# Tcl bytecode optimization: some experiences


Kevin Kenny
GE Global Research
kennykb@acm.org

Miguel Sofer
Universidad Torcuato di Tella
mig@utdt.edu

Jeffrey Hobbs
ActiveState Corporation
jeffh@activestate,com



## Abstract

Tcl's bytecode compiler and engine have now been in place for five releases (8.0 through 8.4). During that time, a number of optimizations have been made. This paper reviews recent optimization work and attempts to quantify the change in performance that may be expected when applying certain common optimization patterns. It also presents directions for future work in improving the performance of both the Core commands and extensions.


## 1. Introduction

In the Dark Ages (that is, about five years ago), the Tcl interpreter had the dubious distinction of being among the slowest programming language implementations anywhere. The slowness came largely from the simplicity of its execution model. It is a string substitution language, with the meanings of strings determined at run time; all names are bound late. Through release 7.6, it was implemented naïvely: it was a strict string interpreter.

Tcl 8.0 [7] inaugurated a new execution model that allows for caching of representations other than strings in a new `Tcl_Obj` structure[1]. It also added a new bytecode compiler and interpretation engine. The goal of the improvements is to avoid performing expensive operations such as parsing and formatting strings, without changing the semantics of the language.

### 1.1 '`Tcl_Obj`' and literals

A `Tcl_Obj` is a structure that represents a "string with an internal representation". The execution engine uses the internal representation to remember information about an object. For example, in evaluating a command such as:

```
myCommand arg1 arg2 arg3
```

the string, '`myCommand`' will be stored in a `Tcl_Obj` whose internal representation will announce that it is of type, '`tclCmdNameType`' and has a value that caches the data needed to execute the command, '`myCommand`'. When the command is evaluated again, this information will be available, and there will be no need to look up '`myCommand`' in the hash table of commands a second time. This caching is made even more effective by the use of a shared literal table: when the command above is parsed, each of the four words is looked up in a table of literal strings belonging to the interpreter. If other commands being parsed also have '`myCommand`' as the first word, the same literal will be used. In much the same way, the literal string '`-1`' can acquire an internal representation as an integer. If it is used in several contexts where an integer is appropriate, the value will be reused without recomputing it from the string.

`Tcl_Obj`'s can be shared; that is, many structures such as compiled scripts, variables, and lists can all share references to the same objects. Each object has a reference count that manages the sharing; the interpreter disposes of objects automatically when they have no references. The internal representation of a `Tcl_Obj` can be replaced with another representation at any time, since any desired representation can be obtained from the string representation. If the string representation changes, however, the object is copied unless there is a single reference. This process is referred to as "copy on write."

### 1.2 The compiler and bytecodes

Tcl 8 defines a specialized bytecode language (BCL) and include a just-in-time compiler that translates Tcl scripts to BCL. It also includes, of course, a bytecode execution engine: the C function `Tcl_ExecuteByteCode` (TEBC for short). TEBC is a stack-based engine similar to Forth or PostScript. Each interpreter has its own execution stack of objects. The instruction set contains general-purpose instructions for stack manipulation, branching and conditional branching. Most of its instructions, however, are specialized to the support of specific Tcl commands; it is a relatively high-level engine with a complex instruction set.

One of the most important parts of a bytecode object — at least in terms of performance — is a table that indexes local variables. The BCL contains specialized instructions for accessing variables in this table, as well as more generalized instructions that access variables by name.

## 2. The compiler/engine subsystem

### 2.1 Overview

The general execution path of a script is:

1. If the corresponding `Tcl_Obj` is not of bytecode type, or if it is not usable in the current execution context, parse the string representation of the script and compile the script, storing the bytecode representation in the `Tcl_Obj`.
2. Execute the script by invoking TEBC.

There are a few exceptions to this path. Most notably, commands such as `[eval]` and `[uplevel]` do not compile the script, because the script is assumed to be a temporary object, and it will

---

1. The name of the structure is something of a misnomer; a `Tcl_Obj` is not an object in the sense that object-oriented programming generally uses the term. `Tcl_Value` would have been a better choice, but the name was already taken.

not be possible to reuse bytecode. Also, if the script is a pure list object (that is, it has a list internal representation but no string representation), then the engine can safely assume that it is a single pre-parsed command, and that command is executed directly without reparsing it.

Since an application can redefine even built-in commands such as `[if]` and `[set]`, the compiler must be extremely modular. In essence, every command that can be compiled to special-purpose bytecodes has its own compiler to generate them. When compiling a script, the compiler checks for the existence of a compiling function for each command that it encounters. If such a function exists, the compiler calls it. Otherwise, the compiler generates instructions to invoke the command at run time.

At present, the interpreter does not export any interfaces to provide functions that compile commands defined by Tcl extensions such as Tk; only the Tcl built-ins themselves can be compiled to in-line code. Most of the built-in commands use special-purpose bytecodes. For this reason, any extension to the compiled command set has to be accompanied by a corresponding extension to the BCL and to TEBC.

## 2.2 What's new in Tcl 8.4?

As with each new release since Tcl 8.0, the compiler, engine and runtime library contain a number of enhancements to make Tcl programs faster.

**The compiler.** A number of commands that were executed at runtime in earlier releases are now compiled; these include `[append]`, `[lappend]`, `[lindex]`, `[list]`, `[llength]`, `[lset]`, `[regexp]`, `[return]`, `[string]` and `[variable]`. Of particular interest is the new `[lset]` command. It, together with `[lindex]`, allow for direct manipulation of lists, essentially treating them as linear arrays. If an algorithm needs to change an element of a structure, accessing it directly by position, `[lset]` is by far the fastest way to do so in Tcl.

The compiler's ability to recognize and cache local variables is much improved over Tcl 8.3. In particular, it recognizes variables that are not set within a procedure; speeding up code like

```
proc x {} {
    global y
    return foo_$y
}
```

In addition, a number of miscellaneous improvements have been made. Constant conditions (*e.g.,* `if {0} {...}`) are resolved at compile time, and procedures with empty bodies compile to nothing; these two changes mean that the `[assert]` command in tcllib costs nothing if assertions are not enabled. Loops using `[for]` and `[while]` are changed to move unconditional branches outside the loop body; in other words, code like

```
while { ... condition ... } { ... }
```

which formerly compiled code looking like:

```
x: ... test condition ...
   branch-if-false y
   ... body of loop ...
   goto x
y:
```

now generates the more efficient:

```
   goto y
x: ... body of loop ...
y: ... evaluate condition ...
   branch-if-true x
```

**The engine.** Of course, the new commands added to the compiler required new instructions in TEBC. In addition, many changes improve various small and frequently-used pieces of the engine. One critical change is the fact that the code that accesses local variables is now inlined in TEBC, rather than calling `Tcl_SetVar2`; this change speeds up almost any Tcl procedure.

In addition, a number of peephole optimizations are added. Code that pushes values on the stack only to pop them again immediately is optimized away. Branch instructions that target other branch instructions are retargeted. Tests for equality are optimized so that comparisons of an object with itself are short-circuited. The `[foreach]` and `[catch]` commands have significant speedups from various miscellaneous improvements.

Finally, there is a significant performance improvement in the area of stack access and management of the lifetime of `Tcl_Obj` values. The top couple of stack locations are maintained in the engine in local variables rather than actually being pushed to the stack. In many cases, this management avoids stack operations entirely in simple commands and expressions.

**The runtime library.** The runtime library has several new features that support these improvements. It has a new set of `Tcl_Obj` internal representations that cache variable references, and a new internal API, `TclPtrSetVar`, that accesses a variable directly rather than by name. The lookup of fully qualified names (names beginning with the `::` namespace delimiter) is significantly faster.

In addition, the performance of commands created by `[interp alias]` is much improved; there is a `Tcl_Obj` internal representation that caches the command reference so that the alias does not need to be looked up each time it is evaluated.

**Summary.** Most Tcl programs see substantial improvements from all these changes, but the precise effect is difficult to quantify because so many of them affect specific language constructs while leaving others unchanged. In the next section, we will examine in more detail how to structure Tcl programs to achieve the best performance that Tcl 8.4 has to offer.

# 3. Speeding up your scripts in Tcl 8.4

The cardinal rule of optimization is worth repeating: *Never optimize without instrumenting.* It's essential that you have a clear idea of what you want to achieve by optimization, and a clear idea of where your code is spending its time. It's possible to waste immense amounts of programmer time improving code that turns out to be used rarely enough that it makes no significant difference to the performance of the program as a whole. Tools like the `[time]` command, the tclbench program[2], and the profiler from TclX[3] are all indispensable.

---

2. The tclbench program is available at SourceForge as a module within the tcllib project: `http://tcllib.sf.net/`
3. TclX is now apparently being maintained as SourceForge: `http://tclx.sf.net`

## 3.1 A concrete example: GC-counting

A careful choice of algorithms that considers the peculiarities of the Tcl core can achieve performance gains of up to an order of magnitude. Some examples can be found in the tclbench benchmark suite; of particular interest are the improvements that resulted from tuning the base64, md5, and sha1 procedures in tcllib. Many other impressive tuning results have been reported on the Tcl'ers' Wiki ([2], [9],[12]).

Bastien Chevreux [3] reports in detail on one particular example, that of determining the fraction of letters in a DNA sequence that are `G` and `C`. Essentially, his problem is to count the number of occurrences of a particular set of letters in a string. Figure 1 shows in graphical form the results of nineteen different implementations that he tried, and their relative performance on Tcl 8.3.4 and Tcl 8.4. The topmost implementation, `GCCont_r1`, is a "naïve" translation of a C function that performed the calculation; the remaining implementations are various attempts, some better than others, to improve its performance. (Details of the various implementations are available in Chevreux's paper.)

Several things are noteworthy in examining these results, which are typical of a concerted effort to optimize Tcl code:

- The speedup that can be achieved by careful coding at the Tcl level is much greater than the speedup obtained by upgrading; the fastest implementation in 8.3.4 is 8.5 times faster than the naïve implementation, while upgrading to 8.4 would gain only a 20% improvement.
- For most programs, Tcl 8.4 is noticeably faster; for a few, the improvement is a factor of three or more.
- The improvement to the performance the various implementations is by no means consistent; some are spectacularly better, while some are even slightly worse, according to the specific language features they use. This non-uniformity underscores the importance of measuring code, rather than simply following guidelines.

Despite the difficulty of offering specific guidelines for making Tcl code run faster, a few general principles can be observed. The next few sections discuss some of these.

## 3.2 Avoid coding in Tcl if a desired function is available in C

Some Tcl commands are implemented in C code that is carefully optimized for speed or memory consumption. In particular, loops in C are much faster than loops in bytecodes, so it pays to look for ways in which your loops can be "flattened" into single Tcl statements.

*Iterate over lists with the* `[foreach]` *command, and use the extended* `[foreach]` *syntax.* Consider the following example:

```
proc test1 { list } {
    foreach item $list {
        set x $item
    }
}
```

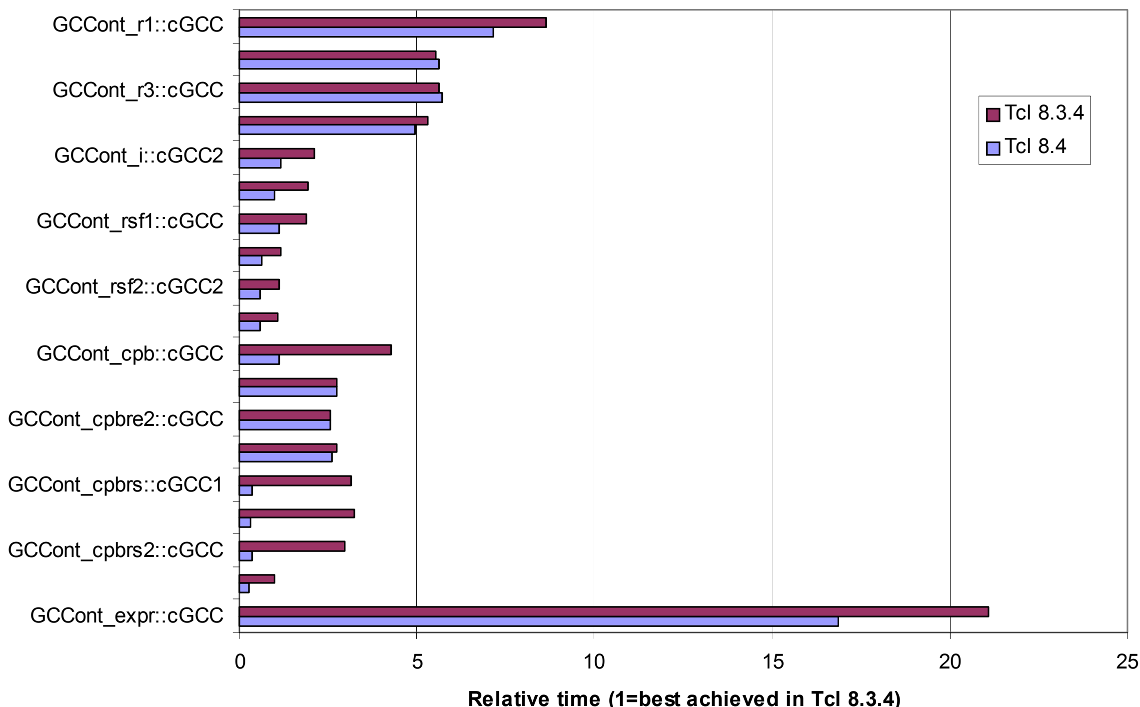


**Figure 1. Benchmark results for competing implementations of a typical Tcl procedure.**

```
proc test2 { list } {
    set imax [llength $list]
    for { set i 0 } \
        { $i < $imax } \
        { incr i } {
           set x [lindex $list $i]
        }
}
set list {1 2 3 4 5 6 7 8 9}
puts [time {test1 $list} 10000]
puts [time {test2 $list} 10000]
```

Running the program gives the result:

```
11 microseconds per iteration
16 microseconds per iteration
```

showing how the C-coded loop is faster. The results are even more impressive when iterating through multiple lists in parallel.

*Generate lists using* `[split]` *or* `[binary scan]` *so as to exploit* `[foreach]`. Consider code that iterates through the characters in a string, reduced to its bare essentials:

```
proc count1 { s } {
    set j 0
    set l [string length $s]
    for { set i 0 } { $i < $l } \
        { incr i } {
        set letter [string index $s $i]
        incr j
    }
}

proc count2 { s } {
    set j 0
    foreach letter [split $s {}] {
        incr j
    }
}

set string {The quick brown fox}
append string $string
puts [time {count1 $string} 10000]
puts [time {count2 $string} 10000]
```

Once again, the code using `[foreach]` wins[4]:

```
80 microseconds per iteration
52 microseconds per iteration
```

An extension to this is that it's often faster to bring an entire file into memory and process it there. The following program shows a line counter implemented as a loop using `[gets]` and then as a C-coded loop:

```
proc test1 { filename } {
    set count 0
    set f [open $filename r]
    while { [gets $f line] >= 0 } {
        incr count
    }
    close $f
    return $count
}

proc test2 { filename } {
    set f [open $filename r]
    set data [read -nonewline $f]
    close $f
    set count 0
    foreach line [split $data \n] {
        incr count
    }
    return $count
}
```

`[foreach]` wins every time, unless memory consumption is prohibitive:

```
905 microseconds per iteration
675 microseconds per iteration
```

## 3.3 Choose how to evaluate scripts

The `[uplevel]` and `[eval]` commands, by design, do not compile the scripts that they evaluate. By avoiding the overhead of compilation, they get the best performance for strings that are evaluated only once, as when they have been read from a file or input from the user. For this reason, they are much slower than bytecode interpretation, and you can get much better performance in code that evaluates the same string repeatedly by using other commands to do it.

The fastest way to evaluate a constant string, if possible, is to compile it into a procedure. Procedures can take advantage of several optimizations, such as the local variable table, that are not available to other bytecode-compiled strings.

If for some reason you can't use a procedure, you can at least evaluate a constant string using one of the commands that caches a bytecode representation:

- `[uplevel #0 $script]` can be replaced with `[namespace eval :: $script]` or `[interp eval {} $script]`.
- `[eval $script]` can be replaced with `[if 1 $script]`.
- `[uplevel 1 $script]` can be replaced with, believe it or not, `[uplevel 1 [list if 1 $script]]`. (Read on to see why this last change works.)

Commands that would otherwise involve substitution can also be sped up, by constructing them as pure lists. Pure lists are the output of commands such as `[list]` and `[lappend]` before they have acquired string representations. The interpreter knows that such lists need no substitutions and always consist of a single command. Rather than constructing a string representation of a pure list and handing it off to the parser, it interprets the first element of the list as a command name and the remaining elements as its parameters. It is therefore must faster (as well as more reliable in the face of special characters) to say

```
eval [list $command $arg1 $arg2]
```

4. Here we are also seeing the difficulty of executing `[string index]` against a UTF-8 string. The `[split]` command is able to walk the multibyte sequences once, while `[string index]` scans them repeatedly.

than to say

```
eval “$command $arg1 $arg2”
```

The following code illustrates the results of some of these techniques:

```
proc test1 { script } {
    uplevel 1 $script
}
proc test2 { script } {
    uplevel 1 [list if 1 $script]
}
puts [time {test1 {set x 1}} 10000]
   # parses the script every time
puts [time {test1 [list set x 1]} 10000]
   # evaluates the command directly
puts [time {test2 {set x 1}} 10000]
   # caches bytecodes
```

Evaluating lists and evaluating bytecodes are both nearly twice as fast as evaluating strings:

```
18 microseconds per iteration
10 microseconds per iteration
9 microseconds per iteration
```

In addition to taking care to evaluate constant scripts, it's important to evaluate constant expressions as well. Always use braces around the arguments to `[if]`, `[while]`, and `[expr]`!

## 3.4 Avoid shimmering

Shimmering refers to repeated conversion among string and internal representations. The conversion is, of course, somewhat expensive, and should be avoided.

One very common source of this shimmering is the use of numbers as indices into arrays. This usage is natural to C, but awkward in Tcl. The Tcl counterpart to C's arrays is the list. Code that accesses arrays linearly with a for loop in C can be replaced with a `[foreach]` loop in Tcl. Code with other access patterns can be replaced with `[lindex]` and `[lset]`. A good example of optimizing code with irregular access patterns can be found at [9], which demonstrates code that shuffles the elements of a list into random order.

Another cause of excess shimmering is mistakenly using lists as strings. If you care about performance, never check whether a list is zero-length with code like

```
[string compare {} $args]
```

This code will generate the string representation of the list to compare it against the empty string! Much preferable is to say

```
[llength $args] == 0
```

which will use the length of the list representation and avoid generating the string. You should also watch out for code that treats numbers as strings. While it was once a Tcl idiom to coerce an integer to a floating point number by appending `.0` to it:

```
set float ${int}.0
```

that method passes it through the string representation. Much preferred is:

```
set float [expr { double( $int ) }]
```

## 3.5 Optimize variable access

The bytecode compiler contains a number of optimizations to make variable access faster, but these optimizations work differently between different constructs. In order from fastest to slowest, the preferred methods for variable access are:

1. Local variables in procedures; these are resolved at compile time and accessed directly by reference.
2. Variables imported into a procedure with `[global]`, `[variable]` or `[upvar]`.
3. Variables accessed via a fully-qualified path name: `$::foo::bar::grill`.
4. Variables represented by a partially-qualified name, that is, one without a leading namespace delimiter: `$foo::bar::grill`. Note that these variables can be cached only in a single context. If a variable name like this appears in multiple procedures, it will shimmer[5].
5. Variables resolved at run time. Excessive use of `[eval]`, `[subst]`, or non-constant first arguments to `[set]` is a performance killer.

## 3.6 Avoid unnecessary copying of internal representations.

The fact that `Tcl_Obj` values are shared, with “copy on write” semantics, can cause excessive copying of the internal representations. The reason is that the engine doesn't know that values are about to go out of scope. Consider the statement:

```
set list [lreplace $list end end]
```

Let's walk through what the engine does with this.

- The variable `list` is looked up, and its value is pushed to the execution stack. There are now two active references to the variable, one in the variable and one on the stack.
- The `[lrange]` command is invoked. Finding two active references to its argument, it is forced to copy it. It can then delete the last element and return the result.

This type of copying is best avoided by using Tcl's commands that perform read-modify-write operations on variables:

- Prefer `[incr x]` to `[set x [expr { $x + 1 }]`.
- Construct long strings using `[append]` and long lists with `[lappend]`.
- Use `[lset]` to modify elements in the middle of lists.

For operations that don't have built-in read-modify-write commands, Donal Fellows offers one solution, the `[K]` combinator.

```
proc K { x y } { return $x }
```

5. This behavior is arguably a bug caused by over-eager sharing of literal strings in the compiler, and may be fixed in a patch release. For Tcl 8.0 through 8.4.0, you should be aware of it.

This procedure is invoked as [K $x [set x {}]. It provides a destructive readout of the variable x:

- The value of x is pushed to the stack; that Tcl_Obj now has two references.
- The empty string is now placed in x, destroying the reference to x's previous value. The Tcl_Obj on the stack now has a single reference.
- The empty string is now pushed on the stack as the second parameter to [K]. [K] unstacks its two parameters and returns the first — which is now unshared. Having an unshared value allows commands like [lreplace] to manipulate the value without copying it. The following example shows two procedures that each add 10000 elements to a stack and remove them again. One is forced to copy the representation of the stack, and the other preserves it.

```
proc test1 {} {
    for { set i 0 } { $i < 10000 } \
            { incr i } {
       lappend list $i
    }
    for { set i 0 } { $i < 10000 } \
            { incr i } {
       set list [lreplace $list end end]
    }
}
proc test2 {} {
    for { set i 0 } { $i < 10000 } \
            { incr i } {
       lappend list $i
    }
    for { set i 0 } { $i < 10000 } \
            { incr i } {
       set list [lreplace \
               [K $list [set list {}]] \
               end end]
    }
}
puts [time test1]
puts [time test2]
```

The performance difference achieved by avoiding the copy is spectacular, nearly a fifty-fold speed improvement:

```
2574945 microseconds per iteration
54666 microseconds per iteration
```

One other thing is worth noting in the examples given above: the use of [lreplace] as opposed to [lrange]. The [lrange] and [string range] commands always copy the sublist or substring that they manipulate. The [lreplace] command, on the other hand, will reuse an existing representation if it is unshared. For this reason, combinations of [lrange] and [lappend], with appropriate use of [K], are the foundation of the struct::stack and struct::queue modules in tcllib.

## 3.7 Prefer [string] to regular expressions

Regular expressions are amazingly powerful, and for complex string-matching tasks, they are the right tool for the job. For simpler operations, though, the [string] command is much faster because much of it can be compiled to in-line code. In particular, you should use [string map] where possible in preference to [regsub], and [string match] or [string first] in preference to [regexp].

## 3.8 Take good care of your internal representations

In some cases, such as generating large lists, converting large strings between UTF-8 and UCS-16, or doing multiple hash table lookups (as in parsing long namespace strings such as ::a::b::c), the interpreter has to do a lot of work to make an internal representation. In performance-critical code, these hard-won internal representations are precious, and it pays to preserve them.

The stooop package in tcllib gives a good example. When using a variable x within an instance y of a class c, it needs to generate a name like ::c::y::x. When the instance is created, the variable name is stored in an array. When the variable is accessed within an instance, the name is retrieved from the array, preserving the internal representation of the name (which contains the variable reference).

## 3.9 Never put off to run time what you could do at compile time.

If you are truly desperate for performance without resorting to C coding, you also should consider giving the bytecode compiler as much of the work as possible. Sometimes doing so involves inlining other procedures. Since Tcl at present lacks an "inline procedure" or "macro" facility, you need to invoke [proc] with a generated string to add inline code. This technique can also be used to inline constants that would otherwise be stored in global variables. Any example of the technique is, alas, lengthy; interested readers should examine [2] for an example where a tenfold improvement was achieved on a particular benchmark involving a random number generator.

# 4. Directions for future work

The discussions above provoke a number of ideas where Tcl's bytecode performance could be improved substantially. Some of the ideas are better investigated than others; this section describes some of the more promising lines of attack.

## 4.1 Macros or inline procedures

One thing that is still fairly slow in Tcl is procedure invocation. Because of the dynamic nature of the language, there is a fair amount of work that the engine has to do when invoking a procedure, including checking for command traces, establishing a new activation context that includes facilities for dynamic lookup of commands and variables, and bookkeeping for [info level] and $::errorInfo.

For simple, short procedures that don't require much of Tcl's dynamic nature, programmers would want the ability to generate inline code. A command with a name like [macro] or [inlineProc] could provide such a facility, and would not require any major redesign of the execution engine.

Another feature that is requested often is to give C-coded extensions the ability to compile extension commands to bytecodes. This facility would be ideal for object-oriented extensions that contain conventional Tcl code but add additional access paths for methods and variables. It would also be an easy way to experiment with additional commands that have “read-modify-write” semantics [1].

## 4.2 Changes to the bytecode language

Another improvement that has been requested frequently is the ability to code directly at the bytecode level. TAL (Tcl Assembly Language) has been used as the working name for such a project.

Implementing TAL would not be horribly difficult. Loading bytecodes without directly compiling Tcl code has already been addressed in the TclPro Compiler. Defining an assembler syntax for the bytecodes would be simple. Providing script-level access to the bytecodes, together with a good macro facility, would permit the generation of code that is both fast and portable.

Unfortunately, current instructions have a high semantic content and are quite specialized to the implementation of specific commands. A Tcl assembly language (TAL) would not be very useful under these circumstances — hand-coded bytecodes would likely perform no better than those generated by the compiler.

In order to allow a wider usage of the instruction set, perhaps even permitting much of Tcl to be coded in TAL and Tcl itself, the engine would need to provide a richer instruction set. The preferred approach is to factor out oft-used coded into new low-level instructions: essentially making Tcl’s virtual machine less of a CISC architecture and more of a RISC one. The current implementation of the bytecode would not adapt readily to the RISC approach, because the instruction execution loop is relatively expensive. (We shall return to this issue in the next section.)

Let us assume for a moment that this problem has been solved, and that a macro facility is available. The benefits that could be obtained from an efficient TAL include:

- A much smaller and stabler C kernel for Tcl, with the rest of Tcl built portably in TAL and Tcl itself - many improvements and bug fixes would only require upgrading some scripts.
- The parser, compiler, and optimizer could all be written in TAL, allowing even different versions to coexist - “fast compile”, “fast exec” and “debug” versions could be loaded at runtime, possibly choosing different versions for different procedures.
- Scripts could define new compiled commands via a compiling function, or define procedure bodies directly in a mix of Tcl and TAL — enabling fast, portable libraries with easily hidden source code. These capabilities are far beyond the possibilities offered by the current TclPro Compiler.

These benefits overlap significantly with the goals of the CriTcl project [8], which is in a much more mature state of development. The two projects are probably complementary. CriTcl would be more suitable for programming low-level algorithms (say, a hash function), or for accessing inherently non-portable API’s provided by a target system. TAL, on the other hand, would be well suited for higher level constructs relying mainly on the Tcl library.

## 4.3 Improving the execution engine

An important part of the cost required to execute bytecodes, and the main barrier to the implementation of a lower-level BCL, is the “execute the next instruction” process. The current implementation involves switching on the value of a byte. The implementation of a typical bytecode instruction ends with a jump to the top of the interpretation loop:

```
top:
switch (*pc) {
   ...
   case INST_LOAD:
      ...
      goto top;
   ...
}
```

In compilers that implement the `switch` statement with indirect jumps (the best case: other compilers generate even worse code), the generated code to advance to the next instruction must:

1. jump to the top of the `switch` statement
2. read the next bytecode `B = *pc`
3. test that `B` is within range
4. compute an offset `off = B * sizeof(void *)`
5. read the target `T = *(jumpTable + off)`
6. jump to `T`

This sequence cannot be optimized much by the C compiler.

Special extensions to the C language may provide some shortcuts. For example, MSVC++ allows the generated code to skip step 3 if enumerated types are used appropriately.

A more interesting example is provided by the “labels as values” extension in GCC[11]. This extension provides functionality similar to Fortran's “assigned `GO TO`”. It allows saving jump targets corresponding to labels in variables, and later jumping to them. It is relatively simple to implement an improved algorithm - a mediocre version of what Forth programmers call “token threading”[6]. One of us tested this approach in an experimental engine[6] with encouraging but not overwhelming results (bytecodes executed 10-20% faster). The C code for a typical bytecode instruction looked like:

```
...
INST_LOAD:
   ...
   goto *(jmpTable[*pc]);
...
```

which compiles to

1. read the next bytecode `B = *pc`
2. compute the offset `off = B * sizeof(void *)`
3. read the target `T = *(jumpTable + off)`
4. jump to `T`

Note that in addition to reducing the number of machine instructions, the code improves the locality of reference of the program

6. The experimental code is still available in the CVS repository, under the branch tag, `S4`, but it is far behind the current implementation.

counter by omitting a jump to the top of the loop. The improved locality also speeds up the code by reducing the probability of cache misses.

A more interesting possibility[10] is to have the Tcl compiler generate indirect threaded code instead, storing, instead of bytecodes, the machine addresses of the C statements that execute the constructs. Under this scheme, the code for a typical bytecode instruction would look like

```
...
INST_LOAD:
    ...
    goto **pc
...
```

This approach avoids both subscript calculation and reading the jump table at run time. This model is used in GNU Forth [5] for threaded code.

This approach has both exciting possibilities and drawbacks:

- It requires using gcc, or else defining small macros that produce the compiler-and-processor-dependent instructions to emulate "labels as values".
- It enables further, finer optimizations to the calling sequence, as studied and implemented in the gforth project. It may also open up the possibility of compiling to machine code by copying and pasting the code from the succesive instructions, accessible through the labels. We have not yet extensively explored the consequences of this idea.
- The best implementation of this method is probably to use a two-stage compiler. The first stage compiles to portable bytecodes, while the second stage generates the non-relocatable threaded or machine code.
- Other similar optimizations could take place while this non-relocatable code. For instance, the second-phase compiler could address literals directly rather than looking them up in a table, code numeric operands in the machine's native format, and perform other machine-specific conversions.
- The "machine code" that is discussed in this section need not be native code. It would at least hypothetically be possible for the "machine code" to be bytecodes for the Java Virtual Machine or the Common Runtime Environment, allowing direct integration with Java or .NET.

The two-stage compiler mentioned above is also needed for compatibility with the existing TclPro Compiler and its companion loader, `tbcload`. Compiled code that has been stored for one machine may need to be loaded upon another, and the enviroment in which it is loaded may not be the same version of the execution engine. A neutral intermediate format is critical to making a threaded-code scheme work.

## 4.4 Addressing literal sharing

There are a few cases where excessive shimmering appears to be unavoidable when executing Tcl code:

- Command and variable names frequently have to be reconverted from their string representation, because they are being evaluated in different namespace contexts. Even when a given string such as 'set' resolves to the same command, it needs to be looked up in each context.
- Simple strings such as '0' have multiple interpretations, each of which has its own internal representation. '0', for instance, can designate an integer zero, a floating-point zero, a character string, a one-element list, and several other things. If code uses the same string in different contexts, the string can shimmer repeatedly.

These problems result from the fact that, in an effort to gain the maximum benefit from cached internal representations, the compiler shares literals across the entire interpreter. The next step toward reducing shimmering is to copy objects more aggressively when developing complex internal representations, and to reduce the scope of literal sharing, perhaps making literals local to individual procedures. It may prove that the best approach is not to share literals at all, but rather to generate a new `Tcl_Obj` for each word in a script.

## 4.5 Polymorphic, mutable objects

Extension writers have also often requested polymorphism of the `Tcl_Obj` internal representation. Typically, an extension writer discovers that polymorphism is required when trying to implement an object that mirrors a C structure. Often, the writer will try to represent the structure as a Tcl list comprising its fields, only to discover that there is no way to have the `Tcl_Obj` contain both representations; it is easy to generate a string representation that converts to the appropriate list, but the structure is lost as soon as the list internal representation is generated. This restriction does not lead to excessive shimmering in practice — because it causes extension writers to avoid representing objects transparently in this fashion! Instead, they normally represent objects by handles: arbitrarily chosen names that designate the objects. This representation causes trouble with lifetime management (the handles are stored in separate hash tables, and are not reference counted), and require additional code to look them up and convert them to object references.

Extension writers, and Tcl programmers in general, also have widely requested mutable objects. The copy-on-write semantics of Tcl are not appropriate to everything. For instance, programmers often want to pass collections of objects by reference and operate on the collections, not on copies. In Tcl, one usually handles pass-by-reference by `[upvar]`, and there is no natural way to store a reference to an object. Alan Perlis's remark that "Lisp programmers know the value of everything, but the cost of nothing," is even more true of Tcl!

Paul Duffin, in his Feather system[4], implemented a number of types of mutable objects, and introduced polymorphism by allowing an internal representation to export multiple interfaces. An object could behave as a list, for instance, by exporting a 'list' interface that allows callers to inspect and set list elements.

No serious effort has yet been made to integrate Feather into the Tcl core. Although the integration effort would involve a fair amount of work, that is not the chief reason for the delay. Rather, a number of issues need to be addressed. Chief among these is that the presence of polymorphic, mutable objects breaks Tcl's contract that "everything is a string." In particular:

- Interpolating an object into another string (as with "`a${x}b`") flattens it to a sequence of characters. If `${x}`

contains an object reference, there is no guarantee that the same object would be constructed if the characters are subsequently converted back to a reference.

- Interpolating a handle into a string presents issues with lifetime management. If `${x}` contains a handle, there is no guarantee that the object will still be available if the handle is converted to a string and back to an object reference.

These issues seem to be a potential source of insidious bugs in scripts. While users of products such as TclBlend and Jacl seem to be able to deal with them readily, careful consideration will be needed before inflicting them upon the broader Tcl community.

In addition to issues of lifetime management, mutable objects introduce subtler changes into Tcl's semantics. In particular, if several variables contain references to the same mutable object, what should variable traces do when one of the variables is used? If the interpreter is required to track down all the uses of the object to fire potential traces, most of the advantages of mutable objects would be lost. If it bypasses the traces instead, it is possible for the string representation of a variable to be accessed or changed without the corresponding traces firing. Neither of these choices seems natural to Tcl.

All these considerations notwithstanding, it seems likely that in Tcl 9, when radical semantic changes can be contemplated, some sort of `Tcl_Obj` polymorphism and mutability will be considered.

## 4.6 New object types

In some cases, when something cannot be fully compiled, at least part of the work of the compiler could be saved. It would be attractive at least to have a couple of new internal representations: `parsedScript`, which would allow parsing a script and reuse of its parse tree in situations where bytecodes cannot be saved, and `hashedString`, which would save the computed hash code for a string for use where the string can be used as a key into different hash tables.

More speculatively, one can imagine a data type that is specific to constructed strings[13] (ones built up by substitution) and to substrings (as extracted by commands like `[string range]` and `[regexp]`). Such a data type could:

- Allow for extraction of substrings without copying them in memory.
- Allow preserving internal representations of parts of strings, so that object references interpolated into strings can be recovered.
- Allow caching information about the provenance of strings, for instance the file name and line number where they occurred. This information would enable line-based debugging, source-file tagging in stack traces, and similar facilities which now require a secondary tool like the TclPro Debugger.

Together with such a data type, we could also envision storing short literal strings in the `Tcl_Obj` structure itself, so that they would not require any additional memory allocation. Both these changes would break a good bit of existing C code, so would have to wait for Tcl9.

# 5. Conclusions

Between Tcl 7.6 and 8.0, there was an improvement in performance of about an order of magnitude for typical Tcl scripts. Releases 8.1 through 8.3 displayed a distressing trend to get incrementally slower (largely because of the demands imposed by Unicode support). In 8.4, the efforts of a number of Tcl maintainers have reversed this trend and made substantial additional improvements to performance.

Many additional optimizations, mostly relating to object copying shimmering, can be foreseen. Some of them simply need programming; others represent changes to the Tcl language semantics and will need a new major release before they can be considered.

Tcl's reputation as a "slow" language is largely undeserved, not least because it is most appropriately used as an interface layer to connect codes in other languages. Moreover, careful coding with attention to how the interpreter manages data can result in order-of-magnitude improvements in Tcl performance.

## References


[1] "The anatomy of a bytecoded command." `http://wiki.tcl.tk/1604`

[2] "Can you run this benchmark 10 times faster?" `http://wiki.tcl.tk/1173`

[3] Chevreux, Bastien, Christoph Göthe, and Sebastian Lepe. "Writing commercial-grade multiplatform end-user applications with Tcl/Tk and PowerTcl." *Proc. 2nd European Tcl/Tk Workshop,* München, 2002. `http://www.t-ide.com/tcl2002e/bach_tclpaper.ps.gz`

[4] Duffin, Paul. "Feather: teaching Tcl objects to fly."

[5] Ertl, M. Anton. "A portable Forth engine." *Proc. EuroFORTH Conf.*, 1993. `http://www.complang.tuwien.ac.at/papers/ertl93.ps.gz`

[6] Ertl, M. Anton. "Threaded code." Unpublished report, Institut für Computersprachen, Technischen Universität Wien. `http://www.complang.tuwien.ac.at/forth/threaded-code.html`

[7] Lewis, Brian. "An on-the-fly bytecode compiler for Tcl." *Proc. 4th Intl. Tcl/Tk Workshop*. Monterey, Calif: USENIX, 1996, pp. 103-114. `http://www.usenix.org/publications/library/proceedings/tcl96/lewis.html`

[8] "Scripted compiler." `http://wiki.tcl.tk/3687.` The paper of Steve Landers and Jean-Claude Wippler presented at this conference describes the ideas more formally.

[9] "Shuffle a list." `http://wiki.tcl.tk/941`

[10] Sofer, Miguel, *et al.* "MS's bytecode engine ideas." `http://wiki.tcl.tk/1685`

[11] Stallman, Richard, et. al. *Using the GNU Compiler Collection (GCC).* Cambridge, Mass.: Free Software Foundation, 2002. `http://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html`

[12] "Tcl performance." `http://wiki.tcl.tk/348`

[13] "Tcl9 and annotated strings." `http://wiki.tcl.tk/3073`