# Integrated Tcl/C Performance Profiling

Brian Griffin and Chuck Pahlmeyer
Mentor Graphics
8005 SW Boeckman Road
Wilsonville, OR 97070
brian_griffin@mentor.com
chuck_pahlmeyer@mentor.com

## ABSTRACT

We present an approach to combine compiled (C) and interpreted (Tcl) call stack information for profiling purposes. We have integrated Tcl proc and C function calls into combined call stack information to provide a more complete picture of program state at profile sample times.

## 1. Introduction and Motivation.

According to mathematician Richard Hamming, "The purpose of computing is insight, not numbers." (Hamming, 1962). In that light, we use performance profiling to gain insight into where our program is spending time. The goal is to provide that "Ah ha!" that leads to finding and correcting a performance problem. To that end, we have devised a method to provide greater insight in our profiling results.

*What is the problem?*
Adequate performance is an important attribute for many software applications. Profiling is a useful tool to provide insight into where a program is spending time. A statistical profiler records the program call stack at regular intervals and collates information to provide statistics on number of samples encountered in various parts of the program. Examples of statistical profilers include Zoom, Oprofile, gprof, google-perftools and Intel® VTune™. Even gstack and gdb can be used as crude statistical profilers; simply do "gstack <pid>" a number of times during program execution or interrupt execution and retrieve the call stack periodically in gdb. Additionally there exist instrumenting profilers which work by modifying function entry and exit. These events are recorded during program runtime and later collated into reports of time and function call count. Examples of this include callgrind, DTrace and gprof (gprof uses both statistical as well as instrumented approaches).

There are also Tcl-specific performance profilers. Although the authors are unaware of any statistical Tcl performance profiler, the ActiveState® Tcl Dev Kit includes a Tcl profiler. DTrace can also be used for Tcl profiling. These profilers instrument the code and record information about every call. This information is summarized into various profile reports.

However, Tcl applications often have a combination of C and Tcl. A section of call stack of C code representing execution of Tcl code shows up in a standard profiler as shown in Figure 1. This provides very little insight into the Tcl language calls that were being executed.

```
Tcl_Eval
  Tcl_EvalEx
    TclEvalObjvInternal
      Tcl_IfObjCmd
        Tcl_EvalObjEx
          TclCompEvalObj
            TclExecuteByteCode
              TclEvalObjvInternal
                Tcl_CatchObjCmd
                  Tcl_EvalObjEx
```

**Figure 1 Sample call stack representing Tcl code evaluation**

*Why is it interesting and important?*
To understand what portion of a program is involved in time consuming activities, it is essential that the profile results provide reference to the original source code, whether that code was compiled (e.g. C) or interpreted (e.g. Tcl). Existing profilers provide information for either the compiled or interpreted code, but not both together.

*Why is it hard?*
The Tcl interpreter is a collection of C functions that execute Tcl command. Tcl command execution shows up in a standard call stack as a series of C function calls with names like `TclExecuteByteCode` and `TclEvalObjvInternal`. The difficulty in providing an integrated call stack is in knowing which C functions in the call stack are executing which Tcl procs. The C call stack alone provides insufficient information for this.

*What are the key components of this approach and results?*
In order to provide integrated C and Tcl call stack information, it is necessary to reference auxiliary information about the state of the Tcl call stack. Our approach uses a standard statistical profiler which collects and processes program call stacks at intervals during program execution. In addition, we maintain a representation of the Tcl call stack as Tcl calls are being made. When a C call stack is processed to include Tcl command entries, the Tcl call stack information is used to replace entries on the C call stack with the appropriate Tcl commands. We correlate position in the C call stack with position in the Tcl call stack by tracking how many `TclEvalObjvInternal` calls have been encountered in processing the call stack. This allows us to present profile results with a combination of C functions and Tcl procs.

## 2. Integrating Tcl procs into call stacks

Mentor Graphics' Questa® simulator has a built-in statistical performance profiler. It uses a timer-driven mechanism to collect call stack information at regular intervals. It provides a user interface to allow interactive exploration of profile results. However, prior to this work, it reported only C-language call stacks, providing limited usefulness for our application which is written in Tcl and C. To get better insight into program hotspots, we sought to integrate Tcl procs into C call stacks.

### 2.1 Overview

To collect and store Tcl call stacks, we utilize `Tcl_CreateObjTrace` to set up a trace function for Tcl command execution. For each Tcl command that is executed, a call to our specific trace function is made. In the trace function, we store a textual version of the command at the level-th location in a static array. When we process a C call stack, we can replace every `TclEvalObjvInternal` call with the corresponding Tcl command. We maintain correlation between the C and Tcl call stacks by matching on each `TclEvalObjvInternal` call—that is, each `TclEvalObjvInternal` call maps to an entry in the `tcl_stack`. A counter is incremented for each `TclEvalObjvInternal` call—the counter value is used to address into the static array of Tcl procs. Figure 2 provides an overview of the processing involved. Simplified code segments in Section 2.2 below provide more detail.

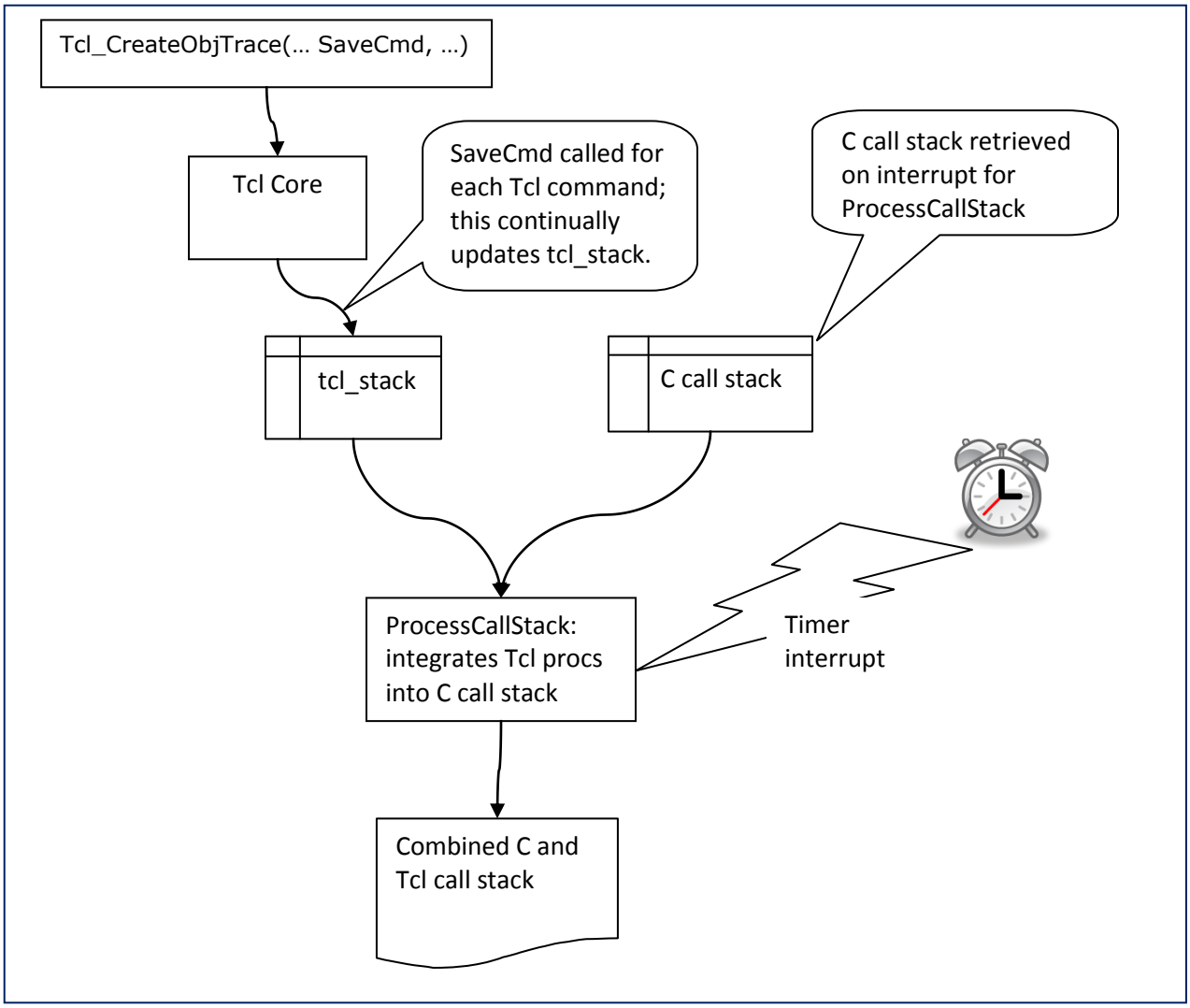


**Figure 2 Flowchart indicating overview of profiling operation**

## 2.2 Implementation

The Tcl call stack (`tcl_stack`) is stored in this compact structure (Figure 3); it consumes only a few tens of KB.

```
typedef struct t_tcl_stack {
    char command[80];
} s_tcl_stack, *p_tcl_stack;
s_tcl_stack tcl_stack[300];
```

**Figure 3 Storage for Tcl stack**

A simplified version of the trace function is shown below in Figure 4. `tcl_stack_max_loc` is a global variable indicating the maximum depth of the Tcl call stack at any moment. This variable is used in the call stack processing routine (`ProcessCallStack`). Note that the processing required in the trace function is small; this is essential since this function is called for each Tcl command that is evaluated. In the example in Section 3 below, about 2,800 profile samples were collected. During this process, `SaveCmd` was called about 21,000,000 times. The vast majority of entries that `SaveCmd` made into `tcl_stack` were unused. The calls to `SaveCmd` were necessary, though, to keep `tcl_stack` current because a profile sample could be taken at any time.

```
int SaveCmd(
    ClientData clientData,
    Tcl_Interp* interp,
    int level,
    const char *command,
    Tcl_Command commandToken,
    int objc,
    Tcl_Obj *const objv[] )
{
    /* There are cases where Tcl trace skips levels in the call stack
     * callback.  Fill any intermediate levels with entries that will be
     * skipped in ProcessCallStack(). */
    for (i=tcl_stack_max_loc+1; i<level && i<300 ; ++i) {
        strcpy(tcl_stack[i].command, "SkippedTclStackEntry");
    }

    if (level < 300) {
        strcpy(tcl_stack[level].command, command);
    }
    tcl_stack_max_loc = level;
    return TCL_OK;
}
```

**Figure 4 SaveCmd trace function implementation**

A simplified version of the call stack processing routine is presented in Figure 5 below. This function is executed for each call stack that is collected and processed. Processing of the call stack starts from the root (e.g. “main” (or “vish_inner_loop” for Questa)) and proceeds toward the leaf function call. The `while` loop processes each entry of the call stack. Each entry is handled in one of three ways:

- We replace `TclEvalObjvInternal` with the corresponding Tcl proc name. We do some manipulation of the reported text depending on the Tcl command being processed. For example, if a `string` command is being processed, we add one or two arguments to make a more informative entry. Also, some Tcl commands like `if` and `foreach` are ignored because we felt that they didn’t add useful information to the call stack.
- Items on the call stack that match “Tcl*”, “Itcl*”, etc. are ignored as they don’t add to our understanding of the processing.
- Other C function name entries are taken as-is.

```
static int ProcessCallStack()
{
    int tcl_stack_loc = 1;
    char *name, *proc;

    while (more entries in call stack) {
       name = name of call stack entry;
        if (name == "TclEvalObjvInternal") {
            char *cmd[4] = { 0 }, lcmd[80];
            strcpy(lcmd, tcl_stack[tcl_stack_loc].command);
            ++tcl_stack_loc;

            cmd[0-3] = first 4 tokens of lcmd

            if ((cmd[0]==0)   ||  (cmd[0][0]==0)                    ||
                (strcmp (cmd[0], "::"                          )==0) ||
                (strcmp (cmd[0], "if"                          )==0) ||
                                ...
                (strncmp(cmd[0], "Transcript::ReturnKey",    21)==0)) {
                proc = NULL; /* Ignore these Tcl commands */
            } else {
                /* Manipulate names for better info in displayed callstack */
                if (cmd[1]            &&
                    ((strcmp ("add"   ,      cmd[0])==0) ||
                           ...
                     (strncmp("."     ,    cmd[0],1)==0)))  {
                    proc = dstrPrintf(&ds, "%s++%s", cmd[0], cmd[1]);
                } else if ((strcmp ("string", cmd[0])==0) ||
                           (strcmp ("switch", cmd[0])==0)) {
                    if ((cmd[1][0]=='-')) {
                        dstrPrintf(&ds, "%s++%s++%s", cmd[0], cmd[1], cmd[2]);
                    } else {
                        dstrPrintf(&ds, "%s++%s", cmd[0], cmd[1]);
                    }
                    proc = dstrValue(&ds);
                   ...
                } else {
                    proc = cmd[0];
                }
            }
        } else if ((strncmp(name, "Tcl", 3) == 0)            ||
                   (strncmp(name, "Itcl", 4) == 0)           ||
                      ...
                   (strcmp (name, "WindowEventProc") == 0))   {
           proc = NULL; /* Ignore these functions in callstack */
        } else {
            proc = name; /* Save other non-Tcl C-level code addresses */
        }
        if (proc) addToDisplayedCallStack(proc);
    }
}
```

**Figure 5  ProcessCallStack implementation**

## 3. Results

We can compare results using standard C call stacks versus ones with Tcl commands substituted. The Tcl code in Figure 6 was used as a simple test case. It exercises the `tok2column` proc – code built into the Questa simulator. `tok2column` is designed to tokenize a string. The rules for tokenizing are different depending on the HDL language in use; in this case we specify "Verilog" as the language.

```
proc doWork { howMany }  {
    set l [list]
    for {set i 0 }  { $i<$howMany }  { incr i } {
        set l [doWork2  $i]
    }
    puts $l
}
proc doWork2 { i  }  {
    set line "The quick brown fox jumps over the lazy dog."
    set l [tok2column Verilog 23 $line]
    return $l
}
time { doWork 1000000 }
```

**Figure 6 Test Tcl code**

`tok2column` is a small routine that runs quickly (less than 50 microseconds). Calling it many times allows us to collect profile statistics and analyze it. In this case we call `tok2column` 1,000,000 times in 44 seconds collecting about 2,800 samples in the process.

## 3.1 Profile Results

The contents of the table below (Figure 7) show the `ProcessCallStack` processing for a portion of a representative call stack. The three columns in the table are:

- The C call stack entries. The `TclEvalObjvInternal` entries are shown in red; these are the items on which we base our C function to Tcl proc mappings.
- The processing done for each entry:
    - "`-->`" means that the C function was taken verbatim.
    - "X" means that the entry was filtered out.
    - "map to …" means that the entry was mapped to a Tcl command. Note that two Tcl commands (`time` and `for`) were suppressed though.
- The entries in the combined C and Tcl call stack.

Note the difference in length of the two call stacks. The combined C and Tcl call stack is much more compact.

| C function only call stack entry | ProcessCallStack action | Combined C and Tcl call stack entry |
|---|---|---|
| vish_inner_loop | --> | vish_inner_loop |
| Tk_MainEx | --> | Tk_MainEx |
| Tk_MainLoop | --> | Tk_MainLoop |
| Tcl_DoOneEvent | X | |
| Tcl_ServiceEvent | X | |
| WindowEventProc | X | |
| Tk_HandleEvent | X | |
| TkBindEventProc | X | |
| Tk_BindEvent | X | |
| Tcl_EvalEx | X | |
| (lines of Tcl*) | X | |
| TclEvalObjvInternal | map to Tcl command | .vcop++Action |
| (lines of Tcl*) | X | |
| Tcl_CatchObjCmd | X | |
| TclEvalObjEx | X | |
| TclCompEvalObj | X | |
| TclExecuteByteCode | X | |
| TclEvalObjvInternal | map to Tcl command | EvalUserCmd |
| tclprim_UserEval | --> | tclprim_UserEval |
| Tcl_EvalObjEx | X | |
| TclEvalObjEx | X | |
| TclCompEvalObj | X | |
| TclExecuteByteCode | X | |
| TclEvalObjvInternal | map to “time”, but suppress | |
| Tcl_TimeObjCmd | X | |
| Tcl_EvalObjEx | X | |
| TclEvalObjEx | X | |
| TclCompEvalObj | X | |
| TclExecuteByteCode | X | |
| TclEvalObjvInternal | map to Tcl command | doWork |
| TclObjInterpProc | X | |
| TclObjInterpProcCore | X | |
| TclExecuteByteCode | X | |
| TclEvalObjvInternal | map to “for”, but suppress | |
| Tcl_ForObjCmd | X | |
| TclEvalObjEx | X | |
| TclCompEvalObj | X | |
| TclExecuteByteCode | X | |
| TclEvalObjvInternal | map to Tcl command | doWork2 |
| TclObjInterpProc | X | |
| TclObjInterpProcCore | X | |
| TclExecuteByteCode | X | |
| TclEvalObjvInternal | map to Tcl command | tok2column |
| TclInvokeStringCommand | X | |
| tclprim_tok2column | --> | tclprim_tok2column |
| lang2lang_type | --> | lang2lang_type |
| Tcl_Eval | X | |

**Figure 7 Processing of example C call stack to combined C and Tcl call stack**

The outputs in Figures 8 and 9 are profile results from the Questa simulator. The contents consist of

1) function name,
2) number of samples in and beneath the function (Under column), and
3) number of samples in the function (In column).

The indentation of the function names indicates calling hierarchy. For example, if function A called function B, B would be shown indented one space with respect to A. The number of samples is used to understand the cost of that function, with and without its children.

This output in Figure 8 shows the depth of a standard C call tree report; multiple call stacks collated together form a call tree. The items hand-annotated with ">>>" prefix and in larger font are Questa-supplied C routines. Note that 110 and 50 lines of "Tcl*" entries were suppressed for readability. Without that substitution, the report would be 224 lines long.

```
Name                                                                  Under(raw)  In(raw)
----                                                                  ----------  -------
vish_inner_loop                                                             2795        0
 Tk_MainEx                                                                  2795        0
  Tk_MainLoop                                                               2795        0
   Tcl_DoOneEvent                                                           2795        0
    Tcl_ServiceEvent                                                        2795        0
     WindowEventProc                                                        2795        0
      Tk_HandleEvent                                                        2795        0
       TkBindEventProc                                                      2795        0
        Tk_BindEvent                                                        2795        0
         Tcl_EvalEx                                                         2795        0
          (110 lines of Tcl* suppressed)
          Tcl_CatchObjCmd                                                   2787        0
           TclEvalObjEx                                                     2787        0
            TclCompEvalObj                                                  2787        0
             TclExecuteByteCode                                             2787        0
              TclEvalObjvInternal                                           2787        0
>>>>>>>>>> tclprim_UserEval                                                2787        0
              Tcl_EvalObjEx                                                 2787        0
               TclEvalObjEx                                                 2787        0
                TclCompEvalObj                                              2787        0
                 TclExecuteByteCode                                         2787        0
                  TclEvalObjvInternal                                       2787        0
                   Tcl_TimeObjCmd                                           2787        0
                    Tcl_EvalObjEx                                           2787        0
                     TclEvalObjEx                                           2787        0
                      TclCompEvalObj                                        2787        0
                       TclExecuteByteCode                                   2787        0
                        TclEvalObjvInternal                                 2787        0
                         TclObjInterpProc                                   2787        0
                          TclObjInterpProcCore                              2787        0
                           TclExecuteByteCode                               2787        0
                            TclEvalObjvInternal                             2787        0
                             Tcl_ForObjCmd                                  2787        3
                              TclEvalObjEx                                  2747        4
                               TclCompEvalObj                               2743        7
                                TclExecuteByteCode                          2712       20
                                 TclEvalObjvInternal                        2679       10
                                  TclObjInterpProc                          2442        0
                                   TclObjInterpProcCore                     2436        3
                                    TclExecuteByteCode                      2399       17
                                     TclEvalObjvInternal                    2381       13
                                      TclInvokeStringCommand                2082        2
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> tclprim_tok2column                        2072        1
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> lang2lang_type                           1843        4
                                        Tcl_Eval                            1750        3
                                         (50 lines of Tcl* suppressed)
                                        sprintf                               28        0
                                         _IO_vsprintf                         28        2
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> HDLTextTok2Col                             221        3
>>>>>>>>>>>>>>>>>>>>>>>>>>>>> yylex                                       173       39
>>>>>>>>>>>>>>>>>>>>>>>>>>>> is_keyword                                   96       54
                                    TclCheckInterpTraces                   163       10
                                     Tcl_Release                            41       41
                                     Tcl_Preserve                           37       37
                                    GetCommandSource                        41        2
                                    Tcl_ReturnObjCmd                        28        0
                                   TclCheckInterpTraces                    125        8
                                    Tcl_Release                             38       38
                                    Tcl_Preserve                            34       34
                                  Tcl_ExprBooleanObj                        34        0
                                   Tcl_ExprObj                              34        2
                                    TclExecuteByteCode                      32       15
```

**Figure 8  Profile results with C call stack entries only**

The profile report in Figure 9 below is one in which our approach has been used to replace "Tcl*" entries with the actual Tcl procs that were being evaluated. This is a screen shot of one of the profile windows in the Questa user interface; the profiler windows allow user interaction with profile data to better manage what is viewed. In this figure, C functions and Tcl commands can be distinguished by the different icons associated with each. The same six entries of Questa supplied C-code can be easily found. Note the interleaving of C functions and Tcl procs. For example, `doWork`, `doWork2` and `tok2column` are implemented in Tcl, `tclprim_tok2column and lang2lang_type` are C functions, `lang2lang_type` calls Tcl proc `::MtiFS::IsVerilogLanguage`, etc.

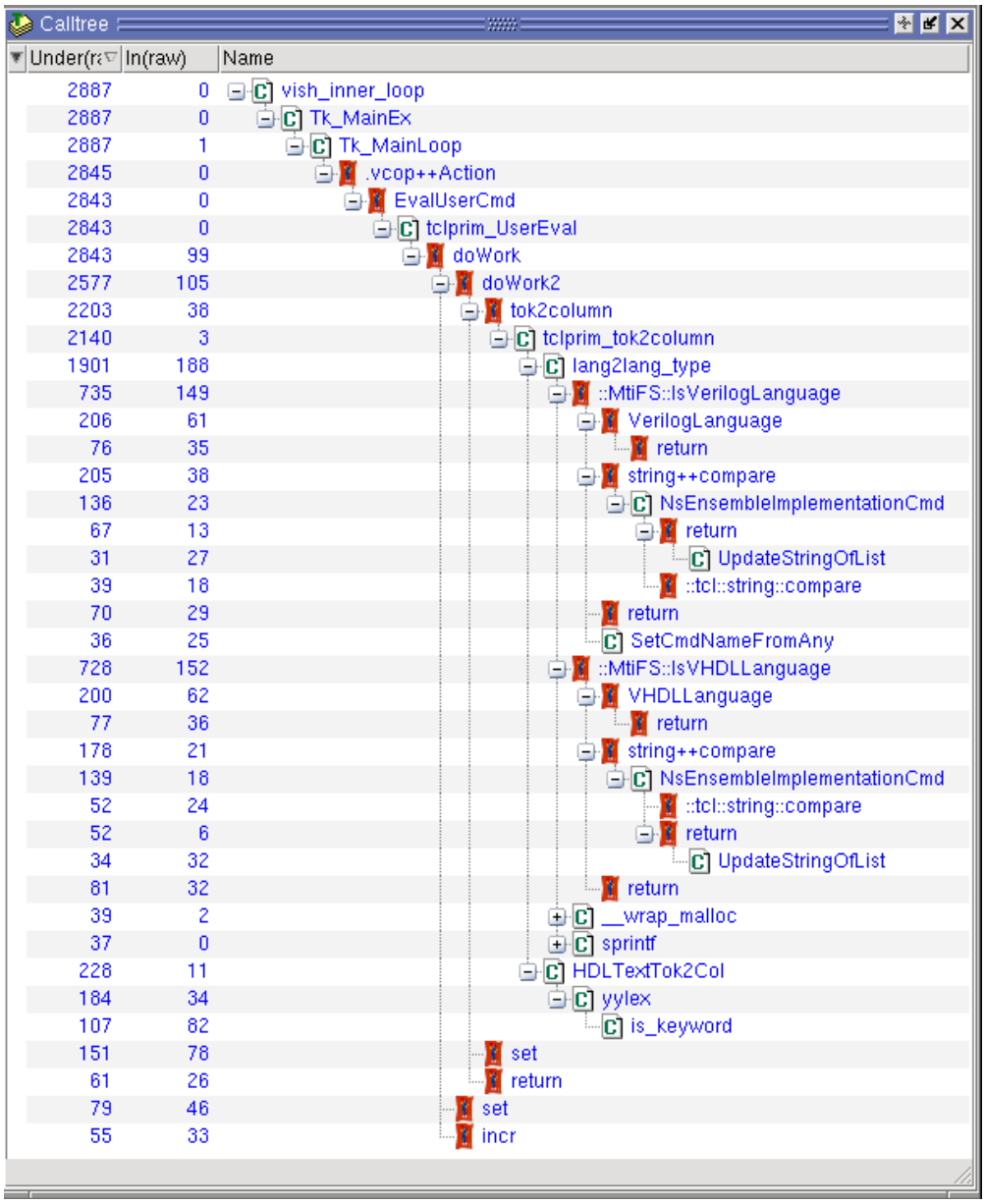


**Figure 9 Call tree with Tcl and C entries**

## 3.2 Using Profile Results to Analyze Performance

We can use this profile data to analyze performance of the `tok2column` proc. In looking at the children of Tcl proc `tok2column`, we see that C function `tclprim_tok2column` takes the majority of `tok2column's` time. We see that `tclprim_tok2column` spends time in two child routines: `lang2lang_type` and `HDLTextTok2Col`. The `lang2lang_type` routine is designed to determine the language type of the incoming language argument; `HDLTextTok2Col` does the actual tokenizing. We can see that `lang2lang_type` takes 1901 samples while `HDLTextTok2Col` takes only 228 – `tclprim_tok2column` is spending 8 times as much time to interpret the language argument as doing the actual tokenizing that the routine nominally does! We look further at the components of `lang2lang_type`: `MtiFS::IsVerilogLanguage` and `MtiFS::IsVHDLLanguage` and see their costs. We can examine the source code of these functions and procs to understand if unneeded work is being done or if necessary work could be done more efficiently; see Figure 10.

```
static int lang2lang_type (Tcl_Interp *interp,const char *lang)
{
      char buf[256];
      sprintf(buf, "::MtiFS::IsVHDLLanguage %s", lang);
      if ( Tcl_Eval(interp, buf) == TCL_OK) {
            if (Tcl_GetIntResult(interp)) {
                  Tcl_ResetResult(interp);
                  return LANGVHDL;
            }
      }

      sprintf(buf, "::MtiFS::IsVerilogLanguage %s", lang);
      if ( Tcl_Eval(interp, buf) == TCL_OK) {
            if (Tcl_GetIntResult(interp)) {
                  Tcl_ResetResult(interp);
                  return LANGVERILOG;
            }
      }
     . . .

proc MtiFS::IsVerilogLanguage { type } {
      if {[string compare -nocase $type [VerilogLanguage]] == 0 } {return 1}
      return 0
}
proc MtiFS::VerilogLanguage {}       { return "verilog" }
```

**Figure 10 Source code for C function and Tcl procs used by test case**

Since we specify "Verilog" as an argument to `tok2column`, we expect `lang2lang_type` to return `LANGVERILOG`. We can inspect `lang2lang_type` and see why `IsVHDLLanguage` and `IsVerilogLanguage` both show up with about the same costs.

The invocation of Tcl procs from C code to determine language type is time-consuming. With this insight into where time is spent in `tok2column`, we can easily reimplement `lang2lang_type()` as a strictly C function to speed `tok2column` considerably.
This example demonstrates the utility of combining Tcl and C routines into a single call tree for performance analysis. Note that `tclprim_tok2column`, `lang2lang_type` and `HDLTextTok2Col` do show up in the first (C function only) profile output (Figure 8). Without the context of the surrounding Tcl procs, though, it is more difficult to understand their role in the overall performance picture.

We have used this profiling feature to track a number of issues in the Questa simulator. For example:

- Tracking GUI sluggishness at a remote customer site. The developer initially suspected the message viewer was involved due to a high number of messages being processed. However, the profiler showed that code that scans simulation events was actually consuming the majority of the time. With this information, the developer was able to understand and address the real problem.
- We've recently integrated the Scintilla editor. Certain of our regression tests ran quite slowly using this editor. The profiler was able to point to the portion of the code that was consuming excess time.

## 4. Limitations and Notes

Our approach required a few small changes to the Tcl core code in order to get a consistent value for `numLevels` that could be matched to depth of `TclEvalObjvInternal` on the call stack. However, these changes weren't completely compatible with other uses of the notion of `numLevels` in the Tcl library. We're using these modifications in the Questa simulator version of Tcl, but they haven't been propagated to the public Tcl version.

We found that if `TCL_ALLOW_INLINE_COMPILATION` was specified in the flags argument to `Tcl_CreateObjTrace()`, the alignment assumptions made for this process were violated. That is, the assumption of number of times that `TclEvalObjvInternal` appeared in a call stack didn't match the depth of the Tcl call stack.

In our testing, doing profiling in this way caused a doubling to tripling of overall execution time. This is due to two primary factors:

- The cost of maintaining the Tcl call stack at all times via the trace functionality. The trace call is somewhat expensive due to internal overhead.
- Deoptimization of the Tcl code due to not specifying `TCL_ALLOW_INLINE_COMPILATION` to `Tcl_CreateObjTrace()`.

Although this is a non-trivial performance cost, the information it provides is generally worthwhile.

The filtering done in `ProcessCallStack` works well for our needs. Different rules could be used to support profiling in a different application or with interest in aspects of the Tcl library itself.

There were places where the trace function `SaveCmd` didn't appear to be called for every level of Tcl command that was executed. For example, the trace function might be called with `level`=12 followed by a call with `level`=15. In this case, we'd enter the value `SkippedTclStackEntry` into the entries 13 and 14 of the Tcl call stack (`tcl_stack`) so that a known value is present on the stack all the way to `tcl_stack_max_loc`, the deepest level of the Tcl call stack. This didn't occur frequently, but did require the handling we provided.

Certain techniques described in this paper are patent-pending as a patent application has been submitted to the US Patent Office.

## 5. Future work

In the Tcl 8.5 environment, it could be useful to have a lighter-weight function to record Tcl command calls. The `Tcl_CreateObjTrace` approach had a large overhead due to mutex locks that were used around each trace call.

In the Tcl 8.6 environment, "stackless evaluation" has been introduced. This will require a different mechanism to record location in Tcl command call stack for correlation with C call stack.

## Acknowledgements

We would like to thank Mark Young of Mentor Graphics for engaging discussions when developing this functionality. We would also like to thank the Questa GUI development team for testing this functionality.

## Bibliography

Hamming, R. (1962). *Numerical Methods for Scientists and Engineers.* McGraw-Hill Education.
*Tcl Library*. (n.d.). Retrieved August 2013, from Tcl Developer Xchange!: http://www.tcl.tk/man/tcl8.5/TclLib/contents.htm