A Compiler for the Tcl Language

Adam Sah and Jon Blow

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720
(asah@cs.Berkeley.EDU, blojo@xcf.Berkeley.EDU)

May 24, 1993

Abstract

Tcl is a highly dynamic language that is especially challenging to execute efficiently. The dual-language nature of the system enforced by the C callback mechanism makes traditional compilation and optimization unrealistic. In addition, the lack of formal data types (and therefore type checking) places severe limits on the ability to provide for efficient data storage at compile time. In this paper, we discuss the many issues involved with compiling Tcl, and present a design for such a system, including the mechanism for embedding a Tcl script into the compiler itself in order to provide user extensibility. The current implementation is presented along with results showing approximately ten times the performance of the existing Tcl interpreter.

1 Introduction

1.1 Overview of the Tcl Language

Tcl[Ous93] is designed to address the need for a "scripting" language, providing high-level control over a program. The interface between Tcl and the running program consists of the Tcl runtime library, which is embedded into the C application code.

Tcl appears to the programmer as a syntactically simple combination of Lisp [Wil86], Perl [WS90] and the Unix Shell language [Bou78]. Like the Unix Shell, it supports nested commands, automatic concatenation, and newline-command-termination. From Lisp, it borrows 'defun'-like syntax, default procedure arguments, and eval(). Like Perl, Tcl's only data type is the string.

In Tcl, every statement can be thought of as a function call, in the form of "cmd arg arg arg", where the 'cmd' pseudo-function is the first whitespace-separated argument and the args are the various whitespace-separated items that follow it. Square-brackets denote nested commands; dollar-signs indicate variable substitution; curly-braces serve to group text into a single argument, without performing substitution.

An example follows:

```
# comments start with a '#' character.
# sets variable a to value "5"
#   (the string, not the number!)
set a 5

# sets b to the string "10".
# The 'expr' command converts the string
#   "5" to the number 5, (it is not
#   performed by the interpeter)
set b [expr $a+5]

# sets c to "5.510"
set c $a.$a$b

# outputs the string "5".
puts stdout $a

# define a new function
#  note default argument of '1'.
# 'proc' is a command taking "fact",
#  "{n 1}" and "if ... " as arguments
#  (curlies are stripped by the interpreter).
proc fact { {n 1} } {
  if {$n <= 1} {
    return 1;
  } else {
    return [expr $n*[fact [expr $n-1]]];
  }
}

# "foo" is called with args "bar", "7"
foo bar 7
```

One of the most interesting features of Tcl is its ability to be embedded and extended through its C language library. From the C programmer's perspective, the Tcl interpreter is treated as an instantiable object (embedded), which is passed the contents of the script to be run. This object exists as a data structure in C working in tandem with a C function library. The Tcl runtime library exposes all of the core commands to the C programmer directly, so Tcl statements may be executed as C function calls to routines provided by the library. This library is identical to the one that Tcl uses to evaluate statements in a script. Thus, programs like `wish` are nothing more than main loops which collect user input and pass it one statement at a time to a routine called Tcl_Eval().

Tcl can also be extended from the C program by registering commands with the Tcl interpreter. The interpreter will call back to your C code when and if that Tcl command is executed. In fact, this is how all of the core language commands are implemented. In the above example, the `set` command is nothing more than a registered callback whose function pointer is the library function which handles set - the very same one which is available to the C language programmer directly! The last statement in the example, "foo" doesn't exist in the default Tcl language - it is either a call to a procedure defined in Tcl itself using the `proc` mechanism, or a callback to a C function defined by the user, where the arguments "bar" and "7" are passed to it, in their string forms.

1.2 Why Do We Need to Compile It?

Most Tcl scripts and commands are designed to operate over high level, expensive constructs, where speed is not the primary consideration. In these cases, the cost of the interpretation is negligible compared to the overhead within individual callback functions. For example, on a Sun Sparc2, the `set` command takes under 75 microseconds. By comparison, creating a new window in X Windows can take a human-noticeable amount of time, even on faster workstations. Using Tcl as a heavyweight integration tool generally does not impact application performance. Even in cases where it does, performance bottlenecks in Tcl can easily be isolated and rewritten in C, in the form of callbacks.

However, this recoding is annoying and inbihits the configurability that Tcl offers; it can also remove some of the modularity of a well-written Tcl-based program. For example, if you discover that a loop written in Tcl is a performance bottleneck, it would be rewritten in C, and a new command registered with the Tcl interpreter. Once this has been done, any commands in the loop are now hard-coded relative to the high degree of configurability that Tcl offers.

1.3 Summary

In the remainder of the paper, we discuss the design and implementation of the compiler in depth. Section 2 describes the compiler interface, an overview of the user's view of the compilation system. Section 3 discusses the high-level design decisions involved with the efficient execution of Tcl. In Sections 4 and 5, the given implementation is presented, first in terms of the compiler, and then in terms of the runtime. Section 6 presents the results of this implementation through a series of small benchmarks, and Section 7 summarizes our efforts and proposes future work.

2 The Compiler Interface

2.1 Previous Work

2.1.1 Emacs Lisp

Emacs is a programmable editor whose underlying language is a variant of MockLisp called Emacs Lisp[Sta86] (or elisp for short). Included in this system is a "byte-compiler" which preparses the code to a binary format. All of the elisp source files are stored in a single directory, with names ending in ".el". Byte-compilation of a source file is output to the same filename with the extension ".elc". When this source file would otherwise be loaded, the system automatically looks for a compiled version of it and will load that in the place of an original source file.

The byte-compiler's output is highly portable. This means that compiled elisp files can be placed on servers, side by side with the original source code, instead of requiring a separate set for each architecture.

2.1.2 Perl

Perl[WS90] is a scripting language with many characteristics of Tcl: it is designed for high-level control over arbitrary input, its syntax is designed around the string data type, and its support for data types is nearly identical- associative arrays, scalar variables, and arrays of scalars (which are similar to Tcl lists).

Perl lacks embeddability, and so fails to a large degree to serve as a high-level control language. Note, however, that Perl does support arbitrary inter-process communication (IPC), so applications which can talk in this way can interface with Perl using this facility.

Perl is compiled on demand each time the script is loaded.

2.2 The User Interface

The Tcl Compiler (TC) operates much like the elisp byte compiler: it produces binary files which the runtime system knows to look for, and if found, will use in the place of the raw source when executing. This runtime is essentially a replacement Tcl interpreter, which takes as input binary data from a file, instead of textual data. Like elisp, TC's output files are portable, and so can be placed on servers.

The reason we chose this model is that Tcl cannot be compiled into pure machine code without the support of a large runtime library. This is due to the user-extensibility system, where builtin commands can be overloaded, removed, added, etc. Even simple statements may have completely different behaviors dependent on some portion of code that the compiler does not have access to. This property is described in further detail below.

One alternative would have been to follow Perl's model, where the source code is read at runtime and compiled on-the-fly before each execution. However, Tcl has a high parsing cost, so it is desireable to preparse source code prior to execution. Additionally, Tcl is used for large applications, some of which are 10,000 lines long. These programs must be compiled prior to execution.

3 Difficulties with Compiling

3.1 Some Terms

To simplify this paper, some terms are explicitly defined. A "statement" refers to an individual line of Tcl code, including all arguments. The first argument is called the "command"; all subsequent whitespace-separated arguments are called "arguments" or "args". If a command was not defined by `proc`, but exists in the interpreter (eg. it is a registered C callback function), it is called a "builtin". Note that the entire Tcl core command set is implemented as builtins.

3.2 Overview

It is tempting to naively design a traditional compiler for Tcl, which outputs pure executable code. However, there are numerous problems with this. First, Tcl is highly dynamic in nature. Commands (functions in the C model) can be called by their string names or rebound at any point through the `rename` command. Similarly, variables can be `unset`, commands may not exist (and hence trigger `unknown` to be called), traces can be placed on all data objects, and so on. To implement all of these features would require both a large runtime library and enormous overhead in each usage, not dissimilar from the same overhead that the existing interpreter incurs.

Second, it is unclear how to efficiently store data for Tcl. Since there are no types in the language, there is no obvious data layout method one can use besides strings. Again, this is what the current interpreter does.

Lastly, Tcl uses C callbacks ("builtins") as first-class functions, where the callbacks have direct access to the interpreter state. Callback functions are free to make changes to, or depend on, the state of the virtual machine; these are called "side effects" in compiler parlance. The difficulty with these side effects in the Tcl/C model is that they are impossible to predict. Without this knowledge, the compiler cannot be sure that any given statement won't cause a `rename`, `unset` or some other state change in the virtual machine.

3.3 Preparsing

A more humble approach is needed. In this vein, one can start by noting that Tcl's "cmd arg arg arg" style statements lend themselves to preparsing. This is because it is always possible to determine the arguments to a given command. At the very least, the compiler can break up the statement into string arguments, which will save some amount of parsing effort normally expended at runtime. Preparsing is very valuable in Tcl; a typical Tcl script spends most of its non-work-related execution time scanning and parsing statements.

However, preparsing statements into arg lists is not a panacea. Many important commands take arguments containing large amounts of data. For example, `if`, `while`, `proc`, `for`, and other commands all take "command lists" in one or more arguments. These cmdlists are often hundreds of bytes long; they effectively can contain entire scripts within them. Clearly, one would like to parse these internally as well. By the same token, we would like to also preparse the boolean expressions associated with `for`, `if`, and `while`, and the list structures used by `lindex` and the other list commands, and so on. In the following example, if arguments weren't individually parsed, the body of this `for` loop would remain in string form, and would require runtime parsing, and thus the compiler would provide little performance improvement.

```
for {set i 0} {$i<1000} {incr i} {
    <many lines of code>
}
```

To accomplish this, a typing system for Tcl is needed, where one doesn't exist. Normally, the types are coerced from string data at runtime, on a per-command basis. It is happenstance that the list format is universal among list commands, for example. Thus, some way is needed to inform the compiler as to the argument types each command expects. Then, if a static string is found at compile-time, we can preparse this to be of that type. For example, since almost all cmdlists are surrounded by curly braces in Tcl source code, the contents are static (no substitution will occur prior to the argument being passed to the command). These can then be parsed at compile-time, and treated as a list of commands. In the case where variable substitution is allowed, such preparsing cannot happen, since the argument value depends on a variable's value, which is unknown at compile-time.

3.4 Solving the Side Effects Problem

While it seems that user-defined C callbacks present an intractable optimization problem, it actually is possible to guess the state of the virtual machine from the Tcl source code. In Tcl, user-defined callbacks are required to use a set of C library routines to access the Tcl internals (ie. variable values). Our system adds hooks to this core Tcl library, which will call out to TC routines when triggered. This allows the TC runtime to maintain more efficient structures, and keep them updated if they change unexpectedly. For example, the TC runtime maintains a table of commands, which is directly indexed, in order to save on the hashing costs normally associated with Tcl variable access. When a C callback function attempts to rename a command, this is trapped and the table entry is updated.

However, this mechanism is still insufficient to allow for more aggressive optimizations, since you cannot determine what the dependencies are until they are triggered. Thus, it is impossible to implement optimizations that involve code motion, elimination of unused variable assignments, and so on.

4 The TC Implementation

4.1 Preparsing- Expression Forest

The Tcl compiler uses a preparsing method, as described above, to output a tree-like structuring of expressions and commands. At the roots are the individual top level script statements, which include only those commands not nested in cmdlists within control structures. For example, all commands found within the arglists for commands like `proc`, `if`, etc. cannot be roots of the tree, although the `proc`, etc. commands can be, if they themselves are not nested.

The node types for this tree include notations for nested commands (`[ ... ]`), concatenated expressions (ie. `$a$b`), etc. An example follows:

Source Code

```
# this is a comment
set a 5

# concatenate the returns of the two nested
#   commands and pass it as an arg to foo.
foo [bar set][foo {5}]
```

Expression Dictionary (Parse Tree)

1.	cmd	"set"
2.	constant	"a"
3.	constant	"5"
4.	cmd	"foo"
5.	cmd	"bar"
6.	constant	"set"
7.	nested statement	[2 args][#5][#6]
8.	nested statement	[2 args][#4][#3]
9.	concat	[2 args][#7][#8]

Top Level Command List (Tree Roots)

```
[3 args][#1][#2][#3][2 args][#4][#9]
```

Notes: "#n" refers to a reference to item number n. The compiler detects multiple uses of the same constants and other primitive objects, and reuses these entries. For example, the entry for the **foo** command (#4) is reused. This system is also able to recognize the difference between commands and static strings, where the command name is the first argument. It also strips the curly braces from the static string in the third line of code. The concatenation and nested commands were discovered at compile time. Lastly, comments have been stripped.

4.2 Argument Parsing

As described previously, it is necessary to preparse arguments in order to achieve real performance gains. To support this, we introduce a type system into Tcl for parsing compile-time constants, and a system for determining which builtin commands expect which types for each argument.

Consider the 'incr' command. In its first argument, it expects to be passed the name of a variable. In its second argument it (optionally) expects an integer value (a string which can be parsed as an integer). Thus, if the input contains the statement

```
incr a 5
```

we can know to treat "a" as the variable whose name is "a", not the string "a". Likewise, "5" here is the integer value 5. Note that if the "5" were replaced by "$b" for example, then we couldn't assume anything, except that the value of b might be a string which has an integer representation. Even this is not necessarily true, since Tcl supports exception handling, so this blunder may be intentional! The handling of these more difficult cases is discussed below in the runtime system, since these are dynamic effects that are orthognal to the problems encountered by the compiler.

It would be entirely possible to hardcode the compiler to recognize these types and the commands which use them. However, Tcl is extendable. This means that a user could potentially author his or her own builtin which takes as an argument a list or some other large structure, which we would like preparse. Thus, it makes more sense to allow the compiler itself to be extensible, so such power users can compile their own commands and arguments.

For this job, Tcl itself is ideal, and hence we embed an interpreter into the compiler. This interpreter reads in a config script at startup which declares data type support and associates them to builtin commands. The Tcl script contains the following two commands, in addition to the Tcl core. The backslahes indicate that the arguments should be part of a single command.

```
type <name> <parseproc> <codegenproc> \
     <loadproc> <printproc>
```

This declares a new argument type, which will use `parseproc` to convert string data into parsed data of this type, `codegenproc` to output into binary form, `loadproc` to load at runtime, and `printproc` to converted back into string form for output. These procs are just names, which are associated statically by the C callbacks to real C function pointers taking specific arguments, using the Tcl hashing mechanism provided by the Tcl core library (Tcl_HashXxxx).

```
builtin <name> <parseproc> <codegenproc> \
        <loadproc> <execproc> {
    { <type1>   <argname1> }
    { <type2>   <argname2> <default2> }
    ...
} <return_type>
```

This declares a new command to be compiled specially, named `<name>`, with procs defined for its compilation and evaluation. Note that all but the execproc may be left empty (passed {} , which evaluates to the null string) because virtually all commands follow a standard style employing known types for each argument. When left empty, TC substitutes a default routine to process the builtin.

For each list element in the body of the builtin declaration, we specify what argument type should be passed to this command. A name is required for identification, debugging, and bookkeeping purposes. The optional third element is a default string for that argument. For example, the 'incr' command is described as follows:

```
builtin incr {} {} {} exec_incr {
    { variable var }
    { integer i 1 }
} integer
```

The "1" is preparsed as type integer and entered into the dictionary. If `incr` is called with only one argument, the runtime will substitute this default value for the second argument.

5 Runtime Issues

As described previously, the TC runtime consists of an interpreter capable of reading in byte encoded Tcl scripts, preparsed as above. In order to support possible state changes such as command renaming, the Tcl core library is modified to update the state of the bytecode interpreter when these changes occur.

It is now appropriate to discuss the actual execution of commands in this new environment. In order to take advantage of the preparsing the compiler has done, we need a new callback interface. This is because the standard argc/argv interface defeats the purpose of preparsing by taking string as arguments.

This implies the need to modify the C callback routines for any commands which are to be compiled. This requirement is entirely reasonable. First, the new interface is very easy to construct from the argc/argv one for a given command. Second, the argc/argv interface is extremely slow due to its use of runtime parsing, and would require modification in any higher-performance system. Lastly, not builtins need to be compiled. Only those which are frequently called or which are passes large amounts of data impact the performance of the final application.

This model needs to be extended in order to minimize the parsing done at runtime, which still happens for non-static strings. First, we modify the data return system from using only strings (Tcl_AppendResult()) to using this parsed form of a typed data pointer when the new style of compiled callbacks are in use. This is needed so that the results of one command can be passed directly to another without reparsing the data return. An example follows:

```
incr a [expr 4+5]
```

Assuming that both `incr` and `expr` are being compiled, we would like the return from `expr` to be the value 9, not the string "9". This could then be directly sent into `incr` without reparsing. The only way to avoid such reparsing is if `expr` doesn't convert its result into string format. Hence the need for this new style of return.

We also need to change the way that variables are stored in the Tcl interpreter. If we take the above example and modify it to read

```
set b [expr 4+5]
incr a $b
```

it becomes clear that if we disallow the value of "b" to be stored in parsed form, we cannot avoid reparsing it before its usage in `incr`. Thus, TC "dual-ports" its variables, storing both a string pointer as well as a compiled data value (the "typed data field"). Now, if `expr` returns an integer, then "b" will store it in the typed data field, and invalidate the string field. `incr` can then be called directly with this parsed value. If "b" were of some other type, it would need to be converted to a string first, then back to an integer, as Tcl currently does implicitly.

If this seems overly complex, recall that Tcl is a typeless language. Lists, boolean expressions, integers and so on are not distinguished by Tcl until individual commands throw exceptions based on bad data. The following code with execute without error:

```
proc cdr {list} {
    return [lrange $list 1 end]
}
set i 2; set j 3; set k "3 4"
linsert "$j $k" [cdr {1 2}] $i$j
```

The output of this command is "3 3 23 4". During the course of execution we have implicitly converted from string to integers, strings to lists, lists to integers, and integers to strings. While this is clearly not an example of good coding practice, it is legal Tcl input, and in many cases similar usages may appear in real source code. It is imperative that the TC runtime be able to perform these operations smoothly.

The "dual-port" implementation provides sufficient machinery to coerce data into parsed form, and keep it there as long as possible. The rule for statements is now simple: for each argument, convert the given data to the proper type, then call the compiled callback interface once the arguments are assembled. The rules for conversion amount to treating the two data fields as caches to the value. If the data is required in a specific type, and does not currently exist in that form, it is converted. If the destination type is a string, its value is stored in the string field; otherwise, the value is stored in the typed data field. The writing of a field is treated as follows:

Currently Valid	Writer	Action
string only	string	string form updated.
typed only	string	string form updated, typed form invalidated.
both	string	string form updated, typed form invalidated.
typed only	typed	typed form updated.
string only	typed	typed form updated, string form invalidated.
both	typed	typed form updated, string form invalidated.

An example is provided:

```
set a 2
set b [expr 4*$a]
incr b
puts stdout $b$a
```

When **a** is initially set to "2", its value becomes the string "2", rather than the integer value 2. This is because we cannot assume its usage as an integer later (and data lossage might result if we guessed wrong). Its usage in the `expr` call is without curly-braces, so again we cannot make assumptions about the compile-time parsing of "4*$a" (consider what happens if **a** is reset to the string "4+5"). As grim as this seems, **b** still receives the integer value "8", because the call to `expr` returned the numeric value 8. Hence the call to `incr` requires no parsing. In the `puts` call of the last statement, **b** requires conversion to string form, but **a** does not.

To see the real benefits of this system, a more realistic example is presented, where we sum the first thousand integers (the hard way):

```
set sum 0
for {set counter 0} {$counter<1000} {
      incr counter} {
   incr sum $counter
}
```

In the standard Tcl interpreter, this requires 6000 calls to hash the strings "counter", "sum" and "incr"; the value of "counter" is parsed 3000 times and the value of "sum" 1000 times; and the comparison string is parsed 1000 times. In TC, no hashing occurs (it happens at load time), the values are parsed precisely once (during the first iteration, when it converted the values to integers), and the comparison string was preparsed, so no effort was required at runtime.

6 Results

The above example involving summing the first 1000 integers required 630 milliseconds on a DecStation 3100 using `wish`. Using TC, it required only 57 milliseconds for speedup of 11 times. Some more examples follow which illustrate the relative strengths and weaknesses of TC. Note that in no case is TC slower than the original Tcl interpreter.

Test 1: Simple variable access and the `incr` command. This is espeically fast under TC, because only the first iteration needs to parse the value of `counter`. The `time` command is used to perform iteration because a `for` loop would interact with the timing data. Note that the times are per iteration in this case.

```
set counter 0
puts stdout [
    time {incr counter} 50000
]
```

Performance- μsec per iter.		
Uncompiled Tcl:	219	μsec
Compiled Tcl:	18	μsec
Speedup:	**12.17x**	

Test 2: Empty loop. The limiting factor of loops are the boolean expressions, which are less efficient to evaluate than simple variable accesses, as compared to uncompiled Tcl.

```
for {set counter 0} {
     $counter<10000} {
     incr counter} {
}
```

Performance- msec total		
Uncompiled Tcl:	3,670	msec
Compiled Tcl:	425	msec
Speedup:	**8.64x**	

Test 3: Nested loops. This shows a more realistic example of the relative speedups of loops.

```
for {set count1 0} {
  $count1<1000} {incr count1} {
  for {set count2 0} {
    $count2<1000} {incr count2} {
  }
}
```

Performance- msec total		
Uncompiled Tcl:	14,090	msec
Compiled Tcl:	1,649	msec
Speedup:	**8.54x**	

Test 4: Pessimistic case- we can do nothing but break the inner command into arguments and pass to the normal evaluation mechanism, because the command is not known at compile time.

```
set oper incr
set count 0
puts stdout [time {
  $oper count 2} 10000
]
```

Performance- μsec per iter.		
Uncompiled Tcl:	244	μsec
Compiled Tcl:	188	μsec
Speedup:	**1.30x**	

7 Conclusions and Future Work

When this project was started, it was unclear that compiling Tcl was even feasible, much less useful. The performance metrics taken clearly show that it is feasible. The latter will be determined as this project nears release and users begin compiling real Tcl scripts. However, the results are auspicious: ten-fold performance improvements are almost always useful.

The existing system is far from complete. There are many types and many commands which have not been implemented, and are currently being left in string form by the compiler and runtime system. Internally, the software has been kludged in several places in order to provide fast proof-of-concept, at the expense of long-time viability, which will have to be replaced in the near future. Lastly, there are the usual array of bugs, which must be eradicated before any attempt is made at production usage.

On the wish list, we include the ability to support direct-conversion functions. This is needed in the case of a arg-type mismatch of two similar types, where one would want implicit conversion (ie. int to float). This has the potential to provide great performance gains in these cases, as compared to the current system which converts to a string and then into the detsination type.

Second, few of the data types are implemented. Of these, the most important are lists. Some preliminary tests indicate possible gains of 1-2 orders of magnitude over the existing system of storing lists as strings.

Lastly, we propose make the replacement callback style (structure pointers instead of string pointers) become the standard for Tcl for some future version. In this system, callbacks which want strings would need to ask for them as a real data type. This has the potential to channel the performance improvements back to users employing the command line, since such a system would not need to rely on the compiler.

We would like to thank the following people for their advice, contributions and other assistance: John Ousterhout, Sue Graham, Raph Levien, Brian Dennis, and the members of the Tcl/Tk Users Group. Ashok Singhal, Colas Nahaboo, and Wayne Throop provided a multi-language test suite.

References

[Bou78] S.R. Bourne. The Unix shell. *The Bell System Technical Journal*, July-August 1978.

[Ous93] John Ousterhout. *An Introduction to Tcl and Tk*. Addison-Wesley Publishing, 1993.

[Sta86] Richard Stallman. *The Emacs Lisp Reference Manual*. The Free Software Foundation, 1986.

[Wil86] Robert Wilensky. *LispCraft*. W.W.Norton, 1986.

[WS90] Larry Wall and Randal L. Schwartz. *Programming Perl*. O'Reilly and Associates, 1990.