Optimizing Tcl Bytecode

Donal Fellows

University of Manchester / Tcl Core Team

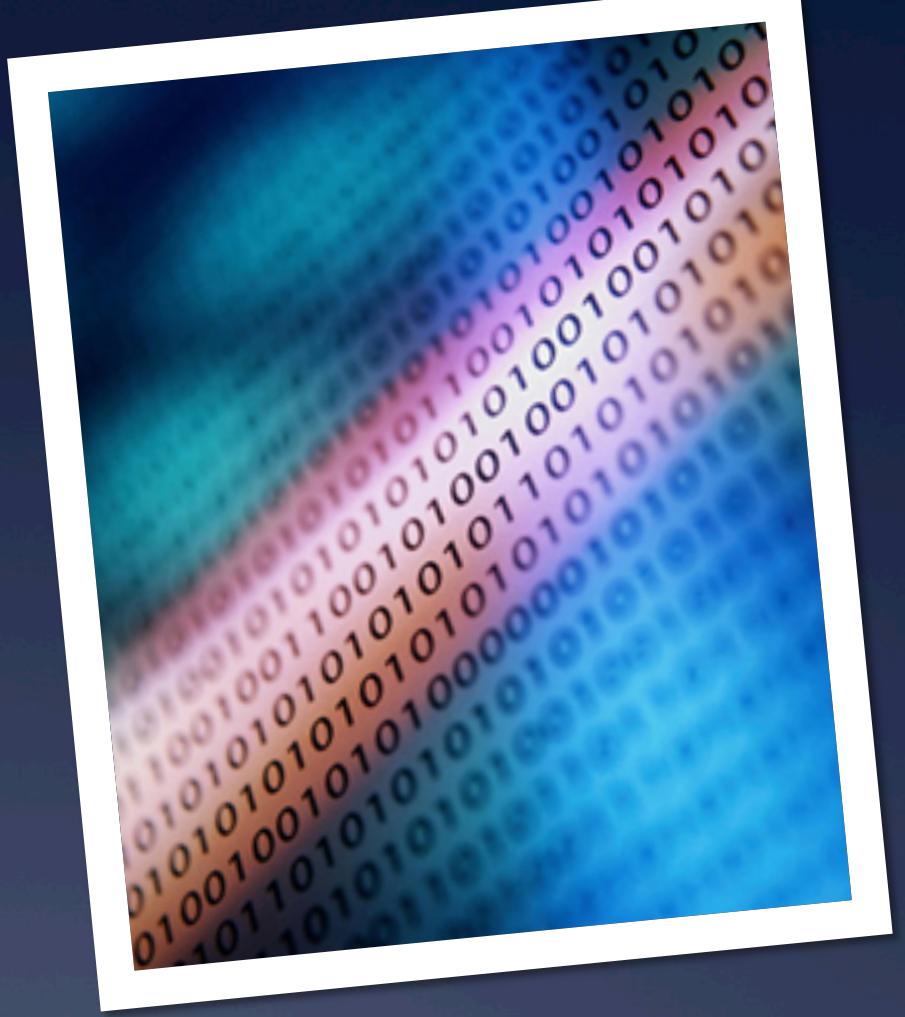
donal.k.fellows@manchester.ac.uk

Outline

1. A refresher on Tcl ***Bytecode***
2. Improving compilation ***Coverage***
3. Improving bytecode ***Generation***
4. A script-readable bytecode ***Disassembler***
5. Towards a true bytecode ***Optimizer***
6. Measured effects on ***Performance***
7. Some future ***Directions***

A refresher on Tcl

Bytecode



Tcl Evaluation Strategy

- Code stored as script (string)
- When required, bytecode interpretation added
  - Stored in `Tcl_Obj` internal representation
- Bytecode evaluated in stack-based engine
  - *Example:* `set c [expr {$a + $b}]`

```
loadScalar1 %v0          # var "a"
loadScalar1 %v1          # var "b"
add
storeScalar1 %v2         # var "c"
pop
```

Looking at Bytecode

- `tcl::unsupported::disassemble`
  - Introduced in Tcl 8.5
    - Same functionality as was achieved in earlier versions by setting `tcl_traceCompile` global
  - Compiles what it is told, if necessary
  - Disassembles the bytecode
    - But not if done by TDK compiler
  - Returns a *human-readable* representation

Disassembly Example

```
% tcl::unsupported::disassemble script {puts "a-$b-c"}
ByteCode 0x0x4e210, refCt 1, epoch 3, interp 0x0x31c10 (epoch 3)
  Source "puts \"a $b c\""
  Cmds 1, src 13, inst 14, litObjs 4, aux 0, stkDepth 4, code/src 0.00
  Commands 1:
    1: pc 0-12, src 0-12
  Command 1: "puts \"a $b c\""
    (0) push1 0     # "puts"
    (2) push1 1     # "a-"
    (4) push1 2     # "b"
    (6) loadScalarStk
    (7) push1 3     # "-c"
    (9) concat1 3
    (11) invokeStk1 2
    (13) done
```

What's Wrong with Bytecode?

* Variable length instructions
  * Many common opcodes come in multiple sizes
* Funky encoding for various lengths
  * Command metadata might as well be read-only!
* Very hard to improve overall
  * Can extend with new opcodes
  * Can compile individual commands better
  * Global optimizations much more challenging

Improving compilation

Coverage

Improving Coverage

* Tcl assembler showed potential
  * `tcl::unsupported::assemble`
* *In theory,* bytecode compiled commands are easier to optimize
  * Can prove safety theorems about them
  * Uncompiled commands are hard
    * Just push arguments and `invokeStk`; no semantics
  * Fully-bytecoded procedures can support more analysis
* To get benefit, needed to increase fraction of compiled commands

Which to tackle?

1. **Prioritize** by requirement for code we want to go fast
   * As little overhead in inner loops as possible
2. **Prioritize** by how common
   * Little benefit to tackling very rare commands
3. **Filter** by how possible
   * Command compilers are non-trivial
4. **Filter** by how fixed in function
   * Bytecode locks in implementation strategy

Methodology

- Identify which commands used in key inner loops
  - Study samples from various performance discussions
    - comp.lang.tcl, Wiki, tcl-core, private emails
- Identify which commands used to generate literals
  - Not just `expr` and `subst`!
  - Official `return -level 0` was known, but non-obvious
    - `lappend x [if {$y} {set y} else {return -level 0 "no"}]`

Methodology

* Identify commands with subcommands ("ensembles")
  * Collect list of all *literal* subcommands used in packages in ActiveTcl Teapot repository
  * Ignore subcommand names from a variable
  * Collate/sort by frequency
  * Manually filter for actual subcommands

*
```
find $TEAPOTDIR -type f -print0 | xargs -0 cat
  | grep --binary-files=text -w $CMD
  | sed "s/.*$CMD *\\([a-z]*\\).*/\\1/"
  | sort | uniq -c | sort -n
```

Subcommand Frequencies

string		dict		namespace		array	
1	string totitle	1	dict keys	3	namespace forget	37	array size
2	string replace	8	dict values	6	namespace inscope	479	array get
8	string trimleft			7	namespace parent	1085	array names
33	string trimright	1	dict with	17	namespace children		
34	string repeat	2	dict unset	50	namespace exists	56	array exists
145	string toupper	3	dict lappend	77	namespace delete	191	array unset
147	string trim	8	dict for	130	namespace import	2511	array set
248	string tolower	15	dict merge	153	namespace origin		
424	string is	18	dict incr	269	namespace ensemble		
		22	dict create	757	namespace export		
28	string last	28	dict append	2681	namespace eval		
245	string index	34	dict exists				
569	string map	297	dict get	30	namespace qualifier		
674	string first	347	dict set	56	namespace which		
898	string match			116	namespace code		
892	string range			132	namespace upvar		
1100	string length			206	namespace tail		
2129	string equal			272	namespace current		
5971	string compare						

Commands with New Compilers

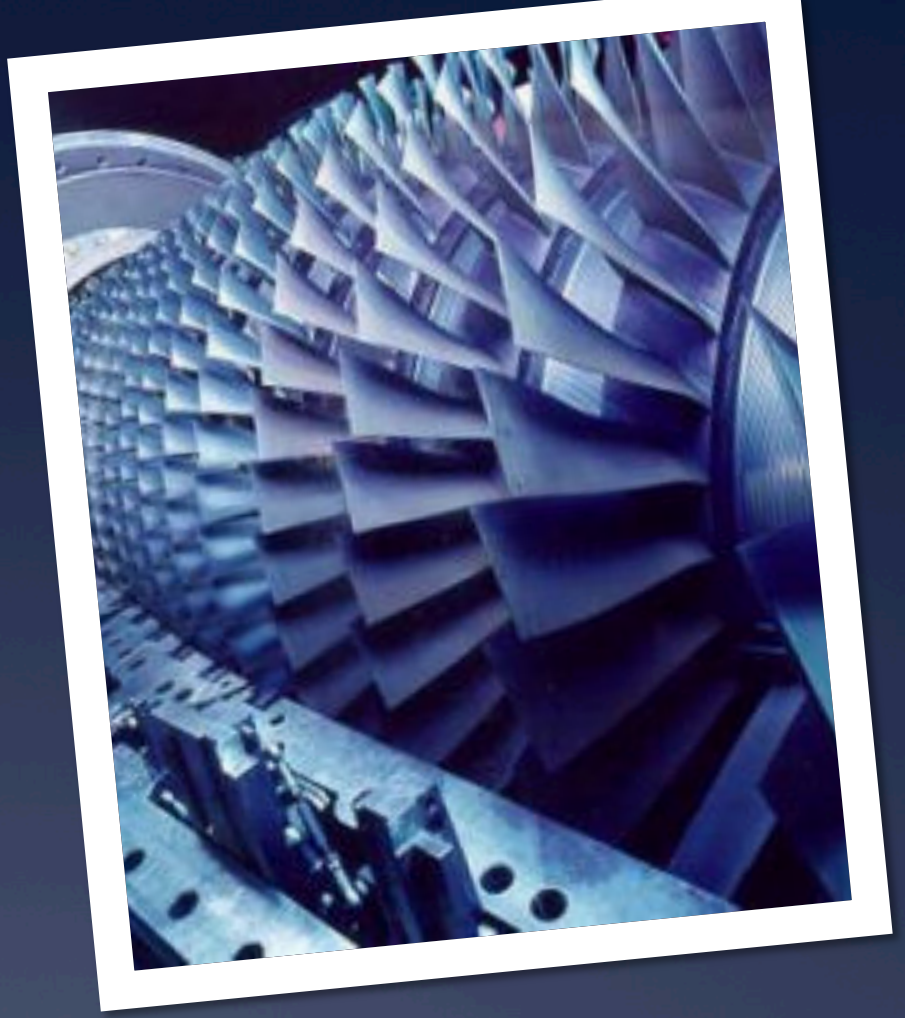
* `array`
  * array exists
  * array set
  * array unset
* `dict`
  * dict create
  * dict merge
* `format`
  * Simple cases only
* `info`
  * info commands
  * info coroutine
  * info level
  * info object class
  * info object isa object
  * info object namespace
* `namespace`
  * namespace code
  * namespace current
  * namespace qualifiers
  * namespace tail
  * namespace which
* `regsub`
  * Simple cases only
* `self`
  * self namespace
  * self object
* `string`
  * string first
  * string last
  * string map
    * Simple cases only
  * string range
* `tailcall`
* `yield`

Future Compiled Commands?

* **Minor**
  * *low impact*
  * *low difficulty*
  * concat
  * eval
  * namespace origin
  * string trim
  * string trimleft
  * string trimright
  * string tolower
  * string toupper
* **Major**
  * *high impact*
  * *high difficulty*
  * array get
  * array names
  * namespace eval
  * next
  * string is
  * uplevel
  * yieldto

Improving bytecode

Generation



Improving Generation: "list concat" via expansion

* Making `list {*}$foo {*}$bar` efficient
  * Now a sort of "lconcat" (for all combinations of arguments)
* Compare old and new versions

Old

```
(0)  expandStart
(1)  push1 0              # "list"
(3)  loadScalar1 %v0      # var "foo"
(5)  expandStkTop 2
(10) loadScalar1 %v1      # var "bar"
(12) expandStkTop 3
(17) invokeExpanded
```

New

```
(0) loadScalar1 %v0    # var "foo"
(2) loadScalar1 %v1    # var "bar"
(4) listConcat
```

Improving Generation: Ensembles

- Bind core ensembles to their implementations
  - Apply basic syntax checks
    - Number of arguments
  - Replace ensemble call with direct call to correct implementation command if possible
  - Otherwise, use special ensemble dispatch
    - Half the mechanism…
- Not for user-defined ensembles
  - Would be very bad for Snit!

```
% disassemble script {info body foo}
[...]
  (0) push1 0    # "::tcl::info::body"
  (2) push1 1    # "foo"
  (4) invokeStk1 2
  (6) done

% disassemble script {string is space x}
[...]
  (0) push1 0    # "string"
  (2) push1 1    # "is"
  (4) push1 2    # "space"
  (6) push1 3    # "x"
  (8) push1 4    # "::tcl::string::is"
 (10) invokeReplace 4 2
 (16) done
```

Improving Generation

* Expanding the set of cases for which existing compilers generate "good" code
  * Avoid doing complex (expensive!) exception processing when no exceptions are present
  * Especially the `try...finally` compiler
  * Also `dict with` with an empty body
* Generating jumps for `break` and `continue`
  * Even when inside expansion inside nested evaluation inside…

A script-readable bytecode

Disassembler

Improving Inspection

* `tcl::unsupported::getbytecode`
  * Currently on a development branch, `dkf-improved-disassembler`
* Returns a *script-readable* version of the disassembly
  * Dictionary of various things
* *Lots* of interesting things inside
  * Opcodes, variables, exception handlers, literals, commands, …
* Can easily build useful tools on top
  * *Example next slide…*

Example: foreach loop

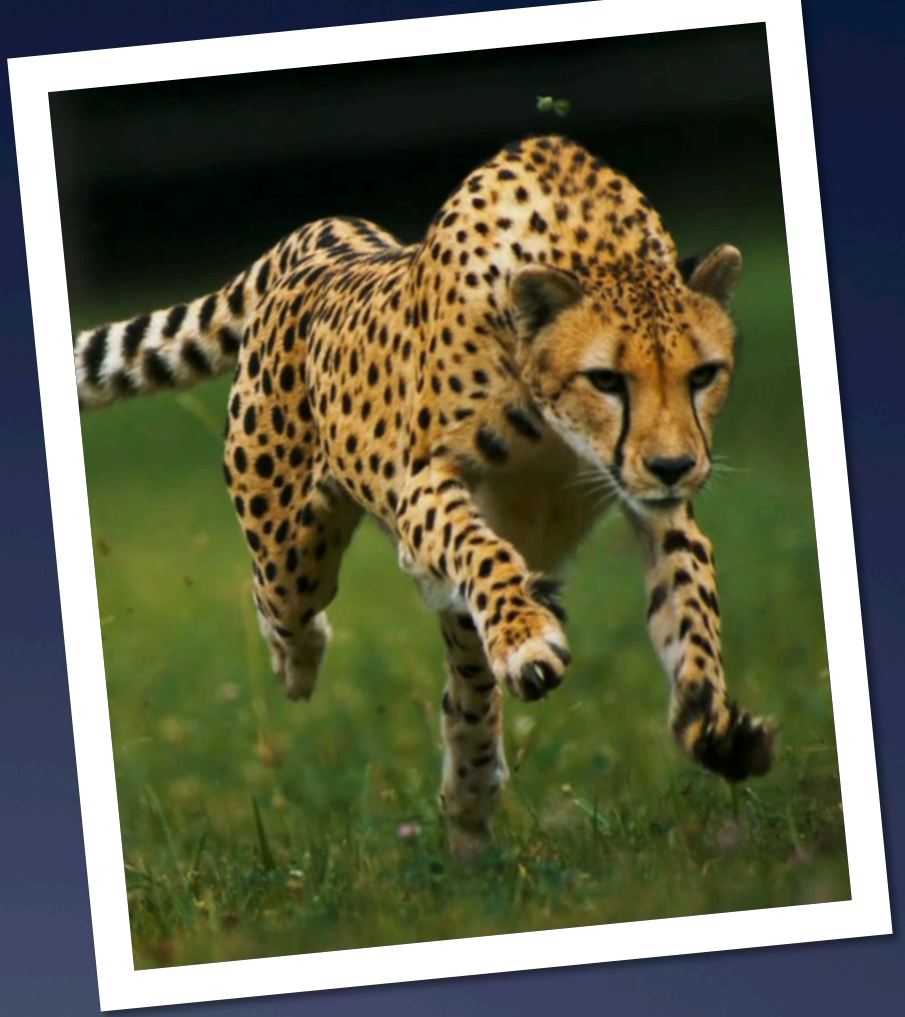
```
::tcl::unsupported::controlflow lambda {{} {
    foreach foo $bar {
        puts [list {*}$foo {*}$bar]
        break
    }
} ::tcl}
```

Inside the Disassembly Dict

* `literals`
  * List of literal values
* `variables`
  * List of variable descriptors (name, temporary, other flags)
* `exception`
  * List of exception ranges (definitions of where to go when an opcode throws an error, a break or a continue)
* `instructions`
  * Dictionary of instructions and arguments, indexed by address
* `auxiliary`
  * List of extra information required by some instructions (foreach, etc.)
* `commands`
  * List of information about commands in the bytecode (source range, bytecode range)
* `script`
  * Literal script that was compiled
* `namespace`
  * Name of the namespace to which the sbytecode is bound
* `stackdepth`
  * Maximum depth of execution stack required
* `exceptdepth`
  * Maximum depth of nested exceptions required

Towards a true bytecode

Optimizer



Optimization

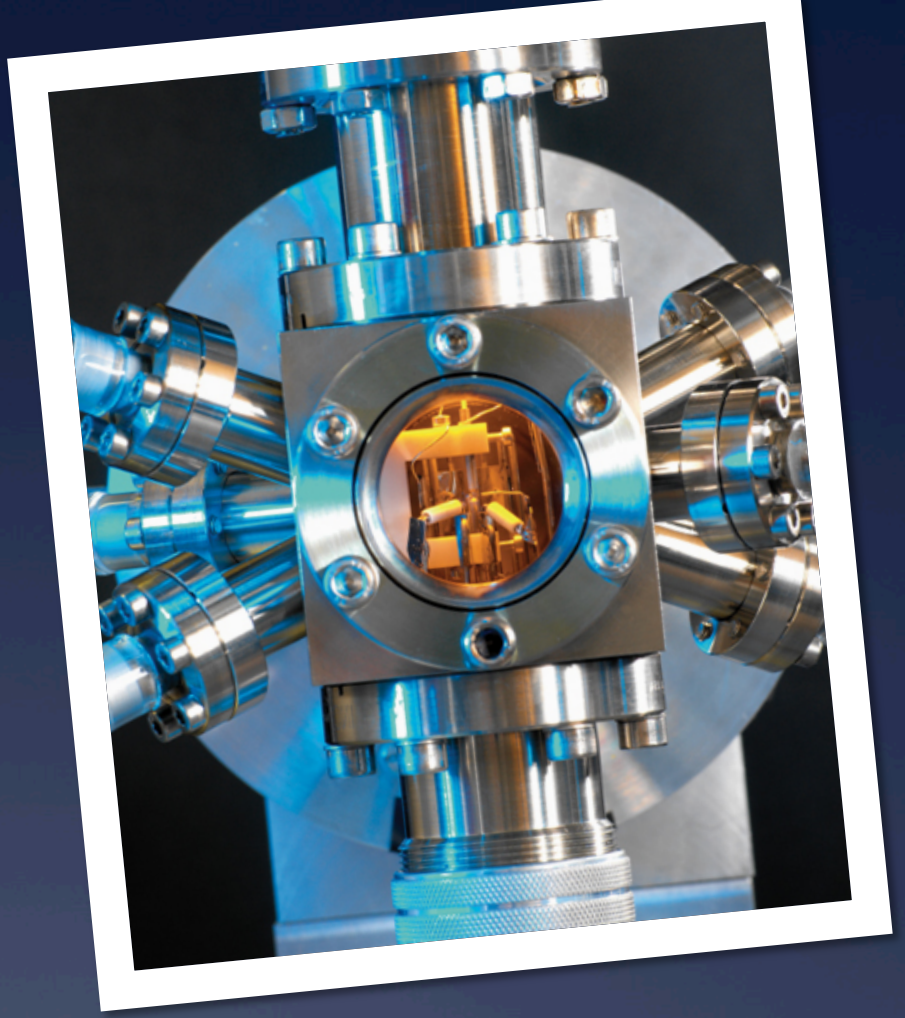
* Tcl now has a formal bytecode optimizer
  * Initial aim: fewer peephole optimizations in bytecode engine
  * Very early days!
* Part of 8.6.1
* Depends on very efficient handling of multi-"`nop`" sequences in bytecode engine

Current Optimizations

* Strip "`startCommand`" where possible
  * Inside `::tcl`, and
  * With *fully-bytecoded* procedures that do not create variable aliases
* Converts zero-effect operations to "`nop`"s
  * "`push` *anyLiteral*; `pop`"
  * "`push` *emptyLiteral*; `concat`"
* Tidies up chains of jumps
  * Avoid jumping to another jump if possible
* Strips *some* entirely unreachable operations

Much still to do

- A number of fundamental optimizations needed
  - Control flow analysis
    - "pop" hoisting to clean up if branches
  - Reordering of instructions
  - Full dead code elimination
- Optimize Tcl using Tcl
  - Close the assembler gap
  - Care required!
    - Optimizing the optimizer could be hard to debug…



Measured effects on

Performance

Methodology

- All timings done with ***same*** build and execution environment
- Measure time to execute a small script
  - Careful to avoid most performance problems
- Invert to get calls/sec
  - "Performance"
- Normalize

```
proc Fibonacci {n} {
    set a 0
    set b 1
    for {set i 2} {$i <= $n} {incr i} {
        set b [expr {$a + [set a $b]}]
    }
    return $b
}

proc benchmark {title script} {
    eval $script
    for {set i 0} {$i < 20} {incr i} {
        lappend t [lindex [
            time $script 100000
        ] 0]
    }
    puts [format "%s: %4f" $title \
            [tcl::mathfunc::min {*}$t]]
}

benchmark "Fibonacci" {Fibonacci 10}
```

Raw Performance (time/iter)

Program	8.5.9	8.5.15	8.6b1	8.6b2	8.6.0	8.6.1
ListConcat	1.1609	0.4097	1.5622	0.5405	0.5433	0.4737
Fibonacci	1.5906	1.2710	1.8087	1.4340	1.4620	1.4114
ListIterate	3.3059	3.0234	3.5981	2.1105	2.1232	2.1599
ProcCall	1.1510	0.8695	1.4590	1.3083	1.3039	1.2996
LoopCB	1.6978	1.0508	1.8496	1.4095	1.4581	1.5382
EnsDispatch1	1.6907	1.0425	2.0192	1.3988	1.4293	0.9404
EnsDispatch2	1.0189	0.4875	1.4117	0.9670	0.3406	0.3763
EnsDispatch3	1.9381	0.5133	1.5587	1.2390	1.2585	1.1909
EnsDispatch4	0.9240	0.4369	1.2799	0.7928	0.7925	0.8167
DictWith	3.7534	2.5671	4.1987	1.9461	1.2926	1.3514
TryNormal	N/A	N/A	27.2137	1.4110	1.4075	0.5086
TryError	N/A	N/A	39.1749	3.8483	3.8556	3.9413
TryNested	N/A	N/A	58.8109	7.6793	7.6454	11.9620
TryNestedOver	N/A	N/A	40.3560	4.1359	4.1963	4.3093

New concat

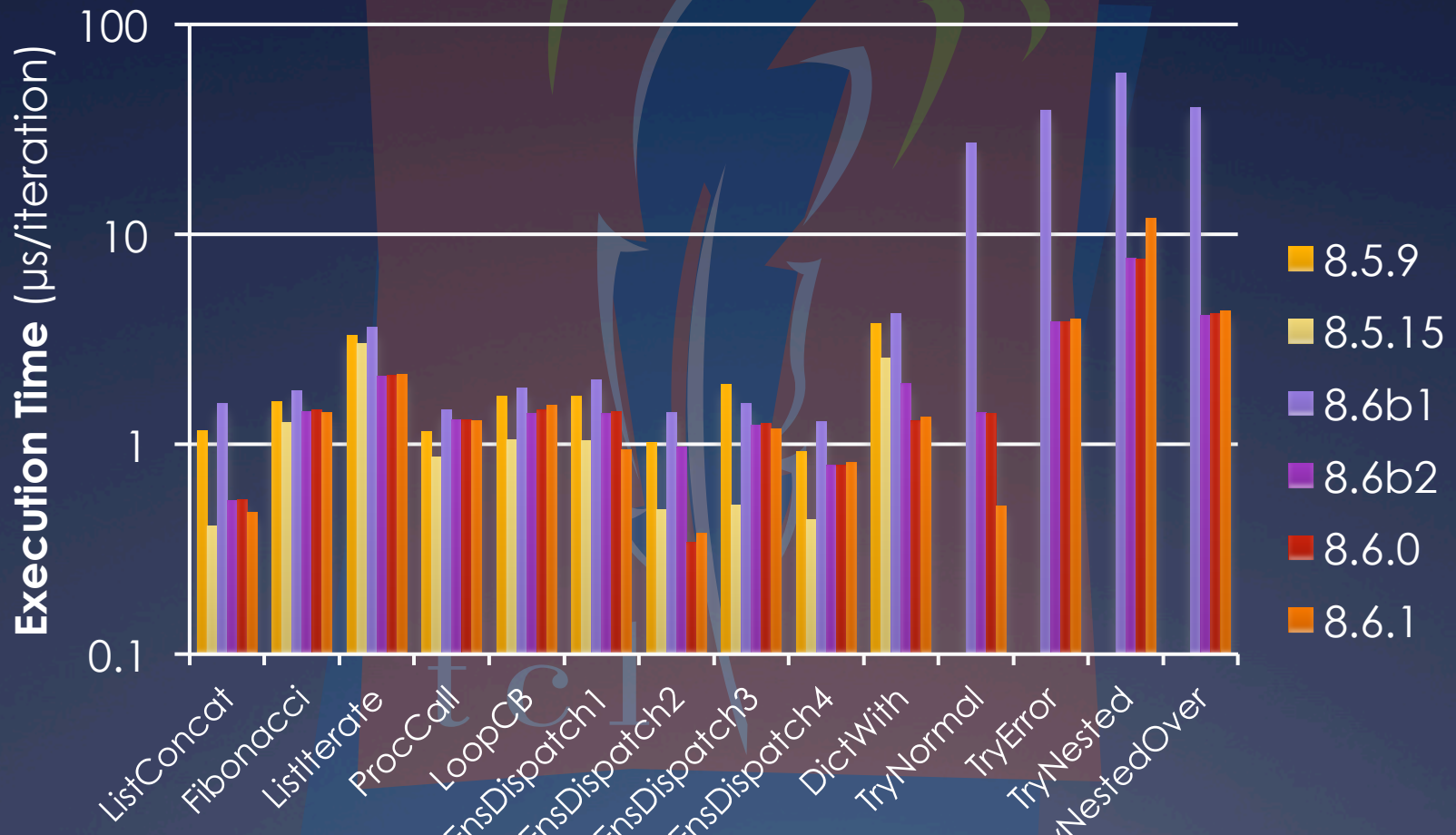
General Operations

Ensembles

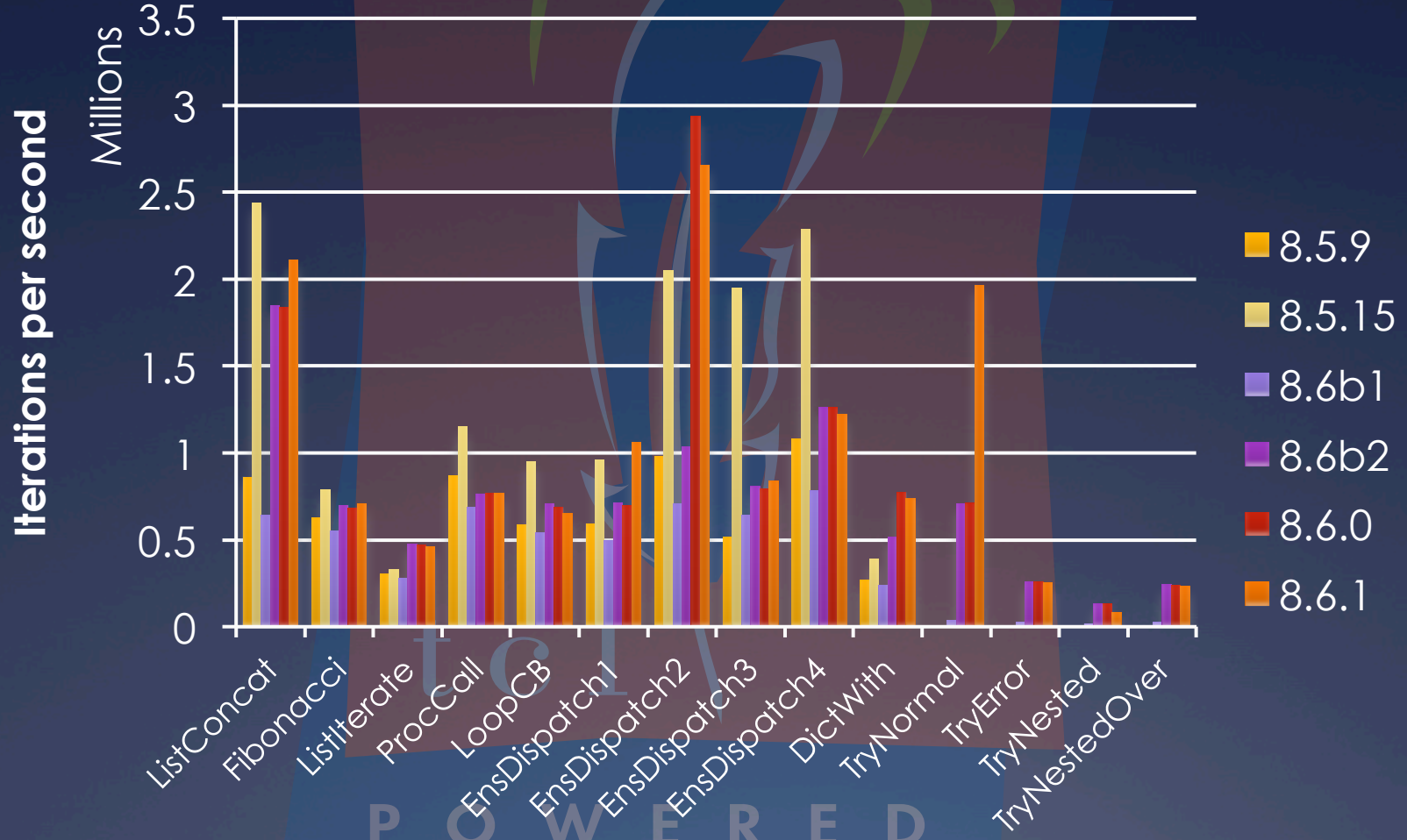
dict with

try

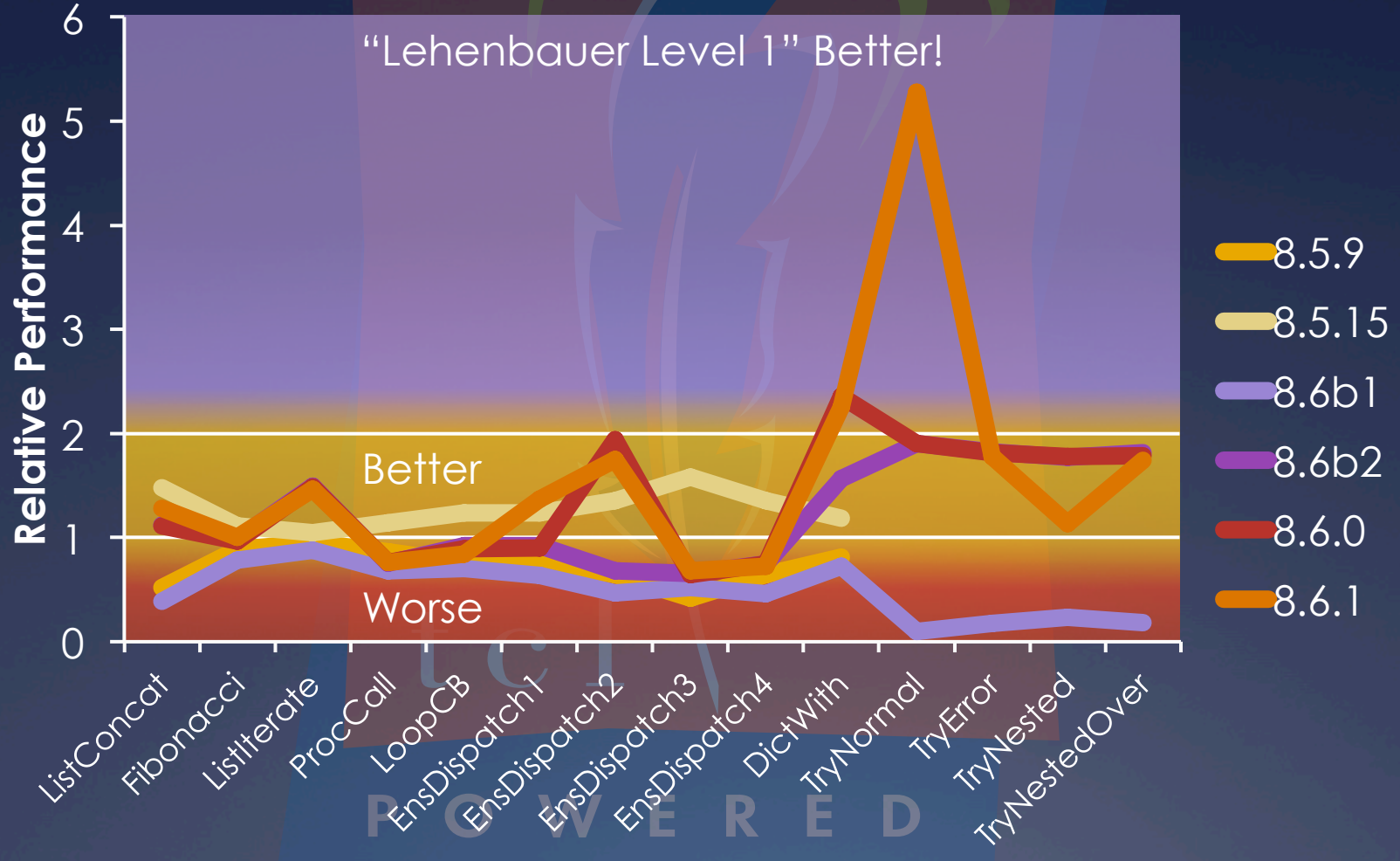
Raw Speed


Execution Time (µs/iteration)
100
10
1
0.1
ListConcat
Fibonacci
ListIterate
ProcCall
LoopCB
EnsDispatch1
EnsDispatch2
EnsDispatch3
EnsDispatch4
DictWith
TryNormal
TryError
TryNested
TryNestedOver
8.5.9
8.5.15
8.6b1
8.6b2
8.6.0
8.6.1

Performance



Relative Performance



Normalized to *mean* of 8.5-series performance (8.6 for try)

Performance Measurement Highlights

* 8.6 is *not* universally faster
  * Procedure calls pay a real penalty (NRE)
* 8.6.0 is *not* universally faster than betas
  * But you probably don't want to worry about that
  * 8.6b2 universally faster than 8.6b1
* 8.6.1 is *sometimes* much faster than 8.6.0
  * `try` now about as cheap as `catch` when no error
* System binary may not be built in fastest mode
* Which C compiler *really* matters

Implications for Optimization

- Improving the compilation of commands provides the biggest gain
  - But only for code that uses those commands
  - Doesn't deliver a quantum leap for most
- General optimization has had little impact so far
- Answering "*Is Tcl getting faster?*" is hard
  - Some things are faster, some are not
  - "It depends"
    - We can easily answer for particular scripts
  - How should we weight each sample script to get an overall figure?

Future

Directions

Where next?

* Integrate "`getbytecode`" into trunk
  * Name?
* Compile more commands
  * Some of the biggest wins will be *very* hard to get right
  * Some should be done without immediate wins, because they strengthen the type algebra
* Compile more cases with existing commands?
* Can we optimize in Tcl?
  * Definitely can't do so yet; can't assemble `foreach`

Where next?

- The command dispatch mechanism is quite a bit more expensive in 8.6
  - Can we improve it?
  - Several performance tests *very* sensitive to this
    - That's one reason why no TclOO benchmarks this time
    - Warning! Might be optimizing for benchmarks, not for reality
- Can we inline sufficiently simple procedures?
  - Suspect it is fairly easy for variable-free code
  - Only really relevant with some variables…

Where next?

- Can we generate native code?
  - Topic for Tcl 9.0!
  - Automatic type annotations are key
- The Lehenbauer Challenges
  - Attaining even Level 1 (speed ×2) is hard
    - Arguably the case for a few scripts
  - Level 2 (×10) is extremely difficult!
    - Bytecode engine is ***not*** fast enough