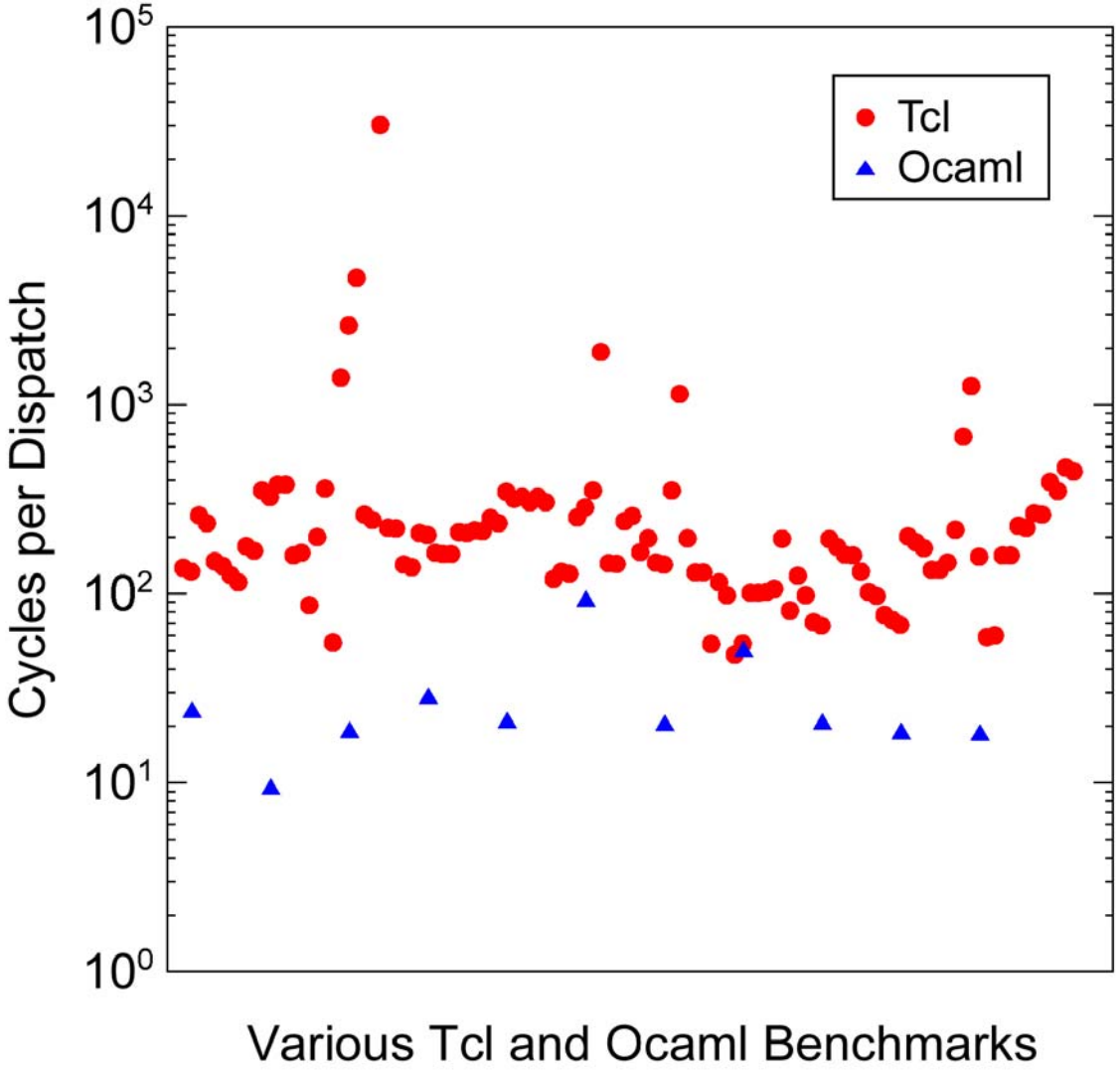







Performance Detail
switch
direct
caten
ct
ct-peep
work
compile
I-cache stall
Cycles (relative to Switch)
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
sw
dir
cat
ct
ct-p
MATRIX mul 20x20
FACT 16 20000
LOOP foreach
LOOP while 10000
MD5 1000
BASE64 encode 2000
BASE64 encode3 2000
MATRIX transpose1

Tcl Opcodes are Big



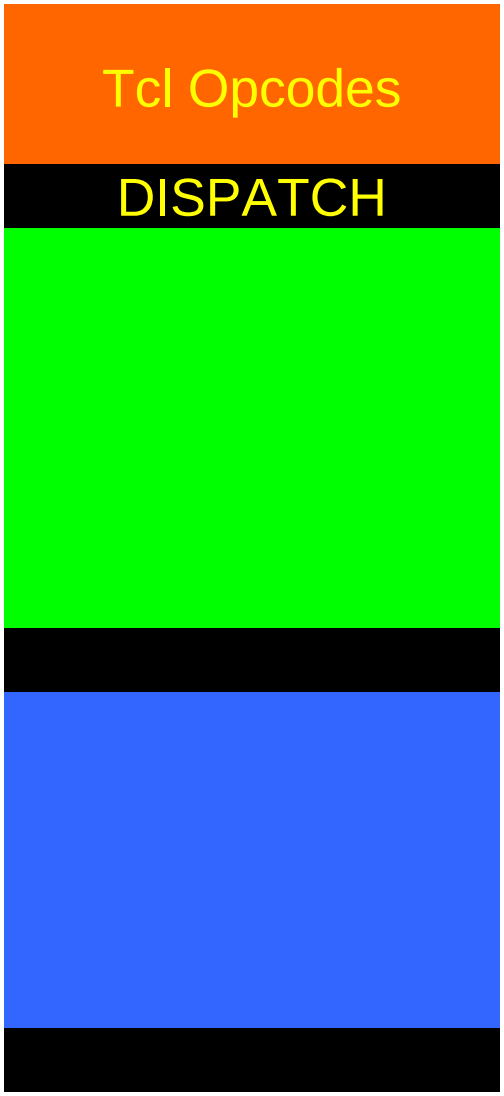
Java	25%
Ocaml	37%
Tcl	5%

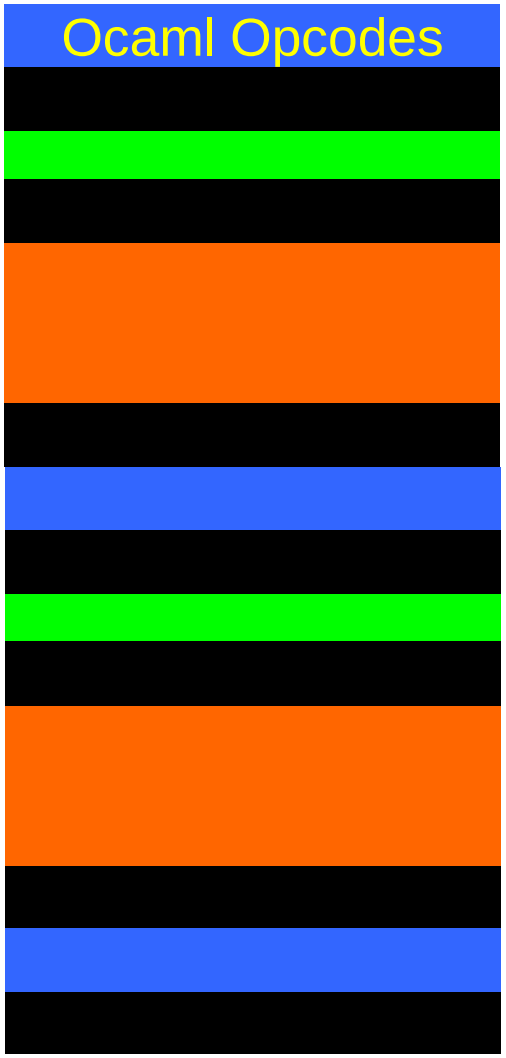
Context Threading Speedup

Conclusions and Future Work

- Context Threading is simple & effective
  - fast dispatch (not Tcl's problem)
  - facilitates optimization
  - inline more opcodes, port to x86, PowerPC
- 12% speedup trivial: Tcl 10x slower than C
  - micro opcodes and a real JIT

Low Dispatch Overhead





Branch prediction on Sparc

- Ultra 1 had "NFA" *in I-cache*
- UltraSPARC III
  - What kind of branch target predictor?
  - "prepare-to-branch" instruction?
  - Consider two virtual programs, on the next slide:

Jekyll and Hyde Programs

```
start:  Vop1
        Vop1
        Vop1
        Vop1
        Vop1
        Vop1
        Vop1
        Vop1
        Vop1
        goto    start
```

Predictable

```
start:  Vop1
        Vop2
        Vop1
        Vop3
        Vop1
        Vop4
        Vop1
        Vop5
        Vop1
        goto    start
```

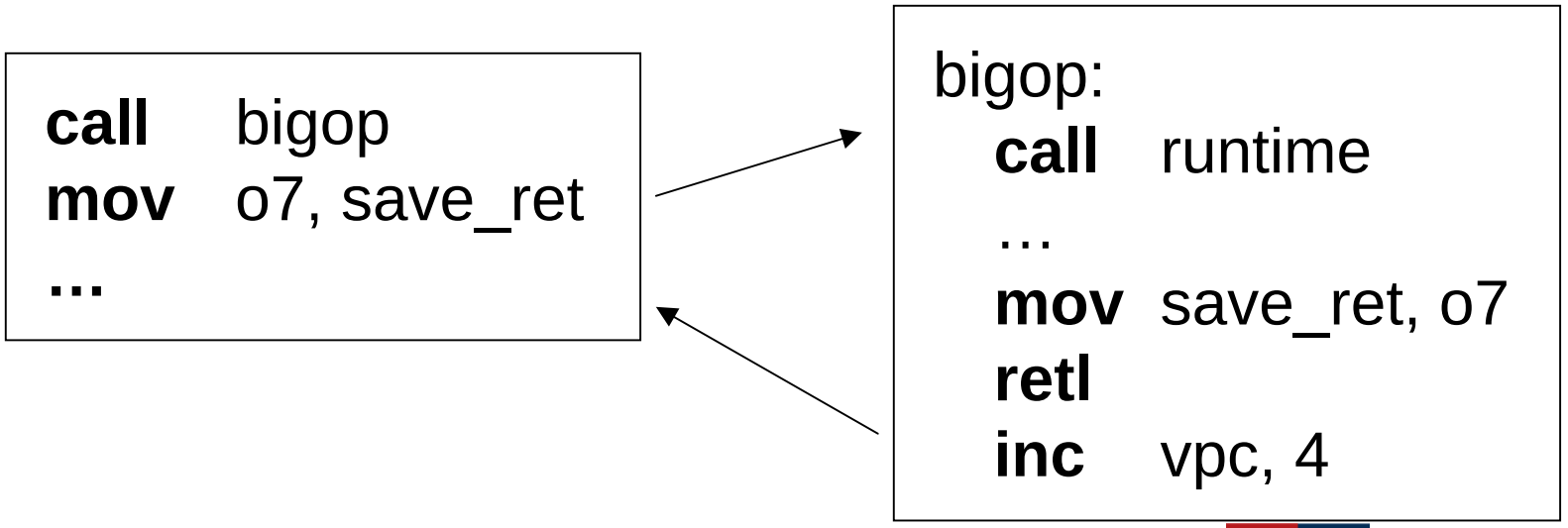
Unpredictable

Mispred vs. predict

Dispatch Type	UltraSPARC III	Pentium 4
switch	17.3	19.2
indirect mispred	14.2	18.6
indirect predict	14.2	11.8
direct **mispred**	**11.2**	**18.7**
direct **predict**	**11.2**	**11.3**
subroutine	6.3	8.4

CT, Tcl, Sparc

- Branch Delay Slot
- Big Tcl bodies nearly all contain calls
  - Calls clobber link register (o7)
  - We save link register in a reserved reg



Compilation Time

- We include compile time in every iteration
- Tclbench amortizes

Dispatch Type	Native Compile time relative to ByteCode
Direct Threading	6%
Catenation	44
Context Threading	35
CT + peephole	38

- Varies significantly across benchmarks