Characterizing and Back-Porting Performance Improvement

Clif Flynt
Noumena Corporation
clif@noucorp.com

Phil Brooks
Mentor Graphics Corporation
phil_brooks@mentor.com

Don Porter
NIST
donald.porter@nist.gov

September 4, 2013

Abstract

The Tcl interpreter is constantly being modified and improved. Improvements include new features and performance boosts.

Everyone wants to use the latest releases with the newest improvements, but corporate users with large code bases may not be able to do this. Reworking an extremely large code base can take longer than the interval between Tcl releases. These users may need a change to be back-ported to the version of Tcl that they are using.

A Tcl release includes many changes and identifying the modification that caused a particular performance boost isn't always simple, particularly if the performance boost of interest was a side-effect of other improvements.

This paper describes the discovery of a thread-performance issue in Tcl 8.4 which was fixed in 8.5, a semi-automated technique for tracking down the code modification that improved the performance, and a discussion back-porting the improvement.

1 Introduction

The first rule for carpenters and seamstresses is "Measure twice, cut once". For software engineers the rule for optimization is just "Measure First".

Work reported at the Tcl Conference in 2005 and 2012 identified an area where Tcl performance was not living up to expectations and provided the critical benchmark to measure the interpreter performance behavior with multiple threads.

1.1 Tcl-2005: 'Pulling Out All the Stops'

Phil Brook's 2005 Tcl conference paper discussed implementation of Calibre LVS's Device TVF feature, in which a highly efficient, though quite limited, calculation engine is given the ability to make calls to a Tcl program allowing for more sophisticated programming capabilities.

1.2 Tcl-2012: 'Pulling Out All the Stops - Part II'

The follow up 2012 paper discussed implementation of threading capabilities on the 2005 calculation engine and identified a bottleneck that is present especially when the calculation engine is creating many strings and formatting numbers into strings. The behavior in the application was reproduced in a stand alone C++/Tcl test program that similarly created many strings from numbers. The application behavior and that of the stand alone benchmark bore telltale signs of a locking problem:

- Real time execution scaling tapers off and even gets worse with additional threads.
- System time required escalates with the number of threads as additional kernel intervention is required to resolve lock contention.

The following graph compares MT scaling across Tcl versions 8.4.19, and 8.5.14:

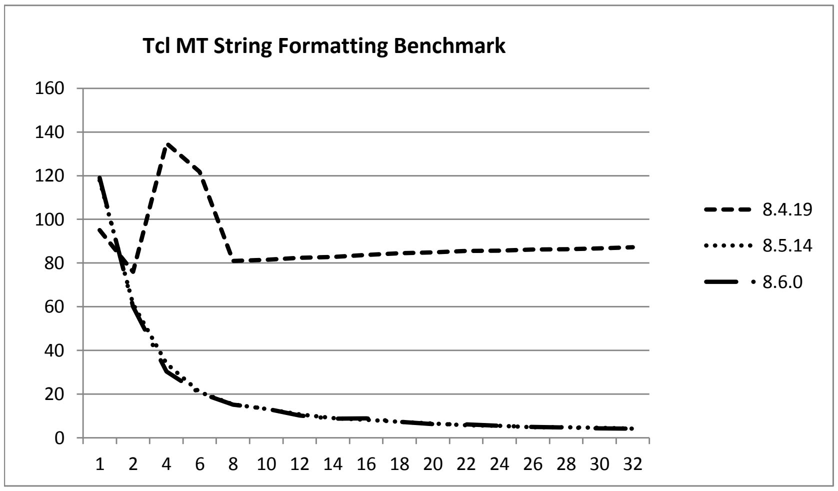

Figure 1: Application run time vs threads for Tcl 8.4, 8.5 and 8.6

When the stand alone test program was run against Tcl 8.5 and Tcl 8.6, the contention issue was not observed. That led to the question: What changed in Tcl that improved multi threading execution time? One other observation between Tcl versions 8.4, 8.5 and 8.6 was that the simple use of expr in a for loop was significantly slower in Tcl 8.5 and again slower in Tcl 8.6. This fact, along with the difficulty of porting a major application with over 1 million lines of Tcl code onto Tcl 8.5 or 8.6 led us to investigate finding the optimization and looking at the possibility of porting it back to Tcl 8.4.

2 Procedure

The benchmark program developed by Phil Brooks was used to characterize several Tcl check-ins. This benchmark performs a number of float-to-string conversions and allows the user to select the number of threads to use to perform the conversions.

The basic flow for performing the benchmarks was

```
For revision in List_Of_Fossil_Identifiers
  Checkout revision
  Build Tcl libraries
  Link benchmark with new libraries
```

```
   Run benchmark and record results
```

The `fossil finfo` command will return a list of check-ins in which a file has been modified. Since we were investigating threading behavior, it made sense to collect a set of fossil check-ins related to changes in the `generic/tclThreadAlloc.c` module with this command.

```
> mkdir tcl
> cd tcl
> fossil open ../tcl.fos
> fossil finfo generic/tclThreadAlloc.c >../ThreadFiles.fo
```

The output of the `fossil finfo` command resembles this:

```
2012-11-26 [1d357d342e] Merge (selected bits of) novem (user: dgp, artifact:
           [2314c6d7b8])
2012-11-26 [45a2eb8ff0] merge 8.5 (user: dgp, artifact: [58f949bc42])
2012-11-26 [ab9713b5f1] merge novem (user: dkf, artifact: [1b8015a219])
2012-11-26 [cdc837ae05] merge trunk (user: mig, artifact: [59ce13b7e5])
2012-11-26 [6b2cf92413] Removed functions marked deprecated or obsolete for a
           long time: Tcl_Backslash, Tcl_EvalFile, Tcl_GlobalEvalObj,
           Tcl_GlobalEval, Tcl_EvalTokens. Remove Tcl_FindExecutable from
           ...
```

The useful lines are the ones that start with a date stamp. The numbers inside square braces are the identifier for this check-in. This set of numbers can be used with `fossil checkout` to check out a specific version of the Tcl source code.

This set of code examines the `finfo` output and checks out Tcl revisions in the order that they were checked into fossil.

```
set top $pwd
foreach l [split $d \n] {
  cd $top

  # extract first 10 characters
  set dt [string range $l 0 10]

  # Sudden death tests - if not date, or no values, skip
  if {[catch {clock scan $dt}]} {continue}
  if {[string trim $dt] eq ""} {continue}

  # If the line starts with a date, get the ID number
  set num [string range $l 12 21]

  # Clean up a folder for new checkout
  catch {exec rm -rf tcl}
  file mkdir tcl
  cd tcl
```

```
# Checkout
exec fossil open ../tcl.fos
exec fossil checkout $num
```

Tcl development is a continuous process on all of the active releases and branches. The chronology of check-ins does not reflect the release or position within a release.

This code snippet opens the `tcl.h` file in the newly checked out sources to find the version number associated with this code.

```
cd tcl
set if [open generic/tcl.h r]
set tclh [read $if]
close $if

regexp {TCL_VERSION[^"]*"([^"]+)} $tclh all rev
```

Once the code is extracted, it needs to be compiled. This is trivial with the TEA based make system and is easily automated.

While Tcl releases are very stable, not every check-in is guaranteed to work, or even compile, so the `make` is executed within a `catch` command.

```
cd unix
exec ./configure --enable-threads -enable-shared=0
set fail [catch {exec make} rtn]
```

The `fail` value is used to determine whether or not to build the benchmark. An application is not guaranteed to be linkable after the components have been successfully compiled, and an application that links doens't necessarily run. The `catch` command gets a lot of use in the sections of the script that make and run the benchmark.

```
 cd $benchFolder

 # Modify Makefile for current release
 exec sed s/REL/$id/g <Makefile.in >Makefile

 # Clean and rebuild benchmark
 catch {exec make clean}
 set fail [catch {exec make}]

 for {set ii 0} {$ii < $runCount} {incr ii} {
   set t1 [clock seconds]
   set fail [catch {exec ./mt_example_$id -tcl -s $count1} rtn]
   regexp {.*REAL TIME=([^ ]+) .*} $rtn aa measure

   # Calculate average
 }
```

The Tcl `time` command was used to evaluate the run time for the benchmark. To protect from system inconsistencies, the benchmark was run multiple times and the average time was used to calculate the ratio between 2 and 6 threads.

The process of checkout, compile and test is time consuming, but will complete in a single night. Once the results were collected, they were graphed as shown below, demonstrating the dramatic performance improvement.

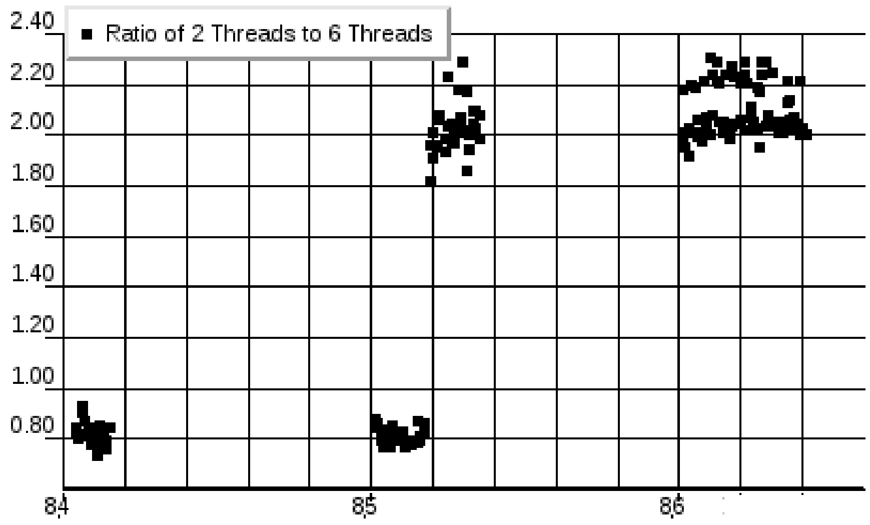

Figure 2: Runtime Ratio vs Tcl Release

The graph is nice for visualization, but does not show where the change occurred.

The raw data, however makes it fairly obvious:

```
...
3af2919289 8.5 RATIO 2.0
7ff2693241 8.5 RATIO 2.0
1cc2336920 8.5 RATIO 2.0
751ccc1989 8.5 RATIO 0.7
edf99c3880 8.5 RATIO 0.8
83aa957ebe 8.5 RATIO 0.8
...
```

In an ideal world, using 6 threads instead of using 2 threads would be result in a runtime reduced by a third, and the ratio would be 3. In the real world there are issues involved with making sure the threads don't collide, switching overhead, etc. The best improvement seen in this set of testing was a bit over 2.

However it's obvious that between the 751ccc1989 check-in and the 1cc2336920 the performance was improved. Examining the code changes, it turned out that the improved thread performance did not come from an improvement in the thread management code, but was a benefit from the merge of a large numerics reform branch.

2.1 Back-port

At this point, the search could easily continue by iterating the automated performance testing script on the merged numerics reform branch to find the precise change responsible. However, having narrowed the matter down to the changes in number handling was sufficient. For someone familiar with the structure and history of that portion of the Tcl source code, it is clear that the problem lies with the treatment of the `::tcl_precision` variable, and the improvement came with a strong move away from using that variable. Searching tools are a great assistant, but knowing the code-base is a key contributor as well.

The `::tcl_precision` variable has a long history in Tcl. It arrived in Tcl 7.0. Setting the variable to an integer value between 1 and 17 specified how many decimal digits of precision should be used when the routine `Tcl_PrintDouble` generates the string form of a floating point value. It had some value as a means to tune performance, with `sprintf()` presumably taking less time to generate shorter strings (and in the days when every value truly was a string, that could matter). However, its greater purpose is to shield unsophisticated Tcl programmers from some of the harsher realities of floating point arithmetic.

```
% expr {1.0/10}
0.10000000000000001
```

The aim is not unreasonable, but the error was in placing this feature in the heart of the value stream of Tcl, and not on the periphery where it would govern display matters only. The consequence over time was a large number of bug reports rooted in the fact that the operation of `::tcl_precision` was at odds with Tcl's value model that "Everything Is A String" (EIAS). An extreme example serves to demonstrate.

```
% set tcl_precision 1
1
% set third [expr {1.0/3}]
0.3
% set compare "0.3"
0.3
% string equal $third $compare
1
% expr {$third == $compare}
0
```

Two equal string values get treated as unequal by some Tcl commands. This is contrary to Tcl's value model that the string representation holds all the value there is to hold.

In the development of Tcl 8.5, one of the goals set and achieved was to make numeric values in Tcl properly conform to EIAS. Strictly speaking this is impossible to fully achieve while the `::tcl_precision` variable still exists and Tcl continues to follow the documented response to its value for sake of compatibility with existing scripts making use of it. (The example above uses a Tcl 8.6.0 interpreter.) However, the continued use of `::tcl_precision` is discouraged in the strongest terms, and the default setting is one that upholds EIAS and also avoids most of the shocking results that motivated its preservation through many earlier calls for its elimination.

```
% set tcl_precision
0
% expr {1.0/10}
0.1
```

Starting in Tcl 8, Tcl values are stored in a `Tcl_Obj` struct, and the `Tcl_PrintDouble` routine is normally only called for producing the string representation of a value of the `double` Tcl_ObjType. In this context, there is no Tcl_Interp to refer to, and so no way to pull a value out of any particular `::tcl_precision` variable. Consequently, in the Tcl 7 to Tcl 8 transition, the actual value controlling precision came to be stored outside of any interp, as one common static variable shared by the entire application. All the `::tcl_precision` variables in all the interps became ways to read and write that common global value through the magic of traces.

Then in Tcl 8.1, as the source code was revised to support multi-threaded operations, the common static variable holding an application wide value for controlling precision came to be shared among all threads, with mutex locking added to guarantee that all writes *and reads* of that value are serialized.

At this point the reader seasoned in multi-threaded programming is thinking "Aha! Of course multi-threaded performance collapses when every thread has significant amounts of double to string conversions to do!" The whole double to string machinery funnels through a serialization bottleneck. Relief came only in the development of Tcl 8.5, when two things happened. First, the entire subsystem for generating the decimal string representation of a floating point value was rewritten, changing the patterns of locking in a way that reduced the scaling problem. Second, an additional design change replaced the application-wide global value for controlling precision with a set of values, one for each thread. This eliminated any need for locking altogether.

Having identified the cause of the performance bottleneck, and the reasons that Tcl 8.5 and later avoid it, the next issue is what can be done in Tcl 8.4 to correct the problem. The solution came into 8.5 as part of a major subsystem rewrite. Forcing that into a patch release of Tcl 8.4 is contrary to the practice of making such major rewrites only with new minor versions. A simpler solution would be to back-port the shift from a single global value to a set of precision control values, one for each thread. However, this solution was avoided because it represents a level of compatibility change that exceeds the normal practice for patch level releases. The amount of incompatibility involved is just too great for those scripts that would notice.

Instead, a new solution was crafted, preserving the global precision control value, and improving performance by replacing the expensive simple-minded serialization scheme with a somewhat more complex scheme, but one far better matched to the actual needs and resulting in far better performance. In practice, the reads from the global precision control value are far more frequent than writes. A locking scheme that reduces the need for locking when only reading the value is highly effective in improving performance. The machinery to make this happen is an adapted back-port of the `ProcessGlobalValue` utility already used in Tcl 8.5 to manage other application-wide values like `[info hostname]` and `[info nameofexecutable]`.

A `ProcessGlobalValue` maintains a master string value, and also keeps a cached copy of that value in a Tcl_Obj for each thread. A master integer epoch value is also maintained, and a cached value of that epoch in each thread as well. When the master value is written, full mutex locks are used to serialize writes. The new master value is stored, and the master epoch is incremented. When there is a need to read the value, however, a scheme of epoch checking avoids the need for locks most of the time. The cached epoch value is compared with the master epoch value and so long as they are the same, the cached Tcl_Obj value is still valid, and the thread proceeds making use of it. Only when an epoch mismatch indicates that the master string value has been written since the last epoch check in a thread does the expense of a lock and a recopy from the master value to the thread cache take place, along with an update of the thread epoch value. This mechanism preserves the single global value that is a feature of Tcl 8.4, while bringing the cost of it down significantly. The patch to make this change was added, tested, and released as part of Tcl 8.4.20.

The change was incorporated into the original Calibre application and the resulting application performance clearly shows an improvement in scaling and continues to scale incrementally even with 32 threads running on a 32 way system as shown in the following graph:

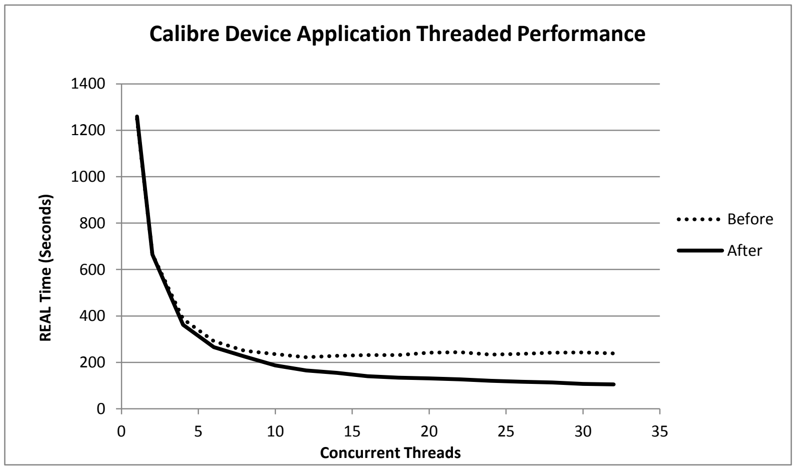

Figure 3: Application run time vs threads

2.2 Conclusion

Keeping Tcl users happy and improving the performance of the kernel is always good. What is interesting in this particular case is the techniques used to identify the time when a performance change occurred using a scripted set of checkouts and builds, and the cooperation of three entities (the three authors) in identifying the problem, reducing the problem set, and fixing the issue.

The technique of using a set of scripts to checkout, configure, build, test is fairly obvious and is applicable to any performance study. The scripts that checked out versions of the Tcl interpreter and built multiple copies are relatively simple.

2.3 Future Work

The scripts used to run this set of benchmarks have been modified and extended in a larger study of the behavior of Tcl with the `tclbench` suite to characterize Tcl behavior over a larger number of subsystems.