TC: An Efficient Implementation of the Tcl Language

Adam Sah

Abstract

Tcl is a highly dynamic language that is especially challenging to execute efficiently. In this paper, I discuss many issues involved in implementing Tcl, and describe a design for a faster system that maintains Tcl semantics, including its C callout mechanism. This design focuses on a method for caching the parsed representation for data values, and lazily converting to strings on demand. This allows most computations to be performed using native types (eg. integers) rather than strings. The current implementation is presented along with results showing a speedup of about 5-10 times over the existing Tcl interpreter.

submitted in partial fulfillment of the Masters Degree in Computer Science at the University of California at Berkeley.

First Reader: John Ousterhout

Second Reader: Paul Hilfinger

This work supported in part by grants from the Hewlett-Packard Corp. under grant #M1308 and a grant from the California MICRO program.

Note: copies of this report and related software are publicly available. Send mail to asah@cs.berkeley.edu for more information, or ftp to ftp.cs.berkeley.edu:/ucb/tcl.

1 Introduction

1.1 Motivation

In the past five years, the Tcl scripting language[Ous93] has gained enormous popularity. As with all such software, this popularity has inspired users to extend the range of Tcl's uses. In particular, longer scripts and scripts with large data structures present a performance challenge that has not been met adequately by the current interpreter.

This "success disaster" leads current users to rewrite sections of Tcl scripts in C, which is about three orders of magnitude faster. This situation is made possible by the C callback mechanism, in which an arbitrary C function can be bound to a Tcl command name. A common approach is to rewrite time-consuming Tcl procedures in C and bind them to the original procedure name. For example, Braverman used this approach to build object oriented extensions into Tcl[Bra93]. Doing this, however, forgoes many of the benefits of using Tcl, including the easy interoperatability with other software systems, ease of maintenance, and the benefits afforded by incremental development as provided by the interpeter.

The performance problems that lead users to rewrite Tcl commands in C stem from the current implementation in which all objects are stored as strings, including both code and data. For example, in the case of code, the `for` command stores the body of the loop as a string, which necessitates its reparsing on each iteration. The same problem is exhibited in data access. For example, indexed lookup into Tcl lists requires a linear parse and scan algorithm because lists are stored only as strings whose whitespace delimits elements.

I have developed an alternate approach where data is stored in a more convenient form in a way that's compatible with Tcl's "everything is a string" model. In this design, we recognize when Tcl requires a value in its string form and convert from our storage form into the value's string form. These *implicit conversions* are used for the features in Tcl that require strings, such as string concatenation of values and the C callout mechanism. I have implemented this approach in a Tcl interpreter called TC.

My thesis argues that such a system will perform favorably against a pure-string system and that this new system will maintain Tcl semantics. The performance claim requires an argument that such string conversions are a rare case. In addition to this semantic argument, I present empirical evidence from some sample scripts showing sizeable improvements for operations on large data objects, repeated execution of scripts (ie. function calls), and operations over data types that have efficient representations in C. Commands that inherently neces-

sitate string-based implementation perform less favorably over Tcl.

The remainder of this paper describes the implementation of this scheme, notably its value-caching system. In section 2, I discuss Tcl's evaluation semantics and introduce TC's evaluation algorithm. Section 3 then proposes a design for a caching data structure. Section 4 discusses the syntactic and semantic issues in parsing Tcl scripts. Section 5 describes the resolution of the memory management issues of this new system. In section 6, I begin the discussion of the secondary design decisions with the details on procedure calls and variable access. Section 7 completes the design with a discussion of the various optimizations made on a per-type basis. Section 8 then validates this thesis with a performance analysis of the resulting system. Finally, section 9 provides an analysis of this work and draws conclusions based on the results. Appendices are provided discussing additional semantic issues, future work, and the source code to the test suites used to measure performance.

1.2 An Overview of the Tcl Language

Tcl[Ous93] was designed to address the need for a "scripting" language, providing high-level control over a program with simple, syntax resembling the Unix shell[Bou78]. In Tcl, every statement can be thought of as a function call, in the form "cmd arg arg arg ...", where the first word of the command selects a procedure to invoke and the subsequent words are string arguments to that procedure. Relative to implementation, there are three important issues in Tcl: argument evaluation as substitution, string storage and semantics, and the C callout mechanism.

In Tcl, argument evaluation consists of string substitution. Each command procedure needs to know how to handle fulyl substituted string arguments. Substitution is denoted by special character symbols in the source text: square brackets denote nested commands; dollar signs indicate variable substitutions; curly braces group text into single arguments without performing substitution. See figure 1 for a sample script.

```
# 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

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

# calling our new factorial function
fact 7
```

Figure 1: a sample Tcl script. Each of the whitespace-separated arguments to commands is subject to substitution as per Tcl rules. For example, the curly braces in the procedure definition are what delay the substitution of the arguments in the statements in the procedure body.

In keeping with this string substitution model, in Tcl

everything is a string. In practice, this means that all objects have string representations, including all data objects, intermediate results and code. The use of strings to denote data objects implies a single "type" for all data structures. Of course, not all operations are legal on all strings; for example, multiplying "abc" by "5" has no meaning and results in a dynamic typecheck failure. In Tcl, type correctness means that each primitive procedure called is able to parse its arguments.

Programmers can create new Tcl commands by defining a C command procedure for each command and calling the Tcl runtime library to associate the command procedure with the particular name. When one of these new commands is called in a Tcl script, the interpreter will invoke the associated C function, passing it the arguments as an array of strings. Indeed, this C callout mechanism is used to implement **all** of the built in command primitives in Tcl, including control-flow constructs. For example, the `if` command is bound to a C routine that takes strings as arguments, which it treats as the predicate, the "then" clause, and (optionally) the "else" clause. Depending on the truth value of the predicate when evaluated, `if` will then evaluate one of the clauses as Tcl scripts.

This turns out to be a delightfully simple solution because of the rich runtime library written in C. At first, it seems like every callback requires all of the machinery needed to look up variables and otherwise interact with the Tcl virtual machine. However, such duplicated code is avoided because the Tcl runtime library provides a function call interface for common operations such as variable lookup (eg. Tcl_GetInt() for integer parsing, Tcl_Eval() for nested statement evaluation, and so on). Thus, the C programmer has the tools needed to easily bundle a set of functionality into a command embedded in Tcl.

2 Evaluation in Tcl and TC

2.1 Evaluation Semantics of Tcl

Before delving into the guts of the TC design, it is important to first understand the semantics of Tcl evaluation. These semantics are based on string substitution of the arguments to commands, giving the language its string-centric bias. For example, if "a" is set to "5" and b is set to "6", then `incr a $b` will mutate the Tcl variable "a" to "11". `incr` is bound to a C function that the interpreter calls, passing it an array of strings, in this case the three element array of "incr", "a", and "11". Thus, it is the interpreter's role to hide the substitution of values from the C callouts that operate over strings.

In the original implementation, the Tcl7.x interpreter breaks this example statement into three arguments, “incr”, “a” and “$b”. Then, it evaluates each one. The first two are constant strings that evaluate to themselves, but the third requires variable substitution. “b” is located in a table of variables and its value is substituted. This substitution takes place into a vector of strings that the interpreter is building, called the *argv array*, that holds an array of string values representing the arguments to the current command. The argv array in the previous example would have the values “incr”, “a” and “6”. The interpreter treats argv[0] as a command and searches for it in the table of commands. The interpreter then calls the C function associated with `incr`. The command procedure implementing “incr” then examines the arguments it is passed and parses them. For argv[1] (“a”), it looks this up as a variable and retrieves its string value. Parsing this value as an integer, it increments this value by the value in argv[2] (also parsed as an integer). The resultant integer is then converted back into a string. This string value is stored in the variable and also returned to the interpreter as the result of the statement. In the case of nested commands (eg. `set a [incr a $b]`), this result is substituted into the argv[] array of the outer statement.

2.2 Profiling Tcl: The Motivation and High-Level Design of TC

The source script:

`time {incr a 5}` <iters>

Within Tcl_Eval():

82%	Tcl_IncrCmd()	the command procedure.
5%	Tcl_FindHashEntry()	command (argv[0]) lookup.
12%	Tcl_Eval() itself	parse,create of argc/argv.
1%	Tcl_Eval()	other (ie. reset results).

Total string handling and parsing overhead: 17%

Within Tcl_IncrCmd(): (the C callback bound to `incr`)

30%	sprintf()	post-increment integer->string conversion.
30%	Tcl_SetVar()	parsing, lookup and setting of variable.
18%	Tcl_GetVar()	parsing and lookup of old value.
14%	Tcl_GetInt()	parsing arguments as integers.
8%	Tcl_IncrCmd()	other (ie. setting result).

Total string handling and parsing overhead within callback: 92%

Total string handling and parsing overhead: 0.17 + 0.816*(.92) = **92%**

Potential speedup for incr: 1.0 / (1.0-.92) = **12.5x**

Figure 2: Profiling a simple Tcl script. This script executes the incr command repreatedly. The explanations are derived from the profile and from examining the original source. Most of the time processing this command is spent in parsing and manipulating strings.

To better understand Tcl's current implementation, I profiled Tcl running a simple script. The results are shown in figure 2. Much of the execution time is spent converting to and from strings, including the parsing and lookup of variable names and commands (includes hashing of the string). Clearly, a large performance gain may be had if operations could be performed on the native machine types, and if virtual machine entities like variables could be accessed more efficiently. It should be possible to cache parsed entities in the common case where they won't change, as in statement parsing.

In TC, the evaluation algorithm described above is modified to cache values (ie. the strings that are passed in the argv array). To accomplish this, TC converts values to the native forms on first usage, and only converts back to string form when needed. On the second evaluation of the example statement `incr a $b`, the *value* of "a" arrives to the command procedure already parsed as an integer, as is the value of "b" (assuming it hasn't changed). Likewise, preparsing means that "incr" has already been found in the command table, "a" itself (not a's value) is already parsed as a variable name and the "$b" is already parsed as a variable substitution.

3 The Dual-Ported Object System

In order to provide both native type storage and compatibility with Tcl, TC stores values in an object structure containing both a string representation and a typed, parsed representation ("native representation"). At any given time, one or both forms will be valid; TC treats each form as a cache of the other and converts the native forms to string form on demand. Thus, TC can mimic Tcl semantics by converting values to strings when needed, with the tradeoff being an extra layer of indirection to access the string representation. Since time-critical computations are often performed using non-string forms, this leads to a speed improvement.

To make this scheme work, the native representation must be typed because different functions are needed to convert the native type to and from string form. For example, if the type is integer, then the native value is stored as a C integer and will be converted from string form (parsed) using a function like atoi(). This type tagging also implies another distinction of values in TC: the same value can be represented in multiple ways, including different "types". For example, the integer 5 is equivalent to the string "5".

In fact, this approach allows an object to have different native representations at different times. For example, figure 3 shows a series of script statements and the corresponding state of the value of the variable "a". When the object value is first assigned a value, only the string form is valid. In response to the typed read request

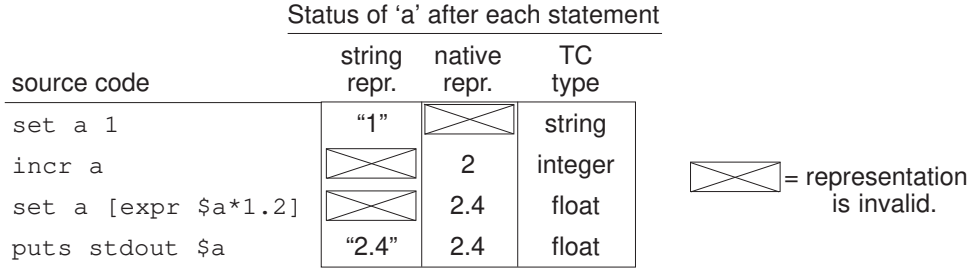

Figure 3. A sequence of Tcl statements and how they affect the representation of the value of 'a'. The statements are shown in order of execution: in the first, `set` assigns a string value to a. `incr` converts this to an integer, increments, and stores it back as an integer. In the third statement, the math expression uses this value (*direct conversion*) as a floating point number to perform the multiplication. Lastly, printing to the screen requires the string representation, which does not change the still valid binary representation. TC's lazy conversion scheme saves conversions in statements two (integer-->string) and three (string-->float).

in the `incr` command, a's value is converted to integer form. `incr` increments this value and stores it back in its native form, invalidating the string value. At this point, further uses of the integer form will require no conversions, affording TC a performance advantage over Tcl7.x, whose `incr` command parses the arguments into integer each time it is executed.

I believe that data values tend to either be read and written as a single type or read only as a single type and as a string. In these two cases, TC can provide the type needed for the computation without converting from one form to another. Intuitively, these two cases are the common case because most programmers work in statically typed languages without such conversions. For example, consider a loop of the form:

```
for {set i 0} {$i < 1000} {incr i} {
   ...
}
```

In TC, the object representing i's value would be converted to integer form once during its first use; thereafter, no conversions would be needed. In section 8.2.2, I measure the performance of such a loop and show that TC executes this loop an order of magnitude faster than Tcl. In the worst case, TC should be no slower than Tcl7.x, which performs all conversions.

3.1 Data Layout of Objects

To implement TC's conversion caching, we need a dual-ported structure containing both a string representation and a typed representation. In theory, only the typed representation is needed. In practice, read-only string use in Tcl appears to be sufficiently common (eg. for C callouts) that this cache will be ineffective without the ability to obtain the string form of a value without losing the preparsed representation. For the string representation, we reuse a structure Tcl7.x uses. This structure is tuned for copying and appending in the presence of a slow memory allocator like malloc() and consists of the triple {*allocLen*, *usedLen*, *strVal*}. *allocLen* refers to the

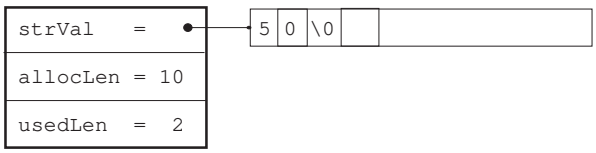

Figure 4: An example of a Tcl value, as stored in memory.

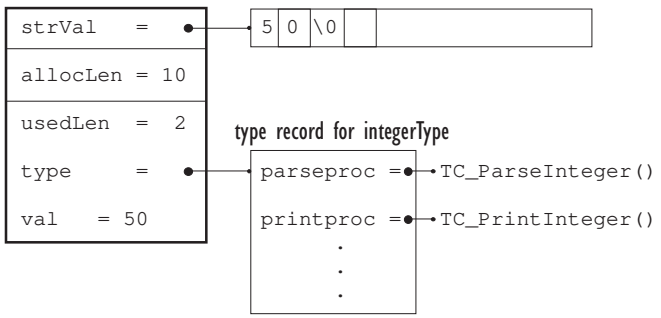

Figure 5: An example of a TC value of type integer. Both the binary representation as well as the string representation are valid.

amount of memory allocated to hold the string; this allows the implementation to over-allocate buffers, which amortizes the costs associated with finding such buffers from the free pool (eg. calls to malloc()). *usedLen* stores the amount of this arena currently occupied by valid data. It is used in copying, appending and other operations for performance. *strVal* contains the address of this arena. Figure 4 shows this data layout.

To store the type-tagged native representation, we extend this data structure to include a pointer to a type record and storage for this representation. The type record contains function pointers to the type-specific methods used in conversion. For each type, two methods are needed. The first is called the *printproc*, and it converts from the native representation to string form, setting the string representation for the object. The second is called the *parseproc*, and it converts from the current form (either a string or another native type) to the desired native type. The generality of parseprocs allows us to avoid the overhead of string conversion when we need to convert between two native types, as in the third statement in figure 3, where we need to convert directly from int to float. Naively, the parseproc could be coded to convert the object to string form using the old type's printproc, and then parse the string, as Tcl7.x does. Instead, the general form of parseprocs allows common *direct conversions* to be treated specially.

Storage for the native representation is placed directly in the object to save a layer of indirection and to reduce the number of memory allocations. In this scheme, a few words of storage are padded onto the object definition so that simple types can be stored entirely there, without having to maintain a separate area offset by a dereference. More complex types will use these fields as pointers to bigger, heap-allocated data structures. Figure 5 shows the layout of a sample object of type integer. The "val" field stores the binary representation and its C-language type is dependent entirely on the value in the type field. This design allows all objects to be the same size, allowing TC to replace malloc() with a fixed size object allocator, which I measured to be about 5x

```
int Tcl_IncrCmd(dummy, interp, argc, argv)
    ClientData dummy;    /* Not used. */
    Tcl_Interp *interp; /* Current interpreter. */
    int argc;           /* Number of arguments. */
    char **argv;        /* Argument strings. */
{
    int value, result;
    char *oldStr, *result;
    char newStr[30];

    /* ---------- error check the number of arguments ---------- */
    if ((argc != 2) && (argc != 3)) {
      Tcl_AppendResult(interp,
                  "wrong # args: should be \"",
                  argv[0], " varName ?increment?\"",
                  (char*)NULL);
      return TCL_ERROR;
    }

    /* ---------- the first argument is a variable ---------- */
    oldStr = Tcl_GetVar(interp, argv[1],
                        TCL_LEAVE_ERR_MSG);
    if (oldStr == NULL) return TCL_ERROR;

    /*---------- treat variable's value as an integer ---------- */
    result = Tcl_GetInt(interp, oldStr, &value);
    if (result != TCL_OK) {
      Tcl_AddErrorInfo(interp, "\n    (reading value
                       of variable to increment)");
      return TCL_ERROR;
    }

    /* ---------- get the step value ---------- */
    if (argc == 2) {
      value += 1;
    } else {
      int incr;

      result = Tcl_GetInt(interp, oldStr, &incr);
      if (result != TCL_OK) {
        Tcl_AddErrorInfo(interp, "\n    (reading
                         increment)");
        return TCL_ERROR;
      }
      value += incr;
    }

    /*---------- update the value for the variable ----------*/
    sprintf(newStr, "%d", value);
    result = Tcl_SetVar(interp, argv[1], newStr,
                        TCL_LEAVE_ERR_MSG);
    if (result == NULL) {
      return TCL_ERROR;
    }

    /*---------- set the result (as an object) and return ----------*/
    interp->result = result;
    return TCL_OK;
}
```

```
int TC_IncrCmd(dummy, interp, objc, objv)
    ClientData dummy;
    Tcl_Interp *interp;
    int objc;
    TC_Obj* objv[];
{
    TC_IntExp* stepVal=NULL, *intVarValue=NULL;
    TC_VarExp* varObj;
    TC_Obj* varValue;
    Var* rec;
    int step, result;

    /* TC provides some handy C macros to match arguments. These
       could easily be incorporated into Tcl. */
    ONLY_MATCH_NUM_ARGS(interp, objc, objv,
                        "varName ?increment?", 2, 3);

    result = TC_CoerceToType(interp, varType, objv[1]);
    if (result != TCL_OK) return TCL_ERROR;
    varObj = (TC_VarExp*)objv[1];

    rec=TC_EvalToVarRecord(interp, varObj, NO_CREATE);
    if (rec == NULL) return TCL_ERROR;
    varValue = rec->value.tc_value;
    result=TC_CoerceToType(interp, intType, varValue);
    if (result != TCL_OK) return TCL_ERROR;

    if (objc == 2) {
      step = 1;
    } else {
      result = TC_CoerceToType(interp, intType,
                               objv[2]);
      if (result != TCL_OK) return NULL;
      stepVal = (TC_IntExp*)objv[2];
      step = stepVal->val;
    }

    /*  touch() the object and set the variable to get the new object. This is needed by
       the copy-on-write mechanism, explained in section 5.3. */
    intVarValue = (TC_IntExp*)rec->value.tc_value =
                    TC_TouchObject(interp, varValue);

    /* In TC, we must explicitly invalidate the string form if it exists. */
    TC_ClearObjectStrVal(intVarValue);
    intVarValue->val += step;

    TC_SetObjectResult(interp, intVarValue);
    return TCL_OK;
}
```

Figure 6. The actual source code for the `incr` command procedure. On the left is the Tcl version; on the right is the TC version; comments are shared. I have attempted to align similar actions in each. In the Tcl version, the arguments are passed as strings, requiring their parsing during each call. In TC, arguments are passed as typed objects, which may have already been parsed by a previous command. If so, the call to TC_CoerceToType() will return without parsing.

Note: the code for TC is slightly more complicated because the Tcl version cheats, parsing the variable name twice: once in `Tcl_GetVar()` and once in `Tcl_SetVar()`.

faster than system malloc().

3.2 Type Conversion

In both Tcl7.x and TC, type conversions take place in the C command procedures. Figure 6 shows the `incr` command procedure as an example. `incr` takes one or two arguments: the first is a variable name and the second is an increment amount; if the second argument is not provided, incr uses an increment of one. `incr` then changes the integer value of the variable by the amount, updates the variable, and returns this value to the interpreter. As you can see, the TC and Tcl7.x versions look much alike. In the place of `Tcl_GetVar()` and `Tcl_GetInt()`, TC provides `TC_CoerceToType()`, which performs the type conversions as needed.

The library routines for conversion, TC_CoerceToType() and TC_CoerceToString(), are shown in figure 7, along with the type-specific methods for floating point numbers. TC_CoerceToType() returns a status flag. Since some values have no reasonable representation in the desired type (ie. “abc” as an integer), any attempt to

```
int TC_CoerceToType(TC_Obj* obj,
                    TC_TypeRecord* destType)
{
    int retn;

    /* already has a valid binary repr. of the desired type? */
    if (obj->type == destType) return TCL_OK;

    /* parseproc is not just for string->destType */
    retn = destType->parseproc(obj, destType);
    if (retn != TCL_OK) return retn;

    /* set the type, since we know what it must be. */
    obj->type = destType;
    return retn;
}

void TC_CoerceToString(TC_Obj* obj)
{
    /* already has a valid string representation */
    if (obj->strVal != NULL) return;

    /* printproc is the type-specific equivalent of sprintf() */
    obj->type->printproc(obj);

    obj->type = stringType;
    return;
}

/* an example printproc */
void TC_PrintFloat(TC_Obj* obj)
{
    TC_FloatObj floatObj = (TC_FloatObj) obj;
    char buf[100];

    sprintf(buf, "%lg", floatObj.val);
    TC_SetStringVal(obj, buf, strlen(buf));
}
```

```
/* An example parseproc.
 * Note: setting the type field is done by TC_CoerceToType(). */
int TC_CoerceToFloat(TC_Obj* obj)
{
    TC_TypeRecord* type1 = obj->type;
    TC_FloatObj floatObj = (TC_FloatObj) obj;
    char *s;

    /* special case of int->float */
    if (type1 == intType) {
        TC_IntObj intObj = (TC_IntObj) obj;

        /* retrieve the the binary representation */
        double val = (double) intObj.intVal;

        /* store back into the same space */
        floatObj.floatVal = val;
        return TCL_OK;
    }

    /* when a special case isn't available, we
     * convert type1->string then string->float */
    TC_CoerceToString(obj);

    /* perform the conversion */
    floatObj.floatVal = strtod(obj->strVal, &s);

    /* This is the equivalent of a dynamic type check in Tcl. */
    if (s == obj->strVal) {
        return TCL_ERROR;
    }

    return TCL_OK;
}
```

Figure 7: the algorithms used in conversion. C command procedures call TC_CoerceToType() and TC_CoerceToString() in the place of such routines as Tcl_GetInt() and Tcl_GetVar(). Note that coercing to strings cannot fail, since all values have a string representation in Tcl. This is why TC_CoerceToString() is made into a separate routine returning void.

convert this value should result in an error. This is the Tcl equivalent of a dynamic type check failure. In contrast, string coercion cannot fail ("everything is a string") and so the interface to string converters does not contain a provision for returning an error. One might compare this with Scheme [R4RS], where not all s-expressions have string representations (eg. continuations), or C, where no printf() format exists for code. In Tcl, even the bodies of functions have string representations and can be printed using the `puts` command.

This design has some ramifications for the rest of the system. First, when a command procedure sets one representation of an object, it must explicitly invalidate and deallocate the other representation. For example, just prior to the value increment in the `incr` command procedure, a call is made to invalidate the string representation, as shown in figure 6.

The second ramification is far greater: we need a new callback mechanism to replace the array of strings that Tcl7.x passes to C command procedures. Conceptually, this is because C functions need to operate over the native data types. In the case of `incr`, you wouldn't want to try to add the string values together- instead, you parse each string as an integer, add them together, and convert back to a string. Since the entire point of TC is to avoid these conversions, it is necessary that TC not pass strings to the C command procedures. As seen in figure 6, TC passes command procedures an array of dual-ported objects rather than an array of strings.

3.3 Backward Compatibility with Tcl7.x

As a matter of practicality, this design was made backward compatible with Tcl7.x's string style callouts. If TC insisted on calling all command procedures with object arrays, I would have had to rewrite all of the Tcl7.x builtin command procedures before the system would begin to work, including ones that have no impact on performance or are difficult to optimize. For example, it makes no sense to optimize `puts` because its execution time is dominated by the necessary I/O of printing to the screen or output device.

The mechanism for ensuring backward compatibility turns out to be remarkably easy to implement. When calling a Tcl7.x-style command procedure, TC coerces each of the arguments to strings, builds a string array and makes the call. The string results are then captured and used as the string representation in TC's result field, which is an object as well. To differentiate between the two styles, C command procedures are either registered as TC-style callouts or Tcl7.x-style callouts; the former are passed an array of objects, the latter an array of strings.

To make this scheme work, the Tcl7.x library access routines, written in C, had to be modified to request up-to-date information from TC. This is because TC cannot use the old Tcl access methods, which require string parsing of the inputs. Using the example of `Tcl_GetVar()`, which retrieves the value of a variable, Tcl7.x takes as input the string name of the variable. TC uses an offset into the current frame, which is faster than the hash table system Tcl7.x currently employs.

These less efficient access routines don't degrade the performance of the system, since most such calls will employ the TC-style mechanism. Thus, the TC access methods were written to maximize their own efficiency at the expense of Tcl7.x-style command procedures. For example, Tcl_GetVar() was rewritten internally to find variables in TC stack frames (see section 6.1 for details), even though the interface must still accept strings. This rewritten version may be slower than the original, since it is based on linear search rather than hashing. Again, this is justified by the rewriting of selected command procedures that work to ensure that such fallback is not a performance bottleneck. Indeed, if all of the command procedures are rewritten using TC-style callbacks, this mechanism is never used.

4 Compiler Preparsing

One of the major sources of delay in the execution of Tcl7.x is the time spent parsing scripts, as seen from the profile in section 2.2. In what now seems like a mistake, I wrote a static compiler to preparse scripts in an effort to eliminate this cost. This section describes the issues in the parsing of Tcl scripts and explains why it is preferable to simply cache parsing as a conversion from string form to an internal form.

4.1 The Parsing of Code in Tcl

In the original implementation of Tcl, code is parsed as it is executed, an expensive operation I wanted to minimize through the use of preparsing. Since Tcl7.x does not cache parsed expressions between computations, if a statement is executed repeatedly as in a loop, it is parsed repeatedly. It is easy to misinterpret this to mean that runtime parsing is the cause of this expense; in fact the key insight is that code is reparsed on each execution. If we can cache this parsing, then we will preserve Tcl semantics while providing a speedup in the common case where code is statically defined. For example, in the loop example from the last section, all of the arguments to the `for` command are statically defined and the cost of parsing should be subsumed by the time spent executing the body of the loop. If this statement is executed only once and the loop has only a few iterations, then the *Tcl*

portion of the execution time will be small, and so runtime parsing isn't a major performance hit. In other words, Tcl overhead is seen in the repeated execution of statements only. Since parsing is cached, a repeated statement is only parsed once, so the amortized cost is small.

One reason all language implementations don't parse statements at runtime is that it precludes global optimization (the optimization across multiple control flow constructs and statements, but within a procedure) and forces the runtime system to check whether a statement has been compiled or not, which can be quite expensive for simple statements. These two effects are related: the latter forces the separation of each basic block so that this check can be made. The entire point of global optimization is to treat multiple basic blocks as a single entity for the purposes of code generation. Numerous studies have shown that such optimizations vastly improve the quality of code generation, especially on RISC architectures, since it leads to intelligent register allocation, instruction scheduling, constant propagation, copy propagation, and other important optimizations[ARZ93]. The solution of cached, runtime parsing of longer code entities has been tried in the Self runtime [Cha91] with reasonable success. However, it is very complex and may not pay off for a language as dynamic as Tcl.

4.2 Dynamic Constructs in Tcl

Even though many constructs in Tcl are static, not all of them are, and thus we need to be able to parse code at runtime, for example `[eval $a]`. By our previous arguments, a compiler therefore serves no use: it doesn't replace runtime parsing, and won't help performance unless it performs global optimization. Since all Tcl primitives are defined in C (including user-defined ones) and since commands may be redefined at runtime, the results of global optimization will be valid only in the case when dynamically-defined commands are not used, and where user-defined C commands are not used. Essentially, this requirement excludes nearly all non-trivial scripts, including any use of the Tk user interface library and all scripts using object-oriented extensions such as Caste. This is because they dynamically define commands as "objects" like Tk does for widgets.

5 Optimizing the Object System

We now turn our attention to the extensions of the basic object system necessary to attain the performance improvements across the board. Essentially, there are three major issues and optimizations. First, no discussion has been made about memory management except a hand-waving argument regarding deallocation of a representation during the writing of the other representation. Here we present a more rigorous algorithm. From this

follows the need for explicit support for automatic memory management, where reference counting is chosen. This leads to a discussion about copy-on-write and its influence on the performance of TC.

5.1 Memory Management

Memory management in TC is more complex than in Tcl, where strings are the only form of storage. For example, the results of computations in TC need to be dual-ported objects in order to retain the native representation across computations. Since the native representation of an object is dependent on its type and may contain references to other structures, we need a type-specific way of deallocating objects, as when results of a computation go unused by the surrounding computation.

This leads to the need for automatic memory management. One solution would be to copy the return value and deallocate the copy when it is no longer needed. This would be quite slow and would necessitate a large number of allocations and deallocations. To avoid this, we need to allow multiple references to the same object and only deallocate an object when no references exist to it. Such automatic memory management is usually called garbage collection and there are a host of known techniques for implementing it[Wil92].

5.2 Reference Counting

For TC, I chose to implement reference counting, even though simpler and faster schemes are publicly available for C, including Boehm-Weiser's[BW88]. There are two reasons. The first is that reference counting behaves as an incremental collector, which avoids the pauses that non-incremental collectors typically incur. It is possible to build an incremental, non-reference-counting collector, but they tend to be quite difficult to implement and debug. The second advantage is that refcounts (the number of references to an object in a reference counting collector) can be used to implement copy-on-write, which is quite useful in Tcl (see section 5.3).

The downside is that refcounts cannot collect circular structures. However, Tcl is based on pass-by-value and offers no way for a callee to mutate arguments it is passed, whether such procedures are written in C or Tcl. This ensures that circularities cannot arise. The proof of this is fairly trivial: to create such a circularity, one would have to modify a structure to point to itself. But since the value being changed in such a computation must be distinct from the value being assigned, in accordance with pass-by-value, no circularity is made. Intuitively, a circular reference cannot exist in Tcl because all values in Tcl can be represented by finite string values. If a circularity were to exist, the interpreter would loop indefinitely attempting to represent this value as a string.

To implement refcounts, TC adds a refcount field to all dual-ported objects, including both code and data. This field tracks the number of data structures, including other objects, that possess references to this object. When the refcount reaches zero, the object is deleted, since it can never be accessed again. Object deletion requires the deallocation of the binary representation. This representation is type-specific and may be arbitrarily complicated, including pointers to other data structures or objects. Thus, TC needs to augment type records with another procedure used to delete objects' binary representations. This function pointer field is called the *deleteproc*.

Rather than try to explain all the picayune details of when refcounts are incremented and decremented, I will simultaneously offer an example and refer the reader to the volume of literature on garbage collection and reference counting [Pey87][Wil92]. Let's walk through the following example:

```
set a [expr {$a + $b}]
```

When this statement is executed, the values {set, a, [expr {$a + $b}], expr and {$a + $b}} have refcounts of 1, signifying their existence in the current statement. References to "set" and "a" are both copied into the array of objects (TC's equivalent of the *argv* array). This copying causes the objects' refcounts to increment. For the third argument, we evaluate the nested statement [expr {$a + $b}]. Like the outer statement, the references to the arguments are copied into an object array and the `expr` command procedure is invoked. Within this routine, a call is made to perform additional substitutions; from this, calls are made to retrieve the values of a and b. Since `expr` is synthesizing a new value from the summand of $a and $b, neither values' refcount will be changed. `expr` then calls a special evaluator for type "mathexpr", which performs the mathematical computation of the expression. The sum is a new object with refcount one, and it is returned. Next, the refcounts of the arguments to the nested command are decremented. The `set` command procedure is then invoked. It retrieves the variable record and decrements the refcount to the existing value, if one exists. It then replaces the value with the one passed as argv[2], incrementing its reference count.

The command is now completed, so the refcount of each of the arguments is decremented. The summand refcount is incremented as the result from the set, but is immediately decremented again, since this value is unused. This return value will not be deallocated, since it is stored as the variable's value, and so must have a positive refcount. Finally, if this is a top-level command, the code objects will have their refcounts decremented. In general, this triggers their deallocation, since top-level statements are only ever executed once. As would be expected, only the variable value lives on beyond this command, with a refcount of one.

5.3 Copy-On-Write

Copy-on-write is desirable in implementing pass-by-value because it minimizes copying costs in the common case when arguments to commands are not mutated. By comparison, command invocation in Tcl currently requires the interpreter to copy each of the arguments' values prior to the call. For large values this is quite expensive (see section 8.2.1 for details). With copy-on-write, the system simply copies the address of the object. If the value is never modified then no copying is needed.

As mentioned previously, refcounts offer the opportunity to implement copy-on-write. This is because copy-on-write requires the system to track which objects are shared. The rule of thumb is: if the object is shared, copy it before writing. If the object is unshared (eg. the writer is the only reference holder for that object), no copying is necessary. To implement these semantics in TC, we use the refcounts to track this shared state. A refcount of one indicates non-shared; greater than one indicates a shared object. Thus, callbacks that wish to change values ("side effects" in programming language parlance) must explicitly make sure they are the only holder of references to the value. This is accomplished by a call to the routine `TC_Obj* TC_TouchObject(Interp*, TC_Obj*)`, as seen in figure 6. If the refcount is one, the object is returned unchanged. If there are multiple references to the object, the refcount is decremented, and a copy is made. The copy, with refcount one, is then returned to the caller.

Since copying may involve copying the binary representation of a given object, and since there is no standard for layout of such data, we are forced to add another type-specific method for copying objects. This *copyproc* returns a duplicate of whatever object is passed.

6 Procedure Calls and Variable Access

The importance of optimizing Tcl procedure calls and variable access should not be understated. In practice, most of the code in a Tcl script resides within Tcl procedures, as opposed to the top-level script command list. While I offer no statistics to prove this argument, the only other mechanism a programmer could use to circumvent use of the proc facility is the C callout mechanism, but this negates the benefits of incremental development offered by the Tcl interpreter.

6.1 Procedure Calls Implementation

Tcl offers a command called `proc` that binds a new command to sets of Tcl statements. The arguments to such commands then become the parameters to such a procedure, and like calls to command procedures, use pass-by-value semantics. For Tcl, this means that arguments are copied into a procedure's call frame, creating local variables for each formal parameter. For TC, formal parameters also become local variables, but we can copy references instead of values, using the aforementioned copy-on-write scheme. In both Tcl and TC, the frame is destroyed on procedure return.

Tcl offers access to variables in the caller's frame through the `upvar`, `global` and `uplevel` commands. `upvar` copies a variable (by name) from a caller's scope into the local one; the `uplevel` command executes a statement in a caller's scope. The `global` command is synonymous with an `upvar` to the top-level scope, so by default, no globals are visible in a Tcl procedure. Again, these semantics affect TC and Tcl identically because both chain procedure invocations together and both provide a direct link to the global scope. Access to prior scopes is effected by copying the relevant variable records from prior scopes into the current one. The variable record is a structure containing bookkeeping information about a variable and a reference to its current value.

The major difference between Tcl and TC is how call frames are represented. In Tcl, frames are hash tables keyed on the string names of variables. In TC, we optimize frames to be static arrays. Before a procedure is first executed, it is scanned for variable references of the form `$varname`. These are collected and each unique variable is given an entry in this array. The references in the source are then replaced by indexes into this array.

There are three issues in this design. First, we have to handle upvar references. Second, we have to efficiently handle uplevel calls. Lastly, we need to handle variable references not captured in the scan of dollar-sign uses, such as in `[set a]`. For upvar references and global variables, TC uses the same approach as Tcl: it follows the chained call frames to the desired one, locates the variable by a search on its string name, and sets a forwarding pointer in the current frame to the variable record in the upscope frame entry. The only difference is that in TC, this is a linear search instead of a hash table lookup. We cannot copy a reference to the actual variable value because we want to share more state information than just the value. For example, a value copying scheme will fail to preserve this link if the variable doesn't already exist in the uplevel scope. In Tcl's semantics, such a link allows either scope to initialize the variable.

`uplevel` is not implemented in the current prototype. To implement it, the TC design would compile the body for the higher scope. Caching of this compilation across repeated statements is complicated: normally, when a body is compiled for the "current" frame, the frame itself is fixed across invocations, even if the contents of variables changes. With uplevel calls, though, the scoping level chosen can be different for different invocations of the uplevel statement. It is possible to concoct schemes to circumvent this. For example, the absolute scoping level for which the body is compiled can be stored with the compiled body. If uplevel is called and the body is compiled for the wrong scope, it can be recompiled.

Finally, TC must handle variable references not captured in the sweep of dollar sign usages. This arises in two cases. The first case is when a C command procedure reads a variable value using Tcl_GetVar() or Tcl_SetVar(). The second case is a statement of the form [set $a 5]. In this case, we capture the use of "a" but not the variable whose name is the runtime value of a. However, `set` is implemented in C and uses Tcl_SetVar() just like C command procedures written by end users. Thus, only the first case need be considered.

To handle this case, TC needs to modify the routines Tcl_SetVar() and Tcl_GetVar() to implement these *dynamic variables*. These variables are always referenced by their string names, so we can add an auxiliary hash table of such uncaptured variables if the reference isn't found in the main list of captured variables. Since this case employs variable name lookup using string names, performance will be poor in any implementation. It is important to ensure correctness, not efficiency, and so I make no attempt to optimize these uncommon cases.

Note that the current prototype implementation of TC does not fully implement all of the these features.

6.2 Array Variables in TC

Tcl provides a facility for associative arrays, which are bound to variables. Unlike C-language arrays, Tcl arrays are indexed on arbitrary strings. For example, $a($b) refers to the element of the "a" array whose index value is the same as the current string value of the "b" scalar variable.

While it is possible to preparse array references, it is not usually possible to preparse the index value. Indeed, not only does the index value typically require repeated evaluation, but it also must be converted to string form before lookup. Consider the case of an integer loop through an array:

```
for {set i 0} {$i < 1000} {incr i} {
  somecmd $a($i)
}
```

Figure 11: abstract syntax tree for `{$a<5}`, without compression (left) and with compression (right).

Although TC can parse the reference to the "a" array variable, the index must be hashed as a string on each reference. Both systems store array values in hash tables keyed on the index's string value. This limits TC to be a lackluster improvement for array variables, since execution should be dominated by the hashing and lookup.

7 Type-Specific Optimizations

7.1 Math Expressions

In a typical language interpreter, the runtime evaluates mathematical expressions by evaluating nodes in an evaluation tree, with a node's result value being the value of the subexpression it represents. In this tree, the data elements at the leaves are numerical values in registers or memory locations. In TC, we can assign a new type to such trees, *mathexpr*, and to ensure that math expressions will not need to be reparsed in repeated evaluations. We can then use a similar tree to represent TC math expressions.

The problem with this scheme is that the nodes in the expression tree are objects tagged with types, which are expensive to build and evaluate relative to the common case of dollar-sign variable references and constant numerical values. Thus, TC uses a special structure to represent a node, with an enumerated tag describing the mathexpr type contained: either an int, a float, a string, a variable reference, or an object. For numerical values, the C language value is placed in this same structure by padding a few words of memory onto its definition.

We can go one step further and special-case common subtrees. For example, a common subtree in a conditional expression is `{$varname <` *intval* `}`, which is commonly used in loops. A special handler for such cases will flatten this tree by storing the entire subtree in a single node. The tradeoff is against code size and complexity, which in TC amounts to two hundred lines of boilerplate code. In TC, I arbitrarily special case the following subtrees:

{ VAR_<op>_VAR, VAR_<op>_INT, INT_<op>_VAR, VAR_<op>_FLOAT, FLOAT_<op>_VAR,
<unaryop>_VAR, VAR_NEQ_ZERO, VAR_EQ_ZERO, VAR_NEQ_ONE, VAR_EQ_ONE }

<op> ∈ { LT, LTE, GT, GTE, EQ, NEQ, PLUS, MINUS, MULT, DIV, AND, OR, ...}
<unaryop> ∈ { UMINUS, BIT_NOT,...}

The four cases of VAR_{NEQ,EQ}_{ZERO,ONE} are for common boolean expressions, and the parser takes advantage of commutativity and swaps the arguments if they are reversed (eg. {0 != $a} becomes {$a != 0}).

7.2 Lists

Tcl lists are currently stored as strings, affording a great opportunity for improvement, but their semantics reflects this implementation, and so might frustrate an attempt to duplicate these semantics in TC. In Tcl, lists are stored as strings and parsed into elements separated by whitespace; “1 2 3” is a valid list with three elements. This provides a great opportunity for TC to improve performance for nearly all list operations, such as indexed lookup, by maintaining lists in parsed form. However, a list abstraction would not preserve semantics because Tcl maintains a list’s string form under update, where TC invalidates the string. For example, if we append an element “d” to “ a b c” we get “a b c d”, not “ a b c d”, which is what Tcl would do. A naive implementation that converts to string form by inserting singleton spaces between objects will fail to preserve these semantics. It is possible to augment the simple list structure to maintain whitespace, but it didn’t seem worthwhile for this prototype, especially since it is a rare case when these semantics are needed.

A TC list is stored as an array of object references. Each reference represents an element in the list, and like other string values, is parsed on demand in TC. If and when the string value for the list is requested, it is constructed by concatenating the string values of the elements together. There was an alternative implementation, where lists are stored as pointers into the original string representation of the list.. I chose not to implement lists this way because it adds a dependency between the string and compiled representations, so that the string representation could not be discarded without updating the compiled representation.

The choice of array storage versus linked list storage trades slower updates for faster lookups. To ease the cost reallocation under append (a common case for updates), TC overallocates the array, using a usedLength and allocLen field just like Tcl and TC do for string values. Performance for mid-list insert and delete still suffers under this scheme because of copying costs.

8 Performance

Performance of TC relative to the original implementation of Tcl is the project’s metric of success. In this way, TC is quite successful, showing improvements of 5x to 10x for many common operations. As seen in the sections to come, some cases did not scale as well as others, reflecting a lower bound in overhead.

8.1 Testing Methodology

The test suite used in this thesis stresses microbenchmarks. This exclusion of more aggregate tests is a result of instability and incompleteness in the current version of the software, not necessarily a foreboding of poor performance or a result of negligence. The microbenchmarks chosen reflect only those language features I attempted to optimize using object-style callouts; exception handling and traces are excluded, as are core command procedures I didn't reimplement to accept arrays of objects instead of arrays of strings.

The system chosen to implement TC was an DEC Alpha 4000/300 capable of about 100 MIPS. It was running OSF/1 v.1.3 at the time of the tests, and I compiled TC using the GNU C compiler. The tests themselves were run while the machine was unloaded and they were each repeated several times.

8.2 Micro Benchmarks

Microbenchmarks attempt to isolate a small piece of a system to demonstrate how efficiently this feature is implemented. In TC, the microbenchmarks I've run show a reasonable improvement for the features I've optimized. The following subsections details each class of optimization.

Description	TC	Tcl	Tcl/TC
calling C command procedures:			
0 args	4.24µs	6.14µs	1.4x
1 args	4.24µs	7.48µs	1.8x
4 args	6.58µs	10.5µs	1.6x
8 args	8.96µs	19.0µs	2.1x
set varname	5.4µs	15.2µs	2.8x
set varname 123	6.8µs	19.5µs	2.9x
set varname $a	7.9µs	29.2µs	3.7x
set a <4Kbyte string>	6.9µs	1.8ms	260x
concatenate five values*	27µs	55µs	2.0x

Table 1. Performance of primitive operations: a="123".
*See Appendix B for additional source code

Dollar-sign variable substitution and setting of large data items is made fast by the object system, but string operations like concatenation show little benefit over existing Tcl. Unless otherwise specified, strings are kept small to minimize the impact of Tcl's string copying costs.

8.2.1 Primitive Operations

This set of benchmarks is designed to test statement execution, data assignment, substitution, and copying. In the first set of these tests of table 1, I measure the time TC and Tcl take to invoke a C command procedure which does no work with its arguments. This number sets a lower bound on performance improvement for inexpensive commands like `set` and `incr`.

The second set of tests measures variable access and assignment. The first test in the set uses the `set` command to return the value of varname, whose value is "123". The second test, an assignment, differs from the first only in the actual assignment– both return the final value of the variable. The difference in execution time therefore shows the time for assignment. In TC, assignment takes (6.8 - 5.4) = 1.4 usec; in Tcl, this is (19.5 - 15.2) = 4.3 usec; TC assignment is roughly 3x the speed of Tcl. Performance is not impressive because TC incurs a large, fixed cost for object creation and Tcl's overhead for string copying is minimized for small sizes strings.

The time for substitution is likewise seen in the difference in execution time between the second and third tests of the second set, which vary only in the source of the new value. This involves local variable lookup, so TC's indexed access offers a significant improvement over Tcl: 1.1 usec versus 9.7 usec for a speed of nearly 9x.

The last set of tests in table 1 demonstrates the best and worst cases of TC's performance relative to Tcl's in terms of data copying. In the 4K string assignment, Tcl must copy the entire string value each time, where TC can copy a pointer and bump a reference counter. The less attractive case is shown by string concatenation, where TC must essentially perform the same work as Tcl in concatenating the string values together. I hypothesize that the performance gain comes from faster variable substitution and from faster statement execution. Subsequent tests (not shown) indicate that this 2x relative performance is fairly consistent across larger numbers of concatenations but decreases as the string copying overhead increases with longer individual strings.

Description	TC	Tcl	Tcl/TC
`for` loop: count from 1->10000	140ms	1450ms	10x
`while`: sum the 1st 1,000 int's	24ms	240ms	10x

Table 2. Loop performance. See Appendix B for source code.

8.2.2 Loops

Loop operations, which repeatedly execute the same script, offer a great opportunity for improvement. In most cases, the cached parsing of the body of the loop will not be invalidated, unless the code is dependent on the loop control variable. As seen from the speedups, this common case is drastically improved by code caching, reaffirming common knowledge that interpreted code is typically an order of magnitude slower than compiled or byte-compiled code.

Description	TC	Tcl	Tcl/TC
procedure call with small args:			
0 args	8.3µs	15.4µs	1.9x
1 arg	10.4µs	35.9µs	3.5x
4 args	16.4µs	66.3µs	4.0x
8 args	27.3µs	111µs	4.1x
procedure call with varargs:			
0 varargs	11.6µs	40.3µs	3.5x
1 vararg	14.2µs	46.4µs	3.3x
4 varargs	18.8µs	59.5µs	3.2x
8 varargs	22.0µs	76.5µs	3.5x
procedure call with default args:			
1 default arg, 0 args passed	11.4µs	40.2µs	3.5x
1 default arg, 1 arg passed	11.9µs	40.0µs	3.4x
4 default args, 0 passed	16.8µs	67.5µs	4.0x
4 default args, 4 passed	17.0µs	71.8µs	4.2x
8 default args, 0 passed	24.5µs	102µs	4.2x
8 default args, 8 passed	28.8µs	116µs	4.0x
procedure call with one 1K arg	11µs	305µs	28x

Table 3. Procedure calls. See Appendix B for source code. The arguments passed in all cases except the last are small, so Tcl's string copying cost is minimized. TC's relative performance should be better in real scripts with longer string data.

8.2.3 Procedure calls

Procedure calls are important to measure because they are the mechanism typically used to bundle small chunks of Tcl code together. As in other languages like C++, Tcl procedures are parameterized on input arguments and allow arguments to have default values and variable numbers of arguments ("varargs"). Default arguments are parameters that are optional– if the caller does not supply an input argument, the calling mechanism will substitute a default in its place. *varargs* is a

facility where the caller can pass more arguments than there are parameters in the callee's definition; in this, a special parameter, "args", is assigned a Tcl list composed of the unmatched arguments. Both facilities are used widely in Tcl scripts and so should be measured.

I have attempted to cover these cases through a series of tests shown in table 3. The first set shows the simple case where neither default args nor varargs are used. As seen, results are typically 3-4x faster for TC, primarily because string copies and variable name hashes are replaced by pointer copies and indexed variable access. The second set of tests exercises the varargs facility. Here TC's relative performance suffers because of the high cost of array-based list construction for short string parameters, as used in the tests. The third set shows performance of the default args facility, which maintains a fairly consistent improvement across different numbers of parameters and defaults. The last test demonstrates the value of copy-on-write for procedures. In Tcl, this one kilobyte string must be copied each time, where in TC, only a pointer is copied.

8.2.4 Array Variable Access

I ran a short battery of tests to determine the relative performance of array variable access in TC. The latter two cases, where the index name is substituted from variables, are a common idiom in Tcl code. Array variable access performs as predicted in section 6.2: a smaller improvement over Tcl than TC was able to achieve for scalars. Better performance can only be achieved by allowing array indexing on non-string types, since this would afford TC the ability to also cache the index value, a native type (eg. integer). Native type indexing would afford faster lookups in cases where the cache is invalidated because string operations such as hashing and lookup set a lower bound on performance and do not scale well for longer string values.

Description	TC	Tcl	Tcl/TC
`set varname(index)`	12.5µs	20.2µs	1.6x
`set varname(index) 123`	13.4µs	24.3µs	1.8x
`set varname($idxname)`	13.0µs	31.6µs	2.4x
`set varname($idxname) 123`	14.4µs	35.4µs	2.5x

Table 4. Performance of array variables: varname(index)="123"; idxname="index".

The speed of array variables is hindered by the evaluation of the index expression, which causes the access to not be cacheable.

8.2.5 Math Expression Evaluation

Math expressions are in the critical path of Tcl performance because of their use in determining control flow (ie. `if`, `while`, etc.), as well as their use in numerical computation. To measure the effects of TC's optimized math expression evaluator, I ran a set of four

Description	TC	Tcl	Tcl/TC
`{$intvar != [set zero]}`	51.5µs	98.6µs	1.9x
`{$intvar != $zero}`	17.7µs	81.9µs	4.6x
`{$intvar != 10}`	14.8µs	79.5µs	5.4x
`{$intvar != 0}`	12.6µs	78.1µs	6.2x

Table 5. Math expression performance: zero="0"
The four tests highlight math-specific optimizations using an expression tree compression algorithm.

tests, all essentially the same expressions, but implemented in different ways. The first test shows the results of using objects as nodes in the expressions tree because the current optimizer has no optimized node for this kind of expression. This suggests that a more aggressive optimization policy would be useful; in this case, it would make sense to support VAR_<op>_OBJECT as a new kind of optimized node.

The remaining tests are more representative of real TC performance for small math expressions. The second test involving comparison of the variables is therefore a more reasonable base case. In this test, the parser is able to use the VAR_<op>_VAR optimization, but requires two variable lookups and overhead for memory management of the results from each. In a slightly optimized implementation of TC, it would be possible to reduce this memory management overhead by accessing variable values without changing their refcounts for read-only cases such as math expression evaluation. The key requirement would be to never return a reference to a variable's value from the evaluation of a node, but instead to copy the {integer, float, or string} value out of the result object so that the reference count for the object never changes.

The last two tests show VAR_<op>_INT and VAR_NEQ_ZERO and again, the difference between them is artifically poor because TC is not as optimized as it could be. In this case, VAR_<op>_INT is performing poorly because of I decided to opt for simpler code at the expense of top performance.

While this suite of tests seem to validate the thesis that type specialization of math expressions will improve performance by reducing layers of indirection and costs for type checking, they seem anomalous: 1 microsecond on the Alpha-based workstation that I ran the tests on, is about 100 instructions, making these numbers seem suspicious; a pointer traversal and associated type checking should not cost 200+ instructions as it does between the third and fourth test. Nevertheless, these numbers are reproducible given the current versions of TC and Tcl, and to the best of my knowledge, no memory is leaking for these tests.

8.2.6 Lists

As one of the only aggregate data structures provided in the Tcl core, lists are very important to overall performance. Thus, a series of tests was run to measure TC's performance over list operations. It is clear that an array-of-elements based storage system will scale better than a string-based one, so the results for `llength`, `lindex`, and `foreach` come as no surprise. Their measurement was more designed to show TC's fixed overhead for list management and object creation. The remainder of this section explains the other cases: `lrange`,

`lsearch`, `lappend` and `lsort`.

The `lrange` command extracts a subrange of a list and returns the newly formed list. For lrange, TC's advantage is less obvious because we now need to create a new list and populate it with objects, which incurs a relatively high cost for bumping the refcounts on each copied reference. By comparison, once Tcl has parsed the list up to the starting element, it only needs to perform a string copy. This explains why TC's relative performance increases between the first two cases, when the length of the subrange is fixed at 6, but where the starting index increases. The cost of refcounting and list creation are the reasons behind the lackluster performance in the last case, where Tcl's parsing overhead is minimized and TC's object overhead is maximized.

Description	TC	Tcl	Tcl/TC
`llength $L2`	9.0µs	69µs	7.7x
`llength $L1`	9.15µs	540µs	59x
`lindex $L1 0`	8.5µs	60.7µs	7.1x
`lindex $L1 100`	8.7µs	276µs	32x
`lindex $L1 200`	8.5µs	498µs	59x
`lindex $L1 5000`	8.2µs	500µs	61x
`lindex $L3 2`	8.5µs	51.6µs	6.0x
`lindex [lindex [ lindex $L3 2] 2] 1]`	24.8µs	128µs	5.2x
`foreach item {a} { }`	11.4µs	41.6µs	3.6x
`foreach item $L2 { }`	15.8µs	127µs	8.0x
`foreach item $L1 { }`	157µs	2.1ms	13.4x
`lrange $L1 3 8`	17.2µs	92.0µs	5.3x
`lrange $L1 153 158`	14.5µs	459µs	32x
`lrange $L1 3 102`	163µs	330µs	2.0x
`lsearch $L2 joey`	16.0µs	114µs	7.1x
`lindex $L2 [lsearch $L2 joey]`	24.2µs	171µs	7.1x
`lindex $L1 [lsearch $L1 150]`	103µs	1.5ms	14.6x
build L2 using `lappend`*	131µs	536µs	4.1x
build L1 using `lappend`*	2.3ms	10.3ms	4.5x
`lsort $L2`	25.6µs	129µs	5.0x
`lsort $L1`	3.2ms	4.3ms	1.3x

Table 6. List operations. *See Appendix B for source code.

List performance varies widely in this implementation based on arrays of objects, but is generally a marked improvement over Tcl, which must reparse its string-based lists on each usage. L1 is the list of the first 200 positive integers (long length with short items). L2 is a list of 9 peoples' names, about 5 characters each.

The `lsearch` command searches for a string element in a list of items; in table 6 we see that this is much more efficient in TC than in Tcl. In both implementations, the string comparison algorithm aborts the comparison early if the two strings are not equivalent, so Tcl's parsing overhead dominates execution time in the common case where there are many failed comparisons before a match is found. This is why the third case, with 150 failed comparisons, is so much faster for TC than Tcl.

`lappend` appends new elements to the list represented by a given variable, and is commonly used to construct lists; in table 6 we see a fairly constant advantage in TC over Tcl. In measuring lappend, it therefore makes sense to construct two test lists, L1 and L2, by appending each element in order. Since TC overallocates the list array, TC amortizes the high cost of reallocating and copying the list for repeated append operations. In this way, TC's append operation scales, as seen by the examples shown. Since Tcl overallocates its string array, it too scales. Presumably, for arrays of longer string values, TC's advantage would be greater, reflecting the cost

for Tcl to copy the strings into the array string.

Lastly, I measured the `lsort` command, which copies the list and sorts the duplicate list, returning the duplicate. `lsort` is faster for TC mostly because of reduced parsing costs, which accounted for 1.4ms of the 4.3ms it took Tcl to sort the 200 element list. In the steady state of a "large" list, both are using the system's qsort() routine using strcmp() for the comparison function, so both should perform identically, since parsing is O(n) and sorting is O(n) for n small list elements.

Because of memory management bugs, I was not able to measure `linsert` or `lreplace` in this version of the system. In truth, their performance would probably have been lackluster. Mid-list insertions and deletions to any array-based implementation requires data copying just like Tcl requires, so TC's performance will improve only inasmuch as string copies are dependent on the length of the strings which correspond with the elements of the list. In TC, the copying cost would depend only on the length of the original list and the location and number of elements insrted or deleted.

Description	TC	Tcl	Tcl/TC
factorial (1), recursive	34µs	205µs	6.0x
factorial (4), recursive	265µs	1600µs	6.0x
factorial (7), recursive	510µs	2996µs	5.9x
factorial (1), iterative	102µs	385µs	3.8x
factorial (4), iterative	239µs	1052µs	4.4x
factorial (7), iterative	378µs	1740µs	4.6x
fibonacci(5), recursive	950µs	4.0ms	4.2x
fibonacci(10), recursive	11.4ms	48.3ms	4.2x
converting a list into an array	5.22ms	14.8ms	2.8x

Table 7. Performance in aggregate examples. See Appendix B for source code.

These examples combine several of the above features in an attempt to simulate multiple feature usage. Due to a few unimplemented features, and the lack of support for Tk, real macro benchmarks were not attainable.

My conclusions are that most of the benefits of smart list implementation come from preparsing and replacing string copying with object pointer copying. Improvements in TC's list algorithms should be assessed on an application-specific case, accounting for scaling and common operations; it may make more sense to choose a different data structure than arrays, and bind the operations to new command names rather than to change TC's list implementation.

8.3 Macro and aggregate benchmarks

In any performance test suite, it is important to include results from real-world examples. Unfortunately, in the current state of development, TC is not sufficiently complete to run such benchmarks. Thus, it seems unrea-

sonable to try to generalize from the language subset presented herein. Instead, I show a few examples where several language features are used together:

Description	TC	Tcl	Tcl/TC
factorial(4), well-written	265µs	1600µs	6.0x
factorial(4), compressed	265µs	1405µs	5.3x
speedup from compression	0%	12%	
list->array, well-written	5.22ms	14.8ms	2.8x
list->array, compressed	5.29ms	10.4ms	2.0x
speedup from compression	-1%	30%	
`set longerName`	6.1µs	17µs	2.8x
`set a`	6.1µs	12µs	2.0x
speedup from compression	0%	29%	
`set longerName 123`	7.3µs	24µs	3.3x
`set a 123`	7.3µs	19µs	2.6x
speedup from compression	0%	21%	

Table 8. Effects of whitespace and long identifiers in source. Source code appears in Appendix B.

TC removes whitespace as part of parsing, but Tcl remains sensitive to this effect, since it reparses statements each time they are executed, including nested statements, like the bodies of `for` loops. "well-written" tests include comments and long identifiers; "compressed" tests use minimal-length length names.

"factorial" and "fibonacci" are implementations of small mathematical functions. Mathematics is one class of CPU-intensive work that appears in a program's "inner loop" and which executes slowly in Tcl. The test contains a mix of control flow code, simple mathematics, and procedure calls.

"converting a list into an array" involves converting a long list into the index-value pairs of an array variable, where odd elements become indices and even elements become values. The test contains a mix of control flow, assignment, and array access.

8.4 Effects of whitespace and long identifiers

Given that Tcl is sensitive to the lengths of identifiers and whitespace, one could write a program that substituted minimum-length identifiers and that strippped whitespace. It is therefore important to show that TC's performance gains mostly derive from this effect, which TC achieves by cached parsing. This is shown at right, where the speedups are typically 10-30%. Such a speedup leaves ample margin for TC to further improve performance, thus proving the claim that such an optimizer would not obviate the need for TC.

8.5 Overall Performance Conclusions

In taking stock of the overall performance of TC, there are three issues. The first is whether the delayed conversion scheme was effective. The second is the raw overhead for execution of statements in Tcl and TC, which is quite high. Lastly, there were a few obvious measurements I left out this report, whose exclusion I explain at the end of this section.

From the performance numbers above, I claim that the original thesis is correct: string semantics aren't inherently slow; string storage and manipulation is slow. This, of course, is only true for the data types whose

methods do not require strings, such as integers. For features such as Tcl-style array variables, it is the string-based semantics that limit performance. Faster execution would require a redesign of this feature. For features that depend on strings, such as concatenation, the only hope is to avoid them in the "inner loop" of programs that need to execute quickly. For example, rewriting the concatenation algorithm in C won't make it become algorithmically cheaper.

One surprising result was the large overhead incurred by statement execution in TC. For example, for simple commands like set, nearly all of the execution time was spent executing the command and little actually assigning the value to the variable. I have looked at the main loop of TC and attempted to optimize it, but there doesn't seem to be much improvement that can be made in the current framework. One design change that might lead to greater performance would be to substitute real garbage collection for reference counting. Reference counting is reputed to be slow because it requires the system to touch objects on all reference count changes, not just during garbage collection, requiring many memory references[Pey87]. I have not experimented with this because it would mean that copy-on-write could not be supported. Copy-on-write provided a dramatic improvement in the cost of calling Tcl procedures and in the cost of variable assignment, so I am reluctant to sacrifice this optimization. One compromise would be to copy on all writes, a policy which does not require reference counting. The tradeoff is that in cases where an object is not being shared, copies are still being made on writes; if this is a common situation, then the copying overhead will be high. I did not measure this policy and it would make an interesting experiment.

There were a few measurements I did not include here because they turned out to be insignificant in the environment I used to design this system. The first was memory usage in TC, which wasn't appreciably greater than Tcl. This is because data objects are shared instead of copied, and because TC reuses objects when they become unreferenced as soon as possible (including code). In a larger script, storage space might become a problem if many procedures are defined or if many small objects are stored. I did not measure these cases, primarily because Tcl scripts tend to be relatively small and are currently limited by execution time, not by space. At the time of this writing, a typical personal workstation has 16-64MB of RAM, so if TC wastes an extra megabyte to store data, this isn't significant. For embedded systems, portable computers and other space-limited systems, this may not be the case, and further measurement would be necessary.

The second timing number I left out was the compilation time for the parser. As mentioned previously, I made a mistake in writing a separate parser for Tcl scripts, whose output was a preparsed script that the runtime could efficiently execute. Since this preparser would be removed in a production version, it seemed futile to

include its performance. Furthermore, in every case I ran (up to a few hundred lines), compile times were essentially instantaneous, limited by file I/O and not by the time to parse scripts.

9 Conclusions

Beyond improving the raw performance of Tcl scripts, TC changes the relative costs of various Tcl constructs, and I claim that this is strictly for the better. Specifically, TC reduces the overhead for parsing, so you don't pay additional costs for using long identifiers or comments. TC also implements copy-on-write, so using larger data objects is not inherently slower under Tcl's call-by-value semantics. This encourages the use of Tcl procedures, which normally require the copying of their arguments before the call is made. By reducing their relative costs, TC encourages users to write cleaner code with more comments, the use of procedural abstraction and the use of Tcl's lists. By reducing the performance benefit, TC discourages the rewriting of Tcl routines in C.

A primary lesson to be taken from this work is that use of strings for storage leads to poor performance. While this is obvious when compared to compiled language implementations, the real costs are not so obvious for an interpreted language in which strings are as important as they are in Tcl. This work quantifies this cost, and demonstrates one way to improve upon string storage without sacrificing string semantics.

On an abstract level, this project demonstrates that changing the model under which you work rarely improves things itself. The benefits instead come from the full exploitation of the new model. For example, the bundling of the dual-ported representation into a single entity for the purposes of memory allocation was at first a trivial change. This then led to reference counts which led to copy-on-write, which in turn made procedure calls and variable assignments less expensive by avoiding extra data copies. I claim that this result was not obvious from the initial, seemingly minor design decision.

References

[ARZ93] Fran Allen, Barry Rosen, and Kenneth Zadeck. *Optimization in Compilers*. forthcoming.

[SICP85] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *The Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, 1985.

[Bou78] Steven Bourne. "The Unix Shell", *Bell Systems Technical Journal*, AT&T., Murray Hill, NJ. (July-August 1978)

[Bra93] Michael Braverman. "Caste: A Class System for Tcl", *Proc. Tcl'93*, Berkeley, CA. (June, 1993)

[BW88] Hans-Juergen Boehm and Mark Weiser. "Garbage collection in an uncooperative environment". Software Practice and Experience, September 1988.

[Cha91] Craig Chambers. *The Design and Implementation of the Self Compiler*. Ph.D. Thesis, Stanford Univ., 1991.

[K+R88] Brian Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ 1988.

[Lee91] Peter Lee. *Advanced Programming Language Implementation*, MIT Press, 1991.

[Ous93] John Ousterhout. *An Introduction to Tcl and Tk*. Addison-Wesley, Reading, MA, 1994.

[Pey87] Simon L. Peyton-Jones. *The Implementation of Functional Languages*. Prentice-Hall, Englewood Cliffs, NJ, 1987

[RM92] John R. Rose and Hans Muller. "Integrating the Scheme and C Languages". Trans. of Lisp and Func. Lang, New York. (June 1992)

[Sah94] Adam Sah. (unpublished survey taken of users frmo comp.lang.tcl during April and May, 1994). forthcoming.

[SB93] Adam Sah and Jon Blow. "TC: A Compiler for Tcl", *Proc. Tcl'93*, Berkeley, CA. (June 1993)

[SBD94] Adam Sah, Jon Blow and Brian Dennis. "An Introduction to the Rush Language", to be published in *Proceeding of the Tcl'94 Workshop*. New Orleans, LA (June 1994)

[Sta91] Richard Stallman. *The GNU Emacs Lisp Programmer's Manual*. Free Software Foundation, Cambridge, MA, 1987

[R4RS] William Clinger and Jonathan Rees, Ed. "The Revised R^4 Report on the Algorithmic Language Scheme", Lisp Pointers IV (July-Sept 1991)

[Wil92] Paul Wilson. "Uniprocessor Garbage Collection Techniques". 1992 Int'l Workshop on Memory Management, Springer-Verlag, 1992

[YS94] Curtis Yarvin and Adam Sah. "A Portable Library for Runtime Code Generation in C", UC Berkeley Tech Report #UCB-CS-94-792

Appendix A Future Work

The obvious question to ask is whether there aren't additional improvements to be found. Although I've tried to answer this in the negative using semantic arguments throughout the paper, another approach is possible, which is to compare the performance of Tcl and TC with languages which are less hostile to efficient implementation. For example, one can peg Tcl as being approximately 3 orders of magnitude slower than C for mathematics and control flow operations. TC then would be 2-2½ orders slower than C. By comparison, interpreted Scheme under SCM, a compact and unheroic implementation, is about 2-5x faster than TC from some small tests I ran that were similar to the benchmarks shown herein. If you compile dynamic languages like Scheme and Smalltalk into native machine code, you would typically find them to be about an order of magnitude slower than C[Cha91]. In other words, for TC to much improve beyond its current implementation would seem unlikely, since these other languages are far less difficult to analyze and optimize than Tcl and only provide a 10x improvement through their efforts.

It is tempting to try to redesign some features of Tcl to answer this performance issue because of the language's enormous popularity. It is worth a study to find the source of this popularity: people don't learn new languages as a way to avoid vacationing in Aruba. If it is the case that the most popular features of Tcl do not hinder performance, then such a redesign might be feasible. Alternatively, we might ask whether the expensive features of Tcl are really all that useful.

For example, a survey of users from comp.lang.tcl [Sah94] revealed that few users had use for "uplevel" or "upvar", except as ways to simulate pass-by-name and other parameter passing schemes. Likewise, the main use of dynamic (re)binding of procedures from C stems mainly from, ironically, performance: at least one object-oriented Tcl package relies on rebinding to create new "objects". However, in its original Tcl implementation, performance was a severe problem and so Caste[Bra93] was later rewritten in C.

I am currently investigating whether a happy medium cannot be found. Features such as implicit string conversions, a C callout mechanism, scripting syntax, and embeddability all seem quite useful and none seem to limit the theoretical performance of the system. The real question is whether the utility of Tcl will be hindered with its harmful features excised. This effort has culminated in a Tcl-like language called Rush that preserves much of Tcl's syntax and semantics, and yet offers performance hundreds of times that of Tcl. Rush will be presented at the Tcl'94 Workshop [SBD94].

Appendix B Test Suite Source Code

```
# n-time is a C command procedure similar to time, but outputs to 3
#   significant figures in floating point.

proc print-results {pref result} {
    set pref "$pref - "
    puts stdout [format "%-25s %s" $pref $result]
    puts stderr "." nonewline
}

proc n-times {{n 1}} {
    set base_times 1000
    return [expr {$n*$base_times}]
}

# 8.2.1  Primitive Operations
# calling C command procedures (np is a registered cmd procedure I added)
print-results "callback: 0 args"  [ntime {np} [n-times 300]]
print-results "callback: 1 arg"   [ntime {np a1} [n-times 300]]
print-results "callback: 4 args"  [ntime {np a1 a1 a1 a1} [n-times 300]]
print-results "callback: 8 args"  [ntime {np a1 a1 a1 a1 a1 a1 a1 a1} [n-times 300]]

# "concat five values"
set bob 2
set ted 4
set alice 9
set mary "A long string"
set leslie text
print-results "concat 5 vars" [ntime {set a $bob$ted$alice$mary$leslie} [n-times 200]]

# 8.2.2 Loops
print-results "for loop: 1->10000" [ntime {for {set counter 0} {$counter<10000} {incr counter} {}} 10]
print-results "while loop: sum first thousand ints" [ntime {set counter 0; set sum 0;
    while {$counter<1000} {
      incr sum $counter; incr counter
    }
} 5]

# 8.2.3  Procedure Calls
proc p0 {} {}
proc p1 {a} {}
proc p4 {a b c d} {}
proc p8 {a b c d e f g h} {}

proc varfun {args} {}

proc d1 {{a a1}} {}
proc d4 {{a a1} {b b1} {c c1} {d d1}} {}
proc d8 {{a a1} {b b1} {c c1} {d d1} {e e1} {f f1} {g g1} {h h1}} {}

print-results "proc: 0 args" [ntime {p0} [n-times 300]]
print-results "proc: 1 arg"  [ntime {p1 a1} [n-times 300]]
print-results "proc: 4 args"  [ntime {p4 a1 a1 a1 a1} [n-times 100]]
print-results "proc: 8 args"  [ntime {p8 a1 a1 a1 a1 a1 a1 a1 a1} [n-times 50]]

print-results "proc: 0 varargs"  [ntime {varfun} [n-times 100]]
print-results "proc: 1 varargs"  [ntime {varfun a1} [n-times 100]]
print-results "proc: 4 varargs"  [ntime {varfun a1 a1 a1 a1} [n-times 50]]
print-results "proc: 8 varargs"  [ntime {varfun a1 a1 a1 a1 a1 a1 a1 a1 } [n-times 40]]

print-results "proc: 1 default args"  [ntime {d1} [n-times 200]]
print-results "proc: 1 real args"  [ntime {d1 a1} [n-times 200]]
print-results "proc: 4 default args"  [ntime {d4} [n-times 100]]
print-results "proc: 4 real args"  [ntime {d4 a1 a2 a3 a4} [n-times 100]]
print-results "proc: 8 default args"  [ntime {d8} [n-times 100]]
print-results "proc: 8 real args"  [ntime {d8 a1 a2 a3 a4 a5 a6 a7 a8} [n-times 100]]

# note: I ommitted most of the a's to save space...
print-results "proc: 1k arg" [ntime {p1 aaa...aaa} [n-times 300]]

# 8.2.4  Array Variables
set varname(idxname) 123
set idxname index
```

```
set varname($idxname) 123
print-results "set varname(index)" [ntime {set varname(index)} [n-times 10]]
print-results "set varname(index) newvalue" [ntime {set varname(index) 123} [n-times 10]]
print-results "set varname(\$idxname)" [ntime {set varname($idxname)} [n-times 10]]
print-results "set varname(\$idxname) newvalue" [ntime {set varname($idxname) 123} [n-times 10]]

# 8.2.5  Math Expressions
set intvar 10
set zero 0
print-results "expr {\$intvar != \[set zero\]}" [ntime {expr {$intvar != [set zero]}} [n-times 10]]
print-results "expr {\$intvar != \$zero}" [ntime {expr {$intvar != $zero}} [n-times 10]]
print-results "expr {\$intvar != 10}" [ntime {expr {$intvar != 10}} [n-times 10]]
print-results "expr {\$intvar != 0}" [ntime {expr {$intvar != 0}} [n-times 10]]


# 8.2.6  Lists
set L1 { 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33
34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67
68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125
126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150
151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175
176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200
201 }
set L2 { mary bob joe mike sue joey kathy sinead susie }
set L3 { {mary bob} {joe mike} {sue joey {kathy sinead} susie} }

# get rid of parsing costs from timings...
lindex $L1 1
lindex $L2 1
lindex [lindex [lindex $L3 2] 2] 1

print-results "llength \$L2" [ntime {llength $L2} [n-times 300]]
print-results "llength \$L1" [ntime {llength $L1} [n-times 300]]

print-results "lindex \$L1 0" [ntime {lindex $L1 0} [n-times 100]]
print-results "lindex \$L1 100" [ntime {lindex $L1 100} [n-times 100]]
print-results "lindex \$L1 200" [ntime {lindex $L1 200} [n-times 100]]
print-results "lindex \$L1 5000" [ntime {lindex $L1 5000} [n-times 100]]
print-results "lindex \$L3 2" [ntime {lindex $L3 2} [n-times 100]]
print-results "lindex \[lindex \[lindex \$L3 2\] 2\] 1\]" [ntime {lindex [lindex [lindex $L3 2] 2] 1}
[n-times 50]]

print-results "foreach item {a} {}" [ntime {foreach item {a} {}} [n-times 100]]
print-results "foreach item \$L2 {}" [ntime {foreach item $L2 {}} [n-times 100]]
print-results "foreach item \$L1 {}" [ntime {foreach item $L1 {}} [n-times 100]]

print-results "lrange \$L1 3 8" [ntime {lrange $L1 3 8} [n-times 100]]
print-results "lrange \$L1 153 158\"" [ntime {lrange $L1 153 158} [n-times 7]]
print-results "lrange \$L1 3 102" [ntime {lrange $L1 3 103} [n-times 7]]

print-results "lsearch \$L2 joey" [ntime {lsearch $L2 joey} [n-times 50]]
print-results "lindex \$L2 \[lsearch \$L2 joey\]" [ntime {lindex $L2 [lsearch $L2 joey]} [n-times 50]]
print-results "lindex \$L1 \[lsearch \$L1 150\]" [ntime {lindex $L1 [lsearch $L1 150]} [n-times 30]]

print-results "building L2 with lappend" [ntime {
    set newlist ""
    foreach element $L2 {
      lappend newlist $element
    }
} [n-times 10]]

print-results "building L1 with lappend" [ntime {
    set newlist ""
    foreach element $L1 {
      lappend newlist $element
    }
} [n-times 1]]

print-results "sorting L2" [ntime {lsort $L2} [n-times 40]]
print-results "sorting L1" [ntime {lsort $L1} [n-times 2]]
```

```
# 8.3 Macro and aggregate benchmarks
# "factorial(n), recursive"
# note: can't do ?: trick bec. tcl will try to eval the else clause...
proc fac {n} {if {$n<=1} {return $n} {return [expr {$n*[fac [expr {$n-1}]]}]}}
print-results "recursive fac(1)" [ntime {fac 1} [n-times 30]]
print-results "recursive fac(4)" [ntime {fac 4} [n-times 7]]
print-results "recursive fac(7)" [ntime {fac 7} [n-times 3]]

# "factorial(n), iterative"
proc iter-fac {number} {
    for {set fac 1} {$number != 0} {incr number -1} {
      set fac [expr {$fac * $number}]
    }
    return $fac
}
print-results "iter fac(1)" [ntime {iter-fac 1} [n-times 30]]
print-results "iter fac(4)" [ntime {iter-fac 4} [n-times 7]]
print-results "iter fac(7)" [ntime {iter-fac 7} [n-times 3]]

proc fib {n} {
  if {$n<2} then {
        return $n
  } else {
        return [expr {[fib [expr {$n-1}]]+[fib [expr {$n-2}]]}]
  }
}

print-results "fibonacci(5), recursive" [ntime {fib 5} 1000]
print-results "fibonacci(10), recursive" [ntime {fib 10} 100]

print-results "list->array" [ntime {
    set isIndex 1
    foreach item $L1 {
      if {$isIndex} {
        set theIndex $item
        set isIndex 0
      } else {
        set array($theIndex) $item
#       puts stdout "array($theIndex) = $item"
        set isIndex 1
      }
    }
} 100]


# 8.4  Effects of whitespace and long identifiers

# factorial.  for well-written version, see above.
proc f {n} {if {$n<=1} {return $n} {return [expr {$n*[f [expr {$n-1}]]}]}}
print-results "f(1)" [ntime {f 1} [n-times 30]]
print-results "f(4)" [ntime {f 4} [n-times 7]]
print-results "f(7)" [ntime {f 7} [n-times 3]]

print-results "list->array" [ntime {
    set isIndex 1
    foreach item $L1 {
      if {$isIndex} {
        set theIndex $item
        set isIndex 0
      } else {
        set array1($theIndex) $item
        set isIndex 1
      }
    }
} 100]
print-results "list->array" [ntime {set x 1;foreach i $L1 {if {$x} {set d $i;set x 0} {set r($d)
$i;set x 1}}} 100]

set longerName 123
set a 123
print-results "set longerName" [ntime {set longerName} [n-times 300]]
print-results "set a" [ntime {set a} [n-times 300]]
print-results "set longerName 123" [ntime {set longerName 123} [n-times 200]]
print-results "set a 123" [ntime {set a 123} [n-times 200]]
```

Memories

See that picture
on the wall.
That's my grand-ma.
Pictures can paint
many, many words-
some more
some less.
I remember our card games
and her sweet nagging.

- in memory of my grand-ma,
Sylvia Feldsher (1912-1984).
-A.Sah'84

The 6:01 Commute

Break-run through the streets of Manhattan;
Fly into the rush hour;
track number... track number... 13! GO!

Riverrun of people pushing into the cars
whoosh! thunk! of closing doors
mish-mash of “personal belongings”,
the adjusting into the window seat.

and the grey-suit businessman next to you,
beer in one hand, briefcase at his feet,
chugs away at the last drops
of what is now only backwash.

and the conductor proclaiming “All Tickets Please,”
as row after row of ordinary people
flash their monthlies
like a membership card to some elite club
that they wished they weren't part of.

and my big, red mohair scarf under my head
as I hug my jacket,
and slide to sleep.

-A.Sah'92

In a Station of the Metro

The apparition of these faces in the crowd;
petals on a wet, black bough.

- Ezra Pound, 1916

Acknowledgements

Most of the design described in this paper has been implemented using equipment at UC Berkeley. The primary development machine was an Alpha-based DECstation 3000/400. Other auxiliary machines included HP 720 “Snakes”, a Sun 3/80, and several DECstation 3100’s. Thanks go out to UC Berkeley Experimental Computing Facility (XCF) and the Berkeley Plasma Theory and Simulation Group (PTSG).

I would like to thank John Ousterhout for his many hours of patient support, guidance and flexibility. Thanks to Paul Hilfinger for acting as second reader. Thanks to Jon Blow for help developing many of the key ideas and his work on the proto-proto-type that preceded this implementation. Thanks to Raph Levien, Chris Long, Brian Dennis, Sue Graham and the crew of the Tcl/Tk lunchs for friendship and numerous ear-pulling sessions. Thanks to Brent Welch for offering to help continue this work and for comments on an early draft. Thanks to the numerous members of sfraves and related, who provided some sanity checks along the way. Lastly, thanks also to my family and old friends in New York, for dealing well with my 3,000 mile relocation, and even coming to visit occasionally.