“Tcl cures 98.3% of all known simulation configuration problems” claims astonished researcher!

Richard Golding, Carl Staelin, Tim Sullivan, and John Wilkes

Hewlett-Packard Laboratories, Palo Alto, CA

HPL–CCD–94–11, 22nd April 1994

This paper was presented at the *Tcl/Tk workshop* in New Orleans, LA, 17–21 June 1994.

“Tcl cures 98.3% of all known simulation configuration problems” claims astonished researcher!

Richard Golding, Carl Staelin, Tim Sullivan, John Wilkes
Hewlett-Packard Laboratories, Palo Alto, CA
{golding, staelin, sullivan, wilkes}@hpl.hp.com

Abstract

We build detailed event-driven simulations of I/O systems as a way to earn a living, and use them to explore storage system architectures. Our major tool (the TickerTAIP simulator) is a large kit of parts that can be configured, combined, replicated, and connected in innumerable ways.Tcl lets us retain our sanity: it makes configuring the simulator a breeze compared to our prior techniques.

In this paper we describe the general approach we adopted, and a few of the tricks and idioms we had to develop to make it work. Our approach and techniques should be useful to anybody using Tcl as a control language for a set of underlying objects.

1 The problem

Our work exploring storage system architectures requires us to do lots of “what if …” kinds of analyses, and for this we’ve chosen to use detailed event-driven simulation. The simulator we use, TickerTAIP, is based loosely on the one described in [Cao93] and [Ruemmler94].

A major thrust of our work is comparative studies: “is it a good idea to tweak the *xxx* algorithm thusly…?” We also range over a largish number of different studies, such as disk request scheduling algorithms, file system designs, disk and smart-controller cache policies, adaptive data layout schemes, and so on.

As a result, we have developed a simulation environment that has about 90 different C++ object classes that can be glued together in different ways. Examples of these classes include:

- *Link*: encapsulates notion of bandwidth and setup overheads for use in a bus (e.g., SCSI);
- *DiskMechanism*: a task that represents a spinning magnetic disk, including timing calculations for operations like seeks, rotational latencies, and data transfer;
- *Cache*: a piece of memory into which data can be inserted;
- *IOsched*: an algorithm that decides which request should be serviced next if there are several (e.g. FIFO, CSCAN, SSTF, …);
- *DMAengine*: a task that moves data between different caches;
- *Disk*: the object that ties together a Link (for the bus speed), a DMAengine, a DiskMechanism, a request scheduler and a Cache.

Each of these objects can usually be configured further: for example, a Link takes a bandwidth and overhead, a Cache has a size and an allocation granularity. Indeed, the configuration data for Disks and DiskMechanisms are so complex that they are objects in their own right.

When we began, we wrote C++ code to control the configuration, using a wealth of command-line flags to select various design alternatives. This rapidly got out of hand: for example, in our current set of simulations for a new storage subsystem we are running about thirty different experiments, each of which compares a set of policy choices (each of which can sometimes have multiple parameter values) against a baseline.

Somewhere around the time we were running out of letters of the alphabet for our command-line interface, we happened across Tcl [Ousterhout94]. The result of that fortuitous event is the subject of this paper.

2 What we do

The basic approach we adopt has been used by others: Tcl scripts control the configuration of the simulator’s components; C++ simulation objects execute the simulation at full speed (sometimes for several hours!) Scripts also control coarse-grained execution of the simulator (such as deciding whether to run another batch or not) and reporting results. We have also put together a set of Tcl scripts for extracting data from our runs and plotting comparative analyses of a baseline configuration and a variant on it. (This package is fetchingly referred to as tongs.)

The differences are in the details: we believe we have evolved some techniques that are not immediately obvious, and that make using Tcl in this way much easier than it would otherwise be.

We've written about 5.3k lines of Tcl scripts so far; roughly a third of this being in the result-extraction and display code. The Tcl is used to manage roughly 40k lines of C++ source code; it took us about 1.4k lines of code to handle the interface between our simulator and the Tcl interpreter in addition to the per-class code used to create objects. The simulator has been in daily production use for several months.

3 Key ideas

This section introduces the key points in our approach.

3.1 Building and naming C++ objects

Each C++ simulation object class has a Tcl function (registered with Tcl as a command of the same name as the class) that builds a new instance of the class, constructs a unique name for it, and enters this into a hashed lookup table that maps the text name to a pointer to the new object. (This structure is very similar in intent to the object table used in SmallTalk interpreters [Goldberg83].) Thus:

```
set link [Link "SCSI-bus" -b 10e6]
```

constructs an object to represent a SCSI bus with a bandwidth (-b) of 10MB/s, generates a name-string for it (:Link:1:SCSI-bus:), and assigns this string to the Tcl variable link. The real trick is that we then treat the Tcl variables holding these name strings just like C++ pointers to the object itself: in our minds, we think that the value $link *is* the link object—obtained by dereferencing the name-string pointer in the Tcl way.

3.2 Dot functions

The first extension we made to this pointer idea was to provide "dot functions", in the style of C++. Thus:

```
$link.bandwidth
```

is a Tcl function that returns the currently-assigned bandwidth of the link, and

```
$link.bandwidth 20e6
```

sets it to a new value.

We found the regular C-to-Tcl interface somewhat verbose, and so resorted to some (highly stylized) macros to encapsulate the interface to them. For example:

```
TCL_ACCESS_FUNCTION_VOID_FUNCTION(Link, reset);
```

creates a C++ procedure that is used to reset stored statistics counters in a Link object, invoked by $link.reset.

3.3 Type checking

The unique names we generate include the class name of the object, and we construct our C++ class inheritance hierarchy using a rule that says a derived class is named by tacking something onto the end of the base class. Thus:

```
Cache
< CacheSegmented
< CacheSegmentedReplacement
```

is a portion of the Cache-object hierarchy. Combining this rule with the access macros allows us to provide dynamic type checking: an object expecting a CacheSpace can check that it hasn't been given a plain Cache (or even a Link!). For example, inside the DeviceDriver Tcl constructor function we find:

```
TCL_GETVAR(Disk, CacheSegmented, cache, argv[4]);
```

which says that the current Disk-object constructor is expecting a pointer to a CacheSegmented object (or something derived from it), which it should put in the variable cache. The "pointer" passed in is of course one of our name-strings, used to index the object table.

3.4 Statistics

Since it is easy to build C++ objects from Tcl, we use this approach rather than building any simulator objects in C++. One pervasive example is in statistics-gathering. Instead of building in statistics-gathering functions wherever they might be needed, we instead store only a pointer to an object from the Stats hierarchy, which looks like this:

```
Stats < StatsHistogram
```

A Stats object accumulates mean, count, standard deviation, and other statistical measures for a value. A StatsHistogram also keeps a density distribution. The Tcl code decides whether to create no Stats object at all, whether it should be a low-cost Stats object, or a more expensive StatsHistogram—and if the latter, how much storage it should use. The C++ code at a measurement point merely tests for a non-null pointer to a Stats object, and invokes it if it is there.

As we build these Stats objects, we append their name-strings to a global list, and then use this when the time comes to print out statistics values—typically at the end of a batch, or the end of the entire simulation.

We of course write out the results using Tcl: each object provides a dot function (.report) that reports its values to a reporter object, which in turn writes the results (in the form of Tcl code) to a file. We call this from inside a Tcl procedure that decides which values are interesting at what stages of the simulation, and directs the results to a particular file. This lets us write different results to different output files easily, and will eventually allow us to direct results to other tools, such as a Tk-based monitor.

3.5 Complex-object initialization

The original code from which we derived several portions of our simulator was written by Chris Ruemmler [Ruemmler93]. One of the things it needed to do was to provide descriptions of the nitty-gritty details of a wide range of disk types (including seek distance profiles, capacity, tracks-per-zone, buffer-cache management policies, and so on). The original approach to this was to build an array of "info" structures, initialized by a C++ file. Besides being error-prone, this was rather tedious to change—especially by the time it took a few minutes to relink the simulator.

We replaced this by Tcl scripts to build a set of info objects, and insert these into a Tcl array, indexed by disk type. This both allowed us much greater control, and simplified the code. Even more importantly, a change meant we now only had to go around the few-second rerun loop rather than the several-minute recompile-relink-rerun loop.

3.6 Overall control

We found it convenient to divide the configuration functions into a number of phases, each represented by a Tcl script:

- *host*: this describes the original system that we are modelling
- *back-end parameterization*: sets default values of parameters to use unless overridden for this particular experiment
- *back-end*: constructs the simulation objects
- *workload*: makes C++ objects to read requests from a trace file or synthesize a stream of requests
- *batch execution*: the Tcl script that actually runs the simulator, deciding whether the results have converged or not, and deciding how large a batch to run next

This has one significant advantage: we can inject additional Tcl scripts between the phases to "post-modify" a standard setup. For example, we can add a script to limit all the host I/O requests to a single bus, regardless of the original host configuration.

To help with default parameter management, we found it useful to write a trivial Tcl procedure called sset, which does nothing if the target variable exists, or otherwise acts like set. The parameter files contain "sset variable value" lines, allowing them to be overridden by explicit set commands passed in on the command line.

3.7 Result analysis/reporting/graphing

Our early .report functions emitted data in a form convenient for post-processing by awk, but we soon found ourselves writing data-analysis scripts in Tcl instead. One problem remained: the cost of scanning large amounts of data is quite high (a typical simulation run emits about 15MB of results). Much time was being spent in converting data back and forth between the emitted format and values in Tcl lists.

We've since moved to emitting the results as Tcl scripts that can be sourced to reload the data we really want. This gives us much more control over what we load (for example, we don't attempt to load all 15MB per run for inter-run analyses), and gives the Tcl data-analysis functions much more immediate gratification when they need access to a value to plot.

We have also built scripts that create plots using the jgraph program. These scripts build an index of the result files, then allow one to create graphs in a declarative style by specifying only the names of the simulator runs to be considered, the names of the variables to be graphed, and the kind of graph desired for each.

4 Lessons we've learned

- Tcl is way cool.
- More and more things have migrated into Tcl: we've never moved anything back for performance reasons.
- $object and dot functions work very well.
- Type checking saved our bacon a few times.
- The macros we use to control the size of the C-to-Tcl interface are themselves rather daunting.
- We've had a few difficulties with the Tcl syntax, the largest being the need to escape newlines inside [square brackets]. Why can't they be treated like {curly braces}?

What we'd like different in Tcl:

- A single, simple way to ask the Tcl interpreter "what kind of object is this thing" (array? list? value?).
- An execution-trace facility like the regular UNIX system shells' -v option.
- A way to do C-style comments (partly because of the need to escape newlines inside square brackets).

5 Futures

- Develop a Tk interface that displays the simulated-system structure, allows statistics objects to be added while a simulation is running, displays "live" status updates and progress monitors, …

- Make the simulator available to other researchers.

References

[Cao93] Pei Cao, Swee Boon Lim, Shivakumar Venkataraman, and John Wilkes. The TickerTAIP parallel RAID architecture. *Proceedings of 20th International Symposium on Computer Architecture* (San Diego, CA), pages 52–63, 16–19 May 1993.

[Goldberg83] Adele Goldberg and David Robson. *Smalltalk–80: the language and its implementation*. Addison-Wesley, Reading, Mass, May 1983.

[Ousterhout94] John K. Ousterhout. *Tcl and the Tk toolkit*, Professional Computing series. Addison-Wesley, Reading, Mass. and London, April 1994.

[Ruemmler93] Chris Ruemmler and John Wilkes. UNIX disk access patterns. *Proceedings of Winter 1993 USENIX* (San Diego, CA, 25–29 January 1993), pages 405–20, January 1993.

[Ruemmler94] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, **27**(3):17–28, March 1994.