The following paper was originally published in the
Proceedings of the Fourth USENIX Tcl/Tk Workshop
Monterey, CA, July 10-13, 1996

An On-the-fly Bytecode Compiler for Tcl

Brian T. Lewis
Sun Microsystems Labs
Mountain View, CA

An On-the-fly Bytecode Compiler for Tcl

Brian T. Lewis

brian.lewis@sun.com
Sun Microsystems Laboratories
2550 Garcia Avenue, M/S MTV29-232
Mountain View, California 94043

Abstract

To improve the speed of interpreting Tcl programs, we are developing an on-the-fly bytecode compiler as part of the Tcl project at Sun Microsystems Laboratories. This new compilation system supports dual-ported objects *that are stored in Tcl variables and passed to procedures instead of strings. These objects allow faster integer, list, and other operations by including an appropriate internal representation in addition to a string. Early performance results show significant improvement for some scripts. On a 167MHz UltraSPARC 1*[1]*,* `lindex` *of the last element of a 100 element list takes 3 microseconds compared to 72 for the current Tcl interpreter. A* `set` *command that stores a new value in a local variable now takes less than 1 microsecond vs. 5.8 to 10 microseconds (depending on the length of the variable name) for the current system. This paper describes the design of the compiler and its current state, outlines its development plan, and gives some early performance results. It also describes some implications of the compiler for Tcl script and extension writers, and describes how to best take advantage of the compiler.*

1 Introduction

Although the current Tcl interpreter is fast enough for most Tcl uses, there are many applications that need greater speed. The traditional approach to improving a Tcl program's performance has been to recode critical portions in C. While this is effective, it is awkward. Also, an increasing number of demanding applications such as the exmh mail user interface [Welch95] are being written entirely in Tcl. Recoding in C also makes the development of portable applications much harder. A significant advantage of Tcl7.5 is that it allows programmers to write scripts that can run unchanged on UNIX®, PC, and Macintosh systems.

1. UltraSPARC and Java are registered trademarks of Sun Microsystems, Inc.

UNIX is a registered trademark, licensed exclusively through X/Open Company Limited.

The two main sources of performance problems in the current Tcl system (Tcl 7.5) are script reparsing and conversions between strings and other data representations. The current interpreter spends as much as 50% of its time parsing. It reparses the body of a loop, for example, on each iteration. Data conversions also consume a great deal of time. Adam Sah [Sah94] found that 92% of the time in `incr`'s command procedure `Tcl_IncrCmd()` was spent converting between strings and integers. Many Tcl programmers avoid using lists today because they know that operations on lists are slow and become slower with long lists. For example, `lindex $a end` requires that the entire list be parsed to discover its last element.

A new Tcl compiler and interpreter are being developed at Sun Microsystems Laboratories to improve the speed of Tcl programs. Our goal for the bytecode compiler is to improve the speed for compute intensive Tcl scripts by a factor of 10.

The compiler translates Tcl scripts at program runtime, or *on-the-fly*, into a sequence of bytecoded instructions that are then interpreted. The compiler eliminates most runtime script parsing. It also makes many decisions at compile time that are made now only at runtime. It can tell, for example, whether a variable name refers to a scalar or an array element, and whether it refers to a local or a global variable. It also compiles away many type conversions. As an example, it can recognize whether the argument string specifying the increment amount in an `incr` command represents a constant integer.

The bytecode interpreter uses *dual-ported* objects extensively. These objects contain both a string and an internal representation appropriate for some data type. For example, a Tcl list is now represented as an object that holds the list's string representation as well as an array of pointers to the objects for each list element. Dual-ported objects avoid most runtime type conversions. They also improve the speed of many operations since an appropriate representation is already available. The compiler itself uses dual-ported objects to cache the bytecodes resulting from the compilation of each script.

An early version of the compiler and interpreter are running. The compiler emits a subset of the instructions that will eventually be supported. The current instructions directly implement variable substitutions and the `set`, `incr`, and `while` commands. Other commands are supported using instructions that invoke the associated C command procedures. These instructions push and concatenate the dual-ported objects that hold command arguments and results. The largest single item of work remaining on the compiler is to compile Tcl expressions and the `expr` command. Most control structure commands use expressions so compiling them should significantly reduce the execution time of almost every script.[1] Other remaining work includes compiling performance-critical commands like `for` and `lindex` into inline sequences of instructions specific to those commands.

The bytecode compiler and interpreter pass most Tcl regression tests. Tests that fail depend on the specific contents of traceback information in error messages or on the exact formatting of the result of list operations. The bytecode system makes fewer recursive calls to `Tcl_Eval` so error tracebacks now have fewer intermediate levels. The new list implementation consistently uses `Tcl_Merge` to regenerate a list object's string representation, while the traditional Tcl system typically ignores portions of strings not directly modified in a list operation. This can lead to such differences as whether sublists are bracketed with braces or quotes: for example, the result of

```
linsert {a b "c c" d e} 3 1
```

with the new system is

```
a b {c c} 1 d e
```

while the traditional system produces

```
a b "c c" 1 d e
```

The Tcl tests will be updated to reflect the new behavior.

1. Expressions are very expensive today. As an experiment, I implemented a new command `untilzero` that repeats a loop while a variable is nonzero. Executing

```
    set x 1000; while {$x>0} {incr x -1}
```

requires 5.1 times more time than

```
    set x 1000; untilzero x {incr x -1}
```

(28661 vs. 5577 usec). These two loops do the same work. The difference is the expression that is reparsed on each iteration. (Although `while` commands are compiled into a sequence of instructions, their expressions are still evaluated today by `Tcl_ExprBoolean`).

Table 1 shows performance results for a few simple benchmarks on a 167MHz UltraSPARC 1.

Table 1: Performance Benchmark Results

Benchmark	Time (usec)		Speedup
	Tcl7.5	New	
null proc with 5 args	64	6	10.6
proc incrementing arg	31	7	4.4
proc of just comments	31	3	10.3
set 20 global variables	210	31	6.7
set 20 local variables	206	21	9.8
incr 20 local variables	368	28	13.1
lindex end of long list	134	3	44.6
linsert end of long list	58	12	4.8
iterative factorial	449	265	1.6
list reverse with while	4985	2376	2.0

Some scripts show significant improvement, especially those that make heavy use of procedure arguments and local variables, ones that manipulate lists, and scripts that benefit from not reparsing. Performance improvements for the larger (and more realistic) benchmarks are modest at this time, largely because expressions are not yet compiled. I expect performance to improve significantly as the remaining compiler and interpreter changes are made.

The next section describes the goals for the bytecode compiler. Section 3 describes the compilation strategy. I present the design of the dual-ported object support next. Section 5 and Section 6 give details about the design of the bytecode compiler and interpreter. Memory requirements for the bytecode system are discussed in Section 7. The current state of the compiler is described in Section 8 while Section 9 discusses related work. I explain next why I do not use Java bytecodes. Section 11 discusses implications of the compiler for script and extension writers. The compiler's development plan is outlined in Section 12.

2 Goals for the Compiler

Besides increased execution speed, the compiler's goals include the following:

- Minimize user-visible changes to scripts. Old scripts must run as before with no or very few changes. It isn't possible to change the Tcl language even if those changes would improve the compiler's effectiveness. The compiler must continue to support Tcl features like traces, `upvar`, and `unset` even though they slow our implementation considerably.
- Continue to support Tcl's dynamic features like rebinding of core commands and runtime-computed variable and command names. However, we may change the behavior of core commands when those changes allow significant performance improvements without effecting the execution of correctly written scripts: scripts that use the semantics documented in the Tcl man pages. For example, list operations will no longer preserve exact white space between list elements.
- Minimize changes to C code in extensions. I expect to do this by providing new parallel core API procedures that contain the changes. Old C code can continue to use the old interface procedures. Those old interface procedures will often be reimplemented in terms of new functionality.
- Minimize storage requirements during both compilation and execution. Tcl's small memory footprint is a significant feature and must be retained.
- Portability. The compiler and interpreter must run on a wide range of platforms. I cannot generate machine code, for example.

3 The Compilation Strategy

The new compilation system relies upon support for dual-ported objects and a new bytecoded compiler and interpreter. Dual-ported objects are passed to command procedures and are stored in variables. Objects contain a string as well as an internal representation. They reduce conversions by holding an appropriate representation such as an array of element pointers for a list. Although objects contain an internal representation, their semantics are defined in terms of strings: an up-to-date string can always be obtained, and any change to the object will be reflected in that string when the object's string value is fetched. Objects are typed. An object's type reflects the set of operations on its internal representation. The set of types is extensible. Several types are predefined in the Tcl core including integer, double, list, and bytecode.

Compilation in the new system is done as needed, or on-the-fly. When a script is evaluated (say as the result of a call to `Tcl_Eval`), it is compiled into bytecodes that are then executed. We use the term *code unit* to describe the collection of bytecode instructions and related information that results from compiling a script. A new Tcl API procedure, `Tcl_EvalObj`, operates much like `Tcl_Eval` to evaluate a script but takes a Tcl object instead of a string; it compiles the object's string value and caches the resulting code unit as its internal representation to avoid later recompilations. The compiler generates instructions for an idealized Tcl virtual machine This machine is stack-based since this allows programs to be represented more compactly; the encoding of most instructions is a single byte. Since programs are compiled, script parsing at execution time is rarely necessary. Some runtime parsing is needed since Tcl scripts can compute new scripts that they later evaluate. Such runtime-created scripts are also compiled on-the-fly. The compiler will eventually generate bytecodes for most of Tcl's core commands. It can make decisions now made by the traditional Tcl interpreter at runtime. For example, the compiler assigns frame offsets to local variables in procedures to avoid the runtime hashtable lookup done for them in the traditional system.

4 Design of dual-ported objects

4.1 The `Tcl_Obj` structure

Dual-ported objects are used throughout the new Tcl system to hold scripts, strings, integers, arrays, lists, etc. For example, command procedures now take an "`objv`" array of pointers to the argument objects. An object has two representations: a string and an internal form. Objects are represented by `Tcl_Obj` structures allocated on the heap.

The definition of the Tcl_Obj structure is shown in Figure 1. This structure is five words: the reference count, a pointer to the object's type structure, a string pointer, and two words used by the type. The string is the object's string representation which is also allocated on the heap. The two words managed by the type hold the object's internal representation: an integer, a double-precision floating point number, two arbitrary words, or a pointer to a value containing additional information needed by the object's type to represent the object. A list object, for example, contains a pointer to a structure with an array of pointers to the objects for the list elements. An integer object contains an integer value.

At least one of an object's representations is valid (non-`NULL`) at any time. Representations are computed lazily, when they are needed. An object that contains only a string and is (so far) untyped has a `NULL` `typePtr`. As an example of the lifetime of an object, consider the following sequence of commands:

```
typedef struct Tcl_Obj {
    int refCount;              /* When 0 the object will be freed. */
    char *string;              /* The Tcl_Obj's string representation. */
    Tcl_ObjType *typePtr;      /* Reflects the object's type. */
    union {                    /* The internal representation. */
        int intValue;          /* -An integer value. */
        double doubleValue;    /* -A double-precision floating value. */
        VOID *otherValuePtr;   /* -Another, type-specific value. */
        struct {               /* -The value as two words (ints). */
            int field1;
            int field2;
        } twoIntValue;
     } internalRep;
} Tcl_Obj;
```

Figure 1: Definition of the Tcl object structure

`% set A 123`

This assigns A to an integer object whose internal representation is the integer 123. Its string representation is left `NULL` to avoid allocating a string on the heap; if the string is needed later, it can be regenerated from the integer[1]. The `typePtr` points to the structure describing the integer type.

`% puts "A is $A"`

A's string representation is needed. It is computed from the object's internal representation. Afterwards, A's internal representation holds the integer 123 and its string representation points to "123". Both representations are now valid.

`% incr A`

The `incr` command increments the object's integer internal representation and invalidates (sets `NULL`) its string representation is since it is no longer valid.

`% puts "A is now $A"`

The string representation of A's object is needed and is recomputed. The string representation now points to "124".

An object's internal form is typically computed on the first type-specific operation, or when an object is converted to a new type. The string is invalidated (set `NULL`) when the internal representation is changed, and vice-versa. The string representation is only regenerated when necessary. For example, the string representation of a `for` loop's index variable will never be recomputed unless it is actually used as a string. I expect that almost all objects will remain a single type, perhaps after an initial conversion.

1. This optimization is possible only when the correct string representation can be regenerated. It can't be used, for example, for the string "000123" since a later command might depend on the leading zero characters.

4.2 Object types

The set of object types is open ended. The Tcl core predefines six object types: integer, double, list, bytecode, boolean, and command name. We expect to create many new types in the future. For example, the Tcl core could use a file pathname type to store a canonical platform-independent representation of a file's path. Also, Tk might use objects to store options for Tk commands.

A Tcl object type is defined by a structure containing pointers to four procedures called by the generic Tcl object code. The definition of this type is shown in Figure 2.

The `Tcl_UpdateStringProc` updates an object's string representation from its internal representation. A type's `Tcl_DupInternalRepProc` and its `Tcl_FreeInternalRepProc`, respectively, duplicate and free an object's internal representation. The final procedure, the `Tcl_SetFromAnyProc`, converts an object from another type by producing this type's internal representation. It can always do this by first updating the object's string representation (if necessary) then generating the internal representation from the string. However, the `Tcl_SetFromAnyProcs` for most object types include special case conversions from some number of other types. An example is the double type's `Tcl_SetFromAnyProc`. This supports faster integer to double conversions by directly converting the integer that is an integer object's internal representation to a double-precision floating point number; it does not regenerate the string representation and then parse it.

As an important optimization, an empty string is represented by an object with a `NULL` string pointer and

```
typedef int (Tcl_SetFromAnyProc)      (Tcl_Interp *interp, Tcl_Obj *objPtr);
typedef void(Tcl_UpdateStringProc)    (Tcl_Interp *interp, Tcl_Obj *objPtr);
typedef void(Tcl_DupInternalRepProc)  (Tcl_Obj *srcPtr, Tcl_Obj *dupPtr);
typedef void(Tcl_FreeInternalRepProc) (Tcl_Obj *objPtr);

typedef struct Tcl_ObjType {
    char *name;       /* Name of the object type, e.g. "int" or "list". */
    Tcl_FreeInternalRepProc *freeIntRepProc;
                      /* Frees any storage for the type's internal representation. */
    Tcl_DupInternalRepProc *dupIntRepProc;
                      /* Creates a new object as a copy of an existing object. */
    Tcl_UpdateStringProc *updateStringProc;
                      /* Updates the string rep. from the type's internal rep. */
   Tcl_SetFromAnyProc *setFromAnyProc;
                      /* Converts the object's old internal rep. to this type. */
} Tcl_ObjType;
```

`typePtr`. Empty strings are common and this optimization helps to reduce storage requirements.

The list type maintains for each list object an array of pointers to the Tcl objects that represent the list's elements. This internal representation allows for fast indexing and append operations (which we believe to be the most common) at the expense of slightly slower insertions into the middle of a list. For example, `lindex` is now a constant time operation; extracting the last element of a list now requires only 3 usec regardless of the list's length while Tcl7.5 takes 15 usec for a 10 element list, 37 usec for a 40 element list, and 72 usec for a 100 element list. `linsert` is also faster; inserting an element at the end of a 60 element list is 4.8 times faster (12 vs. 58 usec).

The element array of a list is initially allocated just large enough to hold the list's elements. However, if a list is grown by, say, an append operation, a new array is allocated that is larger than is actually required by the operation. This overallocation improves the speed of subsequent append or insertion operations. When the list type's `Tcl_SetFromAnyProc` generates the internal representation for a list, it parses the entire list. This means that operations on some lists will fail in the new system that would have succeeded in Tcl7.5: if a list has a syntax error after the elements being operated on, the new system will return an error message where Tcl7.5 would have ignored the bad syntax.

The command name object type is used by the bytecode interpreter to cache the result of command hashtable lookups. Hashtable lookups are expensive (about 1 usec on a UltraSPARC 1, or the same time needed to set a local variable in the bytecode system), so avoiding them on most command invocations significantly improves execution time.

4.3 Storage management of objects

Tcl objects are allocated on the heap. A custom allocator reduces the cost of allocating and freeing objects by maintaining a private list of available free objects.

Because many objects are simply passed as arguments to called procedures, objects are shared as much as possible. This significantly reduces storage requirements because some objects such as long lists are very large. Also, most Tcl values are only read and never modified. This is especially true for procedure arguments, and argument objects can be shared between the caller and the called procedure. Assignment and argument binding is done by simply assigning a pointer to the value. It isn't necessary to copy (and allocate storage for) the entire value. But this raises the problem of knowing when it is safe to free an object. I use reference counting to determine when it is safe to deallocate an object; an object can be freed when the number of references to it drops to zero. I can't use a garbage collector because it would increase Tcl code and runtime memory usage too much.

One advantage of reference counts is that they support an important optimization called *copy-on-write*. Since objects are shared, a new copy must be made before modifying an object. But if an object is unshared–that is, if it has a reference count of one–the object can be modified directly without having to make a copy. Copy on write reduces storage requirements and execution time.

```
push1 <1 byte index>                      loadStk
push4 <4 byte index>                      storeStk
pop                                       incrScalar1 <1 byte index>
concat <1 byte count>                     incrScalarStk
invokeStk1 <1 byte argument count>        incrArray1 <1 byte index>
invokeStk4 <4 byte argument count>        incrArrayStk
loadScalar1 <1 byte index>                incrStk
loadScalar4 <4 byte index>                incrScalar1Imm <1 byte index> <incr byte>
loadScalarStk                             incrScalarStkImm <signed incr byte>
storeScalar1 <1 byte index>               incrArray1Imm <1 byte index> <incr byte>
storeScalar4 <4 byte index>               incrArrayStkImm <signed incr byte>
storeScalarStk                            incrStkImm <signed incr byte>
loadArray1 <1 byte index>                 evalStk
loadArray4 <4 byte index>                 jump1 <1 byte signed distance>
loadArrayStk                              jump4 <4 byte signed distance>
storeArray1 <1 byte index>                jumpFalse1 <1 byte signed distance>
storeArray4 <4 byte index>                jumpFalse4 <4 byte signed distance>
storeArrayStk                             done
```

Figure 3: The current bytecode instructions

5 Design of the bytecode compiler

The compiler is single pass to minimize compilation time. It uses a recursive descent parser that emits instructions for each command as it is parsed.

To hold information needed during compilation, the compiler uses a compilation environment (`CompileEnv`) structure. This holds a code unit's instructions, *object table*, and command location map. The object table is an array of pointers to Tcl objects referenced by instructions. The table has an object for every unique constant in the script that is not "compiled away": for example, the string `"A is "` needed for the command `puts "A is $A"` above is represented by an object table entry. The command location map has source and bytecode location information for each command. This information is used, for example, to find the source command for a bytecode location. The `CompileEnv` structure also contains a pointer to the current procedure's `Proc` structure (if any) to compile references to local variables, and contains fields that describe the length and other properties of the last command word processed. The `CompileEnv` structure is allocated on the C stack and is large enough to hold the instructions and other information for almost all Tcl scripts. This use of stack-allocated space minimizes the number of costly heap allocations. When compilation is finished, a single heap object is allocated to hold the subset of information required to execute the script.

In order to generate instructions for a command, the compiler first checks whether a compile procedure (`CompileProc`) has been registered for it. This is done just after the command's first word is parsed. If a `CompileProc` is found, it is called to generate code for the command. If no `CompileProc` is found, or if the first word involves substitutions that can only be computed at runtime, the compiler emits code to invoke the command's command procedure at execution time. `CompileProcs` exist today for the `set`, `while`, and `incr` commands. Eventually `CompileProcs` will be registered for most core Tcl commands.

At this time, the compiler emits the 35 instructions listed in Figure 3. Some of these implement variable substitutions and the Tcl commands `set`, `incr`, and `while`. The remainder do the work of the traditional Tcl parser by pushing and popping objects, concatenating strings, and calling command procedures. New instructions will be added as more commands are compiled. I expect also that the instruction set will change as I get more experience with the bytecode system.

Most instructions operate on an *evaluation stack*. This stack is separate from the "stack" of Tcl procedure call frames and is also separate from the C call stack. The evaluation stack holds pointers to Tcl objects holding command arguments and results. Each Tcl interpreter has its own evaluation stack. The compiler computes the maximum stack depth needed for each code unit and the interpreter, when starting to execute a code unit, ensures that it has enough stack space. This avoids checking on each instruction whether the stack needs to be grown.

Instructions consist of an opcode byte followed by zero or more operands. Operands are one or four byte integers or indexes. As an example, `push1 <index>` pushes an object onto the evaluation stack. The one byte index refers to one of the first 256 objects in the code unit's object table. Several instructions have four byte variants to support large scripts, while the one byte variants keep the code for small scripts small. Instructions whose names include the “Stk” suffix take an operand from the evaluation stack.

To make local variables faster, the compiler assigns each local variable an entry in an array of variables stored in a procedure’s call frame. This avoids an hashtable lookup on each reference. The compiler also determines whether the variable name refers to a scalar or an array element. These two changes alone make local variable access faster by a factor of 9.5! (From 201 usec to 21 to set 20 locals. Other changes account for a 5 usec improvement.)

Some variables are only created (computed) at runtime. For example, the command `set [gensym] 123` assigns a value to the variable whose name is returned by the procedure `gensym`. To support these runtime computed variables, the compiler emits the instructions `loadStk` and `storeStk` that take the variable name from the top of the evaluation stack.

5.1 Examples of compiled code

Compiling the procedure

```
proc while_1000x {} {
    set x 0
    while {$x<1000} {
        incr x
    }
}
```

generates a code unit with the instructions

```
# set x 0
0  push1 0           # push object "0"
2  storeScalar1 0    # store into local x
4  pop               # discard value
# while {$x<1000} {\n incr x\n }
5  push1 1           # push "$x<1000"
7  jumpFalse1 8      # false => goto pc 15
# incr x
9  incrScalar1Imm 0,1# increment local x
12 pop               # discard value
13 jump1 -8          # goto pc 5
15 push1 2           # while result is ""
17 done
```

The number at the left of each instruction is its bytecode offset. The `push1 1` instruction at offset 5 pushes a string object containing `"$x<1000"`; the instruction’s operand specifies the second object in the code unit’s object table. This string is passed to the Tcl expression code at runtime since expressions are not yet compiled. The `storeScalar1 0` at offset 2 stores the object at the top of the evaluation stack into the scalar local variable at offset 0 in the call frame’s array of local variables.

This procedure currently runs 1.4 times faster with the bytecoded system than in Tcl 7.5 (26954 vs. 38550 usec). I expect this performance to improve when expressions are compiled.

As a more complex example, the procedure

```
proc lreverse_with_while {a} {
    set b ""
    set i [expr [llength $a] -1]
    while {$i >= 0} {
        lappend b [lindex $a $i]
        incr i -1
    }
    return $b
}
```

generates the instructions

```
# set b ""
0  push1 0           # push ""
2  storeScalar1 1    # store into local b
4  pop
# set i [expr [llength $a] -1]
5  push1 1           # push "expr"
7  push1 2           # push "llength"
9  loadScalar1 0     # load local a
11 invokeStk1 2      # call llength,2 args
13 push1 3           # push integer obj -1
15 invokeStk1 3      # call expr, 3 args
17 storeScalar1 2    # store into local i
19 pop
# while {$i >= 0} {\n lappend b [lindex ...
20 push1 4           # push "$i >= 0"
22 jumpFalse1 23     # false=>goto pc 45
# lappend b [lindex $a $i]
24 push1 5           # push "lappend"
26 push1 6           # push "b"
28 push1 7           # push "lindex"
30 loadScalar1 0     # load local a
32 loadScalar1 2     # load local i
34 invokeStk1 3      # call lindex, 3 args
36 invokeStk1 3      # call lappend,3 args
38 pop
# incr i -1
39 incrScalar1Imm 2,-1
42 pop
43 jump1 -23         # goto pc 20
45 push1 0           # push ""
47 pop
# return $b
48 push1 8           # push "return"
50 loadScalar1 1     # load local b
52 invokeStk1 2      # call return, 2 args
54 done
```

Here the `invokeStk1` instructions are used to invoke command procedures at runtime. In the next few months, the compiler will be modified to emit command-specific instructions inline for most Tcl core commands. This procedure runs 2.0 times faster with the current bytecoded system than in Tcl 7.5 (2376 vs. 4985 usec for a 60 element list).

5.2 Some compilation problems

a) Variables must be accessed in the correct order

Compiled code must read, write, and delete variables in the correct order. This is because traces must run the correct number of times and in the correct order. Consider the following example:

```
expr{$a} + $b || {$c} + $d
```

The variables must be read in the order `b`, `d`, `a`, then `c`.

In the traditional Tcl system, the interpreter reads variables `b` and `d` when substituting their values. When `expr` is called, it does a second round of substitutions on its arguments itself, and so reads the variables `a` and `c`. The order in which variables are read is shown above. Compiled code must read the variables in the same order. I may alter the variable read, write, and delete behavior of some operations to improve the implementation, but I will only do this if the changes do not modify the semantics of those operations or of the `trace` command as described in the Tcl man pages. For example, the traditional Tcl interpreter implements `lappend` using `Tcl_SetVar2` to append each new list element. This triggers read and write traces for each appended element. I may compile code to append the new items all at once and run the traces a single time.

b) `expr`*'s substitutions can change the apparent expression*

As described above, `expr` does a second round of substitutions on its arguments. This can make the expression's apparent interpretation and the obvious code wrong. Consider the following:

```
% set x 2
% set y {$x+5}
% expr $y*15
=> 77 —this is the correct result but it is
not divisible by 15!
```

From the expression `$y*15` it looks like the final result is a multiple of 15, but this is wrong. `expr` is passed `$x+5*15`, which after `expr`'s second round of substitutions becomes `2+5*15` or 77.

This problem only happens when `expr` does a second round of substitutions. If `expr`'s argument is not enclosed in braces, the best I can do is to generate "optimistic" code for the apparent expression and check at runtime whether this code might be wrong. It can only be wrong if variables substituted in the first round require more substitutions in the second round. Typically this isn't the case and the interpreter can execute the compiled code. Otherwise, the interpreter needs to back off and invoke `expr` to interpret the expression.

If `expr`'s argument is enclosed in braces, the apparent code is always correct and the test can be dropped. So, expressions protected by braces will execute *faster*. This includes expressions used in `if`, `while`, and other control structure commands.

c) Global variables may not be truly global

In the same way that it currently does for local variables, the compiler could assign each global variable an index in the table of globals and use this index in instructions. It can only do this for variables, however, which it knows to be truly global. Tcl lets a `global` command appear anywhere, including after the use of a local variable with the same name. This is an error, and must be reported as such, so the compiler can only "compile away" global variables known to be global. It can safely do this for `global` commands that appear at the top of a procedure, which is the usual location anyway. Those that appear elsewhere will have to be implemented by a `global` instruction that will do the appropriate checking. This means that `global` commands placed at the top of procedures will be faster.

6 Design of the bytecode interpreter

The bytecode interpreter uses a traditional while loop that switches on the opcode of each instruction:

```
for (;;) {
    opCode = *pc;
    switch (opCode) {
        case INST_INVOKE1:
        ...
}
```

I checked first whether an alternative implementation would be faster. This used an array of procedure pointers, indexed by opcode, to implement each instruction. However, this proved about 20% slower, independent of machine or compiler.

The compiler emits a `done` instruction to terminate the main interpreter loop if no `return` or `error` command is executed. This instruction trades space for time and avoids the need to continually test for the last instruction.

The evaluation stack holds arguments for commands. When invoking a command procedure, the procedure's `objv` array (the array of pointers to argument objects) is set to the address of the evaluation stack element holding a pointer to the object with the command name; no pointer copying is needed. The interpreter caches a pointer to the top of the stack in a local variable.

If a Tcl program redefines a core command, any code that uses that command must be invalidated. To implement this, the interpreter increments a counter, the *compilation epoch*, whenever a core command is redefined. When a script is compiled, the current compilation epoch is stored in its code unit. Before executing a code unit, the bytecode interpreter checks whether the code unit's epoch matches the current epoch. If not, the interpreter discards the code unit and recompiles its script.

I have reimplemented the command procedures for most commands to be object-based: that is, to take an `objv` array and to return an object result. These object-based command procedures are called directly by the bytecode interpreter. The remaining string-based command procedures are implemented using a wrapper procedure. This wrapper generates an `argv` string array from the string representations for the argument objects, calls the string command procedure, and constructs a string object holding the result. I expect eventually to make all command procedures object-based.

7 Memory requirements for the bytecode system

Strings are a compact way to represent Tcl scripts: no separate instructions or other data representations are needed. The bytecode system improves the speed of executing Tcl scripts at the cost of additional storage for code units and dual-ported objects. How much additional memory is needed?

The body for the procedure `while_1000x` in Section 5.1 is 56 characters. Its code unit requires 18 instruction bytes. Its object table contains pointers to three Tcl objects: an integer object for the source string `"0"` (for which no string is allocated on the heap), an object pointing to `"$X<1000"`, and an empty object representing the result of the while command. Since each Tcl object requires five words, the object table requires 80 bytes including the storage for the one heap string. The command location table for this procedure's three commands requires 3 entries of 4 words each, or 48 bytes. So, the total memory for this procedure's code unit is (18 + 80 + 48) or 146 bytes[1], 2.6 times the storage for just the source characters.

The body for the second procedure in Section 5.1, `lreverse_with_while`, is 131 characters. Its code unit requires 55 instruction bytes. Its object table has nine objects (for `""`, `"expr"`, `"llength"`, `"-1"`, `"$i >= 0"`, `"lappend"`, `"b"`, `"lindex"`, and `"return"`) and requires a total of 261 bytes. There are six commands so the command location table requires 96 bytes. The total memory for this procedure is then (55 + 261 + 96) = 412 bytes, or 3.1 times the source size.

Note that five of the nine objects for this code unit were allocated just to hold the names of commands to be invoked by `invokeStk1` commands. One of the main benefits of compiling commands into inline sequences of command-specific instructions may be to reduce the storage needed for programs. In this case, removing just those command name objects would save 155 bytes! The count of instruction bytes would increase a little, but the code unit's storage would still drop to approximately 1.9 times that of the source.

These memory results are preliminary. The actual storage needed for Tcl scripts will change as more commands are compiled inline. I expect to look for further opportunities to reduce memory requirements. For example, it should be possible to find a more compact representation for the command location tables.

8 Compiler status

At this time (May 1996), the basic support and infrastructure for the new bytecode system is complete. The dual-ported object support is finished. The Tcl core implements six object types. Objects are passed to and returned by command procedures and are stored in variables. The compiler emits inline instructions for several key instructions. Support routines exist that allow the development of new `CompileProc`s for commands to be added at the rate of about one a day. The largest remaining item of work is to compile Tcl expressions. Another large work item is to support Tcl namespaces. The specific functionality for namespaces has not been decided, but it will probably be similar to George Howlett's proposal [Howlett94] and Michael McLennan's [incr Tcl] namespace support [McLennan95].

Performance improvements to date are modest for most code: about a factor of two for scripts that use expressions or control structures (since they use expressions). The fact that performance is significantly faster for scripts that make heavy use of variables or lists is promising. The key reasons performance isn't better for all scripts yet include:

- Few commands have command-specific instructions generated for them. A procedure call to a command procedure is still being made for most commands. Also, many objects are pushed, popped, have their

1. This ignores any overhead words required by the heap implementation.

reference counts incremented and decremented just to fabricate the arguments for the command procedures. Appropriate instructions for each command will reduce this significantly.

- `expr` isn't compiled yet. The `expr` command is itself used often and expressions are used in many control structure commands.
- Too many little code units are compiled and executed. This is primarily because control structure commands are not yet directly compiled into a linear sequence of instructions. As an example, an `if` command's `then` and any `else` subcommands are compiled separately, and are executed when the `if` command's command procedure recursively calls `Tcl_EvalObj` on the bytecode objects for their scripts. This results in extra procedure calls and execution overhead as well as extra storage use. This will improve when instructions for those subcommands are emitted inline.
- Repeated compilations. This is because some command procedures are still string-based and can't save the bytecodes of a compiled subcommand in an object. Consider the following:

```
expr {$n*[llength $a]}
```

This is slowed today because the nested command `llength $a` is recompiled, executed, and its bytecode deallocated each time the `expr` is evaluated. This is because `expr` does command substitutions on its arguments, and recursively calls `Tcl_Eval`. This, in turn, compiles the expression but the code unit resulting from the compilation is discarded afterwards. This is a temporary problem that will end when expressions are directly compiled.

- Compilation is expensive at the moment. The cost of compiling `lappend pkgs "stdPkg"` is about twice that of executing it once. Most of the compilation time is spent allocating objects for each word or part of a word in a script. In this script these are the words "`lappend`", "`pkgs`", and "`stdPkg`". When the compiler emits command-specific instructions, most of these allocations will disappear. But even now, the compilation time for most realistic scripts is only a small part of their execution time: a recursive factorial procedure computing the factorial of five spends only 1% of its time compiling.

9 Related work

Adam Sah's TC system [Sah94] provided a speedup of about 5-10 over traditional Tcl. His system introduced the use of dual-ported objects. TC implemented lists using arrays of pointers much as I do. It also used reference counts to decide when to free objects. Like our system TC used reference counts to implement copy-on-write. This minimized copying in procedure calls and saved a considerable amount of storage. In an attempt to reduce the cost of script execution, TC statically preparsed scripts. This did not benefit most Tcl/Tk scripts since most scripts require runtime parsing. Because of this, he suggested instead caching the result of parsing, which is effectively what our system does. His system also implemented several other optimizations, including implementing procedure frames as arrays and compiling variable references into indexes. Unfortunately, Adam Sah never released TC.

Forest Rouse and Wayne Christopher developed the ICE Tcl compiler [Rouse95] that is available from ICEM CFD Engineering. This compiler translates Tcl to C code, which is then compiled. It speeds up typical Tcl/Tk applications by a factor of between 5 and 20. ICE Tcl tracks the dynamic types of Tcl variables in C code using a mechanism similar to our object system. Its `Tcl_Var` structure has fields for integer, double, list, and string representations and includes a flag word that indicates which of these representations is valid. Unlike our system, more than two representations may be valid at any time. This offers the potential for improved speed at the cost of additional memory, greater complexity, and more difficult use. One drawback of translating to C is the significant expansion in application code size (a factor of 20-30 in some cases) and complexity of application development. ICEM has announced plans to develop a bytecode compiler to avoid these problems.

10 Why not use Java bytecodes?

If it proved feasible, using Java bytecodes to implement Tcl would have a number of advantages. The Java virtual machine is widely available (e.g., in the Netscape browser). Using Java bytecodes might also simplify interoperation between Tcl and Java code.

Unfortunately, using the Java virtual machine would be too slow or take too much memory, at least with current Java interpreters. The basic problem is the semantic mismatch between Java bytecodes and Tcl. Consider the Tcl `set` command. Tcl variables behave very differently than Java variables. I can't use a Java instruction like `astore` (store object reference in local variable) to store a Tcl value into a Tcl variable since it doesn't handle by itself such Tcl details as variable traces, `unset`, or `global`. The best I could do would be to translate a Tcl set command into a sequence of several Java instructions that did the appropriate checks. Unfortunately, the number of Java instructions to implement each Tcl command would make the compiled program too big. A more realistic scheme is to generate Java bytecodes that call one or more Java methods to do the actual work for each Tcl command. With this number of Java method calls, acceptable performance would depend on using a Java machine code compiler. But these compilers won't be free.

Another problem is that much of the interesting code in Tcl/Tk and its extensions is in C. Java code can call "native" methods implemented in C, and vice-versa, but this is awkward and the capability is disabled in Netscape (and probably most other Java implementations) for safety reasons.

11 Implications for current script and extension writers

11.1 Implications for scripts

Use lists. They are now even faster than arrays since indexing elements requires no hashtable lookup.

You should not rely on the string representations of lists having a particular syntax. That is, you should use list operations like `lindex` to manipulate lists. Also, list operations will now parse the entire list when converting an object to a list. `lappend`, for example, no longer ignores arbitrary text in the list it is appending an element to. This means that you shouldn't use list operations to manipulate values that aren't lists. Use string operations to manipulate arbitrary strings.

Use braces around expressions, including those used in control structure commands. This lets us generate inline instructions to evaluate the expression without the need to check for second-level substitutions that might invalidate the code.

Put all `global` commands at the start of procedures.

The execution traceback information in error messages will change. Since the compiler will generate inline instructions for what currently are recursive calls to `Tcl_Eval`, error tracebacks will be somewhat flattened. They should be more understandable, however, especially since I should be able to include source line numbers.

11.2 Implications for extension C code

Convert string-based command procedures to use objects. These will execute faster and will be able to take advantage of type-specific operations that operate on internal representations appropriate for those types.

As described above, don't use the list API procedures to operate on values that aren't lists and don't rely on them preserving white space between list items.

12 Future work

A substantial amount of work remains on the compiler. This includes the changes described above, in particular:

- Compile inline code for `expr` commands.
- Implement the remaining changes for Tcl variables. This includes compiler-allocated entries for global variables, and better `global`, `unset`, `uplevel` and `upvar` support.
- Generate inline instructions for high payoff commands. For example, I won't immediately compile the `clock` or `history` commands.
- Add namespace support.

I expect to release an initial version of the bytecode compiler about four months from now.

I have no plans at this time to do type inference for Tcl expressions as done by David Koski [Koski95] and Guy Steele [Steele94]. This can be very effective—David Koski got speedups of more than a 1000 for some floating point intensive Tcl code—but type inference is difficult to do correctly in a language as dynamic as Tcl.

13 Conclusion

I have described the design and current state of an on-the-fly bytecode compiler for Tcl. I expect this compiler to eventually improve the speed of compute-intensive Tcl scripts by a factor of about 10. Part of the compiler's speedup derives from its use and support for dual-ported objects. Early results with the compiler are promising but considerable work remains.

14 References

[Howlett94] Howlett, George. "Packages: Adding Namespaces to Tcl." Proceedings of the 1994 Tcl/Tk Workshop, New Orleans, Louisana, June 1994.

[Koski95] Koski, David. "A Tcl Compiler." Unpublished class project report, University of Wisconsin, October 1995.

[McLennan95] McLennan, Michael. "The New [incr Tcl]: Objects, Mega-Widgets, Namespaces and More." Proceedings of the 1995 Tcl/Tk Workshop, Toronto, Canada, July 1995. Also described by the web pages at `http://www.tcltk.com/itcl/namesp.html`.

[Rouse95] Rouse, Forest R. and Christopher, Wayne. "A Tcl to C Compiler." Proceedings of the 1995 Tcl/Tk Workshop, Toronto, Canada, July 1995. A commercial version is described by `http://icemcfd.com/tcl/ice.html`.

[Sah94] Sah, Adam. "TC: An Efficient Implementation of the Tcl Language." Master's Thesis, UC Berkeley Dept. of Computer Science report UCB-CSD-94-812. 1994.

[Steele94] Steele, Guy. Unpublished Common Lisp program that translates a subset of Tcl to C.

[Welch95] Welch, Brent. "Customization and Flexibility in the exmh Mail User Interface." Proceedings of the 1995 Tcl/Tk Workshop, Toronto, Canada, July 1995.