Tcl and Java Performance

by
H. John Reekie, University of California at Berkeley
Christopher Hylands, University of California at Berkeley
Edward A. Lee, University of California at Berkeley

Abstract

Combining scripting languages such as Tcl with lower−level programming languages such as Java offers new opportunities for flexible and rapid software development. In this paper, we benchmark various combinations of Tcl and Java against the two languages alone. We also provide some comparisons with JavaScript. Performance can vary by well over two orders of magnitude. We also uncovered some interesting threading issues that affect performance on the Solaris platform.

"There are lies, damn lies and statistics"

This paper is a work in progress, we used the information here to give our group some generalizations on the performance tradeoffs between various scripting languages. Updating the timing results to include JDK1.2 with a Just In Time (JIT) compiler would be useful.

Introduction

There is a growing trend towards integration of multiple languages through *scripting*. In a famously controversial white paper (Ousterhout 97), John Ousterhout, now of Scriptics Corporation, argues that scripting −− the use of a high−level, untyped, interpreted language to "glue" together components written in a lower−level language −− provides greater reuse benefits that other reuse technologies. Although traditionally a language such as C or C++ has been the lower−level language, more recent efforts have focused on using Java. Recently, Sun Microsystems laboratories announced two products aimed at fulfilling this goal with the Tcl and Java programming languages. The goals of these products are to "create a synergy between Java and Tcl" (Johnson 98).

The first product, Tcl Blend, is a C interface between the Tcl interpreter and the Java virtual machine (JVM). The Tcl programmer is provided with a few commands that create and manipulate Java objects. Tcl Blend originated in a simpler interface named TclJava which was created by Scott Stanton and Ken Corey (Stanton and Corey 96) of Sun Laboratories. The second, Jacl (Lam 97), is an implementation of the Tcl interpreter written entirely in Java. This project originated as a research project developed by Ioi Lam and Brian Smith of Cornell University.

The combination of scripting and Java is taking place on other fronts. Users of the Python language, for example, are looking at integrating Python and Java, with very similar goals to those of the Sun team (Cunningham *et al* 97 , Hugunin 97). The creator of Perl, Larry Wall, has written a Java−Perl interface for O'Reilly Software (JPL 98). JavaScript, although primarily a language for scripting Web Browsers, nonetheless provides a seamless interface to Java code running inside the same browser.

In this paper, we examine the relative performance of Tcl, Java, Tcl Blend, and Jacl. We also take a brief look at JavaScript and identify some problems with benchmarking it. Our test suites are extended versions of those developed by Kernighan and Van Wyk (Kernighan and Van Wyk 97) in a paper comparing a number of scripting and user interface languages.

The graphs in this paper are generated dynamically (from pre−generated data) by a Java applet called **ptplot**, which allows zooming in on the graph display. If you are reading this paper on paper, we highly recommend that you also take a look at the online version at `http://ptolemy.eecs.berkeley.edu/papers/98/scriptperf`. (The data can be zoomed by dragging a rectangle from a point on the plot down towards the lower right.)

The languages

Java hardly needs an introduction. It was developed at Sun Microsystems, and has been promoted a "better C++." Java is byte−compiled, although just−in−time compilers to native code can improve performance substantially

(Kernighan and Van Wyk).

Tcl is an interpreted language that was originally developed by John Ousterhout, then of UC Berkeley. It is presently being developed by Sun Laboratories. Tcl was originally fully−interpreted, but the most recent release (8.0) includes a byte−compiler (Lewis 96).

Tcl Blend and Jacl aim to provide a seamless interface between Tcl code and Java. For example, the following piece of Tcl code creates a Java object and then calls two methods of that object:

```
set obj [java::new StringBuffer "I am a... "]
$obj append "StringBuffer!"
puts [$obj toString]
```

JavaScript was created by Netscape for dynamic control over HTML pages −− the JavaScript code is embedded in the HTML source, and executes within the browser while it displays the page. JavaScript can also be used in server−side scripts. JavaScript looks very much like Java, so porting the tests from Java to JavaScript was fairly straight−forward.

Despite its name, JavaScript is unrelated to Java, but it is able to access Java classes and methods via Netscape's LiveConnect software. The following piece of JavaScript code creates and accesses a Java object:

```
var myString = new java.lang.String("  a trimmed string  ");
document.write(" −−"+myString.trim()+"−−");
```

Methodology

Kernighan and Van Wyk provide their test sources on Kernighan's web site. We downloaded these tests and created a makefile that compiles the Java files, and a driver script named `timing.sh` that runs the tests using various languages. To test Tcl Blend, we ran the Java versions of the tests, which use the Java `Date` class and reports back the elapsed time in milliseconds. To test Jacl, we ran the Tcl versions of the tests, which use **gtime** to report back the elapsed time in milliseconds. We also ported some of the tests to JavaScript, but we were not able to get consistent results for all of the tests under JavaScript.

The driver script follows Kernighan and Van Wyk's methodology fairly closely. Where possible, each test is run 10 times and the times are reported. With some of the languages and tests, we found that we could not run the test 10 times. We note these cases below.

We ran the tests on a Sun 200Mhz Ultrasparc with 128Mb of memory running Solaris 2.5.1. The complete details of how we generated the test bed, including the sources for our additional test, are given in the *Scripting Performance Test Bed Details*Appendix. We did not run the tests on any other platforms.

Java

To provide a baseline measurement, we ran Kernighan and Van Wyk's Java version of the tests using Sun's JDK1.1.5. The source code was unmodified, and we reported the times computed by the internal Java timer. We compiled the Java classes with optimization turned on.

Kernighan and Van Wyk chose not to include the start−up time of the Java run−time environment in their Java times. We are not sure if this is the right approach, since startup time can play a critical role in determining the overall usability of small scripts. To take an example from our own experience, we ported a plotting program called **xgraph** from C and X11 libraries to Java. Although this gives this program a new lease of life (including its use to display the graphs in the on−line version of this paper), the startup time ballooned from barely−noticeable for the C version, to several seconds for the Java version.

Tcl

For another baseline measurement, we ran Kernighan and Van Wyk's Tcl versions of the tests, using both Tcl7.6 (the version they used) and the newer Tcl8.0p2. The main difference between the two versions is that Tcl8.0 includes a byte compiler (Lewis 96).

We considered using Tcl's internal **time** command, which returns elapsed time in microseconds, instead of **gtime**.

We decided against it, in order to produce results consistent with Kernighan and Van Wyk. In the future, it might be worthwhile to modify the Tcl tests to use the internal command.

Tcl Blend

To test the performance of Tcl Blend, called Kernighan and Van Wyk's Java versions of the tests from a Java−enabled Tcl interpreter. Here is an example, in which the sumloop test is run (`jtclsh` is part of a patch to Tcl Blend created by one of the authors, which sets the CLASSPATH appropriately and calls `java`):

```
carson 3% setenv CLASSPATH .
carson 4% jtclsh
jtclsh% set n [java::new sumloop]
java0x1
jtclsh% set v [java::new {String[]} {1} {125000}]
java0x2
jtclsh% $n runsumloop $v
125000 53 msec
125000 53 msec
jtclsh%
```

The Tcl Blend C code was compiled with gcc−2.8.0 and `-O2`optimization. The Tcl Blend Java classes were compiled with jdk1.1.5 and `-O` optimization. To time Tcl Blend, we followed Kernighan and Van Wyk's methodology and used the internal timer in the Java tests.

Jacl

Jacl is a 100% Java implementation of the majority of Tcl8, so we ran Kernighan and Van Wyk's Tcl versions of the tests by using a script that sets the CLASSPATH properly and starts up the `java` runtime interpreter.

To time Jacl, we followed Kernighan and Van Wyk's methodology for timing Tcl and used the **gtime** command to time execution of the entire Jacl process, including the Java startup time. As a point of reference, on our test machine, starting up Jacl and then immediately exiting took about 0.75 seconds.

Jacl is much slower than the other languages. In some cases, tests that took under 10 seconds in Java took over 30 minutes in Jacl. We therefore ran each test four times for Jacl, instead of 10 times as we did for the other languages.

Note that since Jacl is a Java implementation of Tcl8, we are really interested in comparing Jacl performance with that of raw Tcl8. When comparing Jacl and Tcl8 with Java, we must take into account the fact that Jacl and Tcl8 times include the startup time. In general, we found that Jacl was so slow that the extra 0.75 seconds made little difference in the overall results.

JavaScript

To time JavaScript, we ported Kernighan and Van Wyk's Java tests to JavaScript. (The code is in `driver/js/timing.js`.) For the test we ran Javascript on (sumloop, ack, array1, string, assoc), we started a new Netscape 4.04 process and visited the appropriate web page from `driver/js/index.html`. We then copied the output with the mouse and pasted it into a file. We did not port the cat, wc, tail, and suml tests to JavaScript because these tests involve large files, so we would really be testing the speed of the network.

We found that there was wild variation in time that the same test took. Since one Netscape process runs all the individual tests in a class of tests, memory allocation could be skewing the timing of the later individual tests.

Basic Features

sumloop

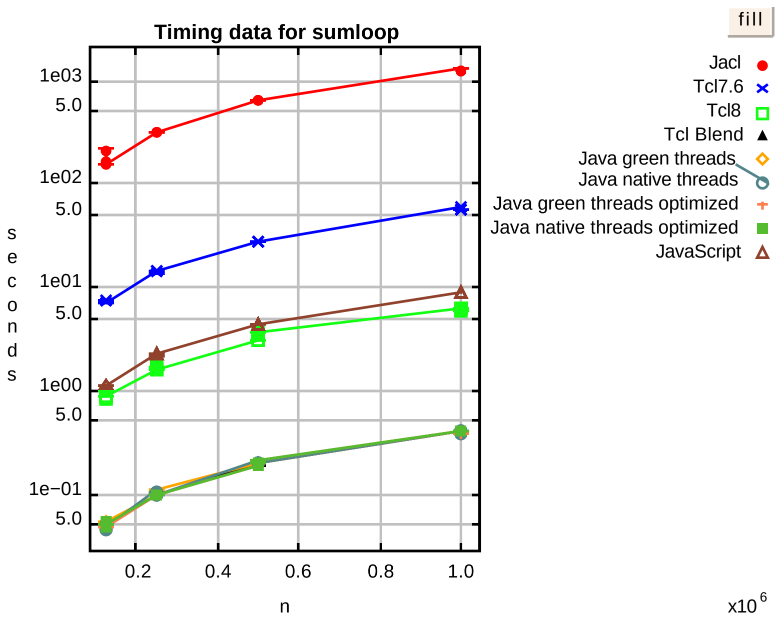
The sumloop test consists of a simple loop that increments an integer and adds it to an accumulator. The Java implementation written by Kernighan and Van Wyk looks like this:

```
sum = 0;
```

```
for (i = 0; i  n; i++)
    sum++;
```

n is set to 125000, 250000, 500000, and 1000000 for successive tests.

As one would expect, execution time increased linearly with *n*. Jacl was by far the slowest performer, being about two orders of magnitude slower than the byte−coded Tcl interpreter, and almost three orders of magnitude slower than Java. As we expected, the Java and Tcl Blend lines coincide almost exactly for this test.



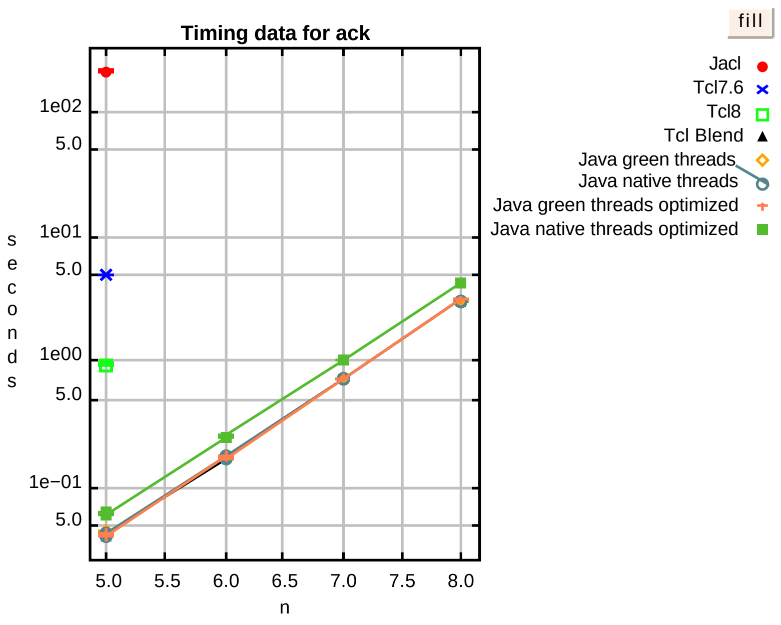
ack

This tests compute Ackermann's function, which is an excellent test for recursive function calls. According to Kernighan and Van Wyk, ack(3,k) requires at least 4^(k+1) function calls and reaches a recursive depth of 2^(k+3)−1.

Here is the Tcl implementation of the Ackermann function:

```
proc ack {m n} {
    if {$m == 0} { return [expr $n+1] }
    if {$n == 0} { return [ack [expr $m−1] 1] }
    return [ack [expr $m−1] [ack $m [expr $n−1]]]
}
```

We attempted to run this test for ack(3,5), ack(3,6), ack(3,7), ack(3,8). Tcl7.6, Tcl8.0 and Jacl ran out of stack space for ack(3,6) and greater. (Kernighan and Van Wyk note this problem in their paper.) The JavaScript version, running in Netscape4.04 under Solaris, causes Netscape to segfault while computing ack(3,5). The obvious conclusion is to not use these languages for intensely−recursive functions.



Arrays and strings

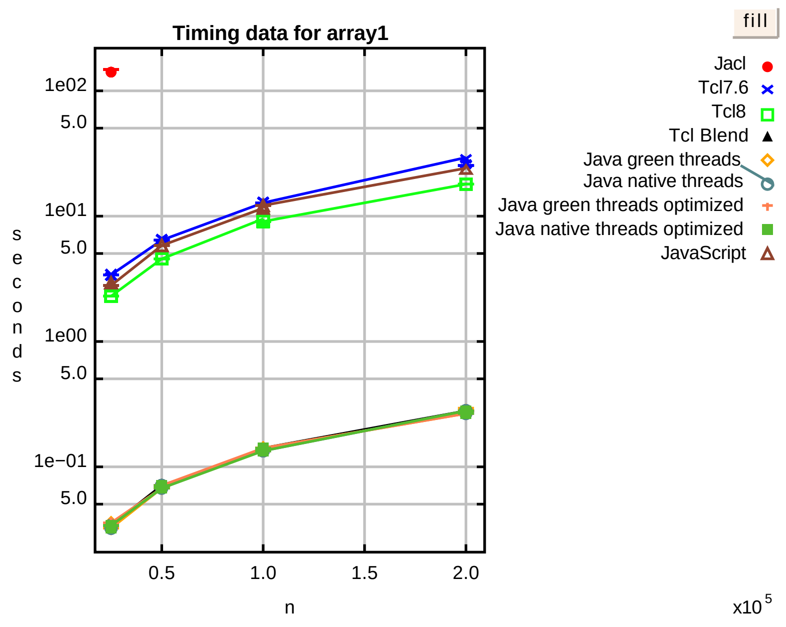
array1

This test sets elements of an array to integer values, and them copies the contents to another array. The JavaScript implementation looks like this:

```
for (var i = 0; i  n; i++)
    ip[i] = i ;
for (var j = n−1; j >= 0; j−−)
    jp[j] = ip[j];
```

We ran this test with *n* to 25000, 50000, 100000 and 200000. Jacl threw an out−of−memory exception with *n* set to 50000. JavaScript successfully ran the test twice, but would cause the operating system to consume all available memory if the test was run any more times. As a result, we took only two measurements for JavaScript.

Other than its memory problems, JavaScript surprised us in this test by turning in a relatively good performance. Nonetheless, all the scripting languages were an order of magnitude slower than Java.



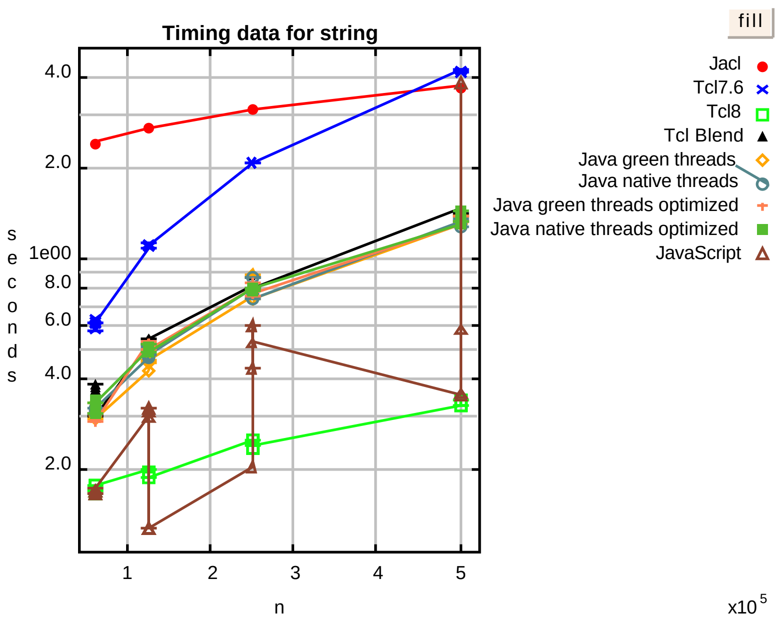
string

This test performs various operations on a string, such as getting a range of characters, taking the length, and concatenating strings. Kernighan and Van Wyk's Tcl implementation looks like this:

```
for {set j 0} {$j  10} {incr j} {
    set s "abcdef"
    while {[string length $s] = $n} {
        set s "123${s}456${s}789"
        set s "[string range $s [expr [string length $s]/2] end] \
               [string range $s 0 [expr [string length $s]/2]]"
    }
}
```

We ran this test with *n* equal to 62500, 125000, 250000 and 500000. The JavaScript version of the test consumed all system memory (Netscape 4.04 under Solaris).

As noted by Kernighan and Van Wyk, the performance of the memory allocators plays a large factor in this test. Notable is the fact that the byte−compiled Tcl code is consistently several times faster than Java, showing the advantage of languages with C−code native functions. Another interesting point was that even Jacl was sometimes faster than the fully−interpreted Tcl version. The JavaScript data is all over the place, because JavaScript was thrashing the system's memory allocator.



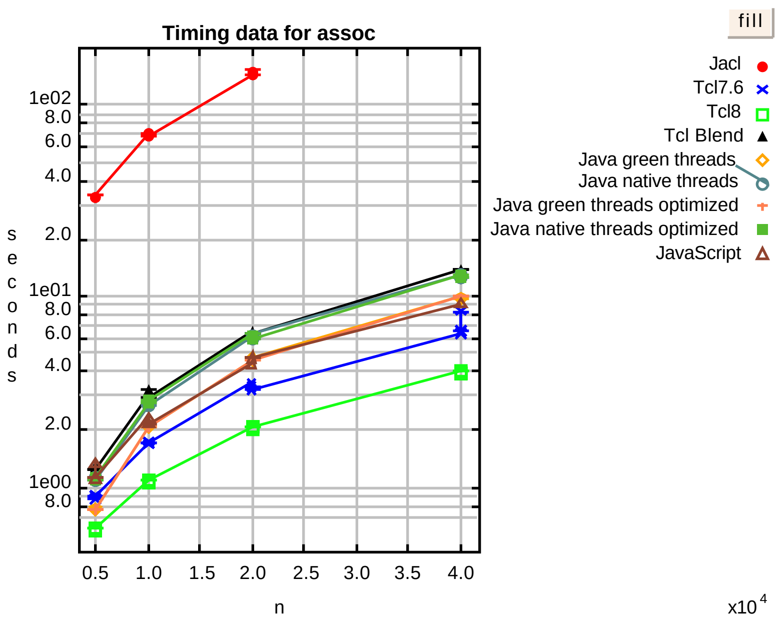
assoc

This test performs lookup and setting operations on associative arrays. The Java tests use the Java HashTable class, while the other languages have associative arrays built into the language.

Kernighan and Van Wyk's Java implementation looks like this:

```
Hashtable ht = new Hashtable();
...
c = 0;
for( i = 1; i = n; i++)
    ht.put(Integer.toString(i, 16), new Integer(i));
    for (i = 1; i = n; i++)
        if (ht.containsKey(i+""))
            c++;
```

All of the languages we tested, except Jacl, were quite close in this test. Jacl threw out−of−memory exception for *n* set to 4000. The byte−compiled Tcl test show some odd variation in times that we are unable to explain.



Input−Output

The input/output tests produced a number of surprising results. First, the byte−coded Tcl interpreter often performed worse then the naive (7.6) Tcl interpreter. In addition, both of these were often faster than Java, presumably because their I/O is hand−coded in C, whereas the Java I/O is coded in Java (and quite complex as well).

A third result that we were not at all prepared for is the performance difference between Tcl Blend and Java. In most of the tests, this comes in at about a factor of two. We were mystified by this at first −− surely it should be the same, as for all the other tests? After some experimentation (read "desperate hacking"), we discovered that the performance difference is due almost entirely to the threading package used by the Java virtual machine: the native threads package on Solaris cause I/O to run at half the speed achieved with the Green threads package.

We confirmed this fact with the `tail` test for several different combinations. In the following, the JNI test is a simple C program that calls Java directly through the Java Native Interface:

Java, green threads	16.9
Java, native threads	31.6
Tcl Blend, native threads	36.3
JNI, native threads	36.1
JNI, green threads	17.3

This data presents something of a quandary for developers, as well as benchmarkers! Native threads should be more robust, and only native threads will properly use a multi−processor machine (we have a few in our research group). On the other hand, native threads present a serious performance obstacle. In the case of Tcl Blend, we have to use native threads, since Tcl Blend does not work with Green threads.

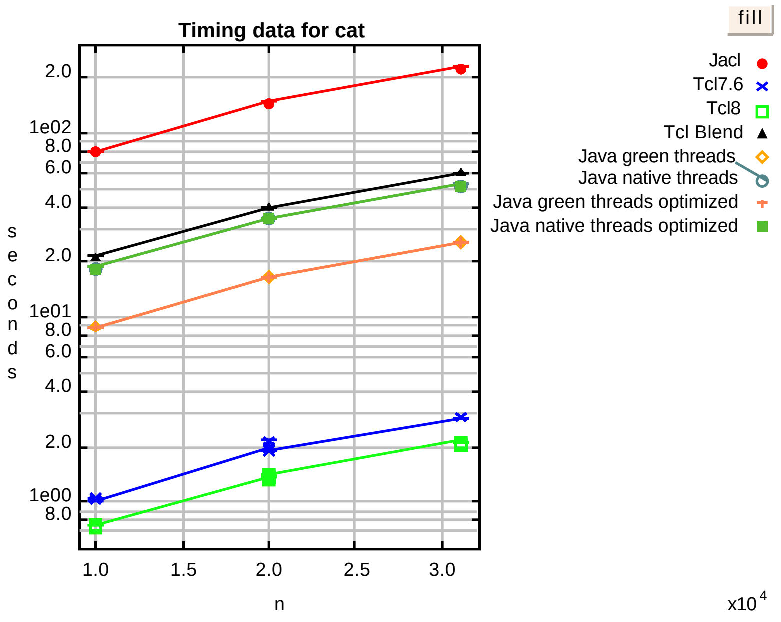
cat

This test reads in a file and then writes it out to another file. The code is straight−forward, except for the case of Jacl. The Tcl version of the test uses the Tcl `fconfigure` command, which is not yet implemented in Jacl, so we wrapped the `fconfigure`call in a `catch`. Kernighan and Van Wyk's Tcl implementation of the `cat` test with our minor change looks like this:

```
catch {fconfigure stdout -buffering full}    ;# not needed on unix
```

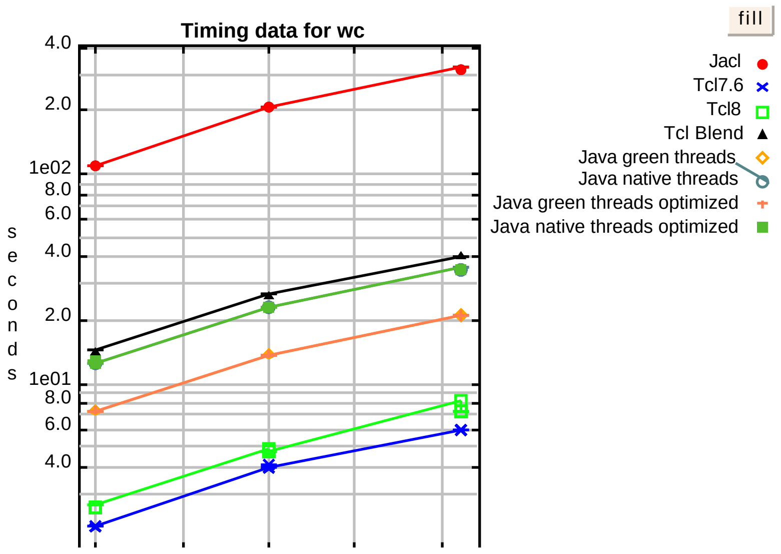
```
foreach f $argv {
    set fd [open $f]
    while {[gets $fd line] >= 0} {
        puts $line
    }
    close $fd
}
```

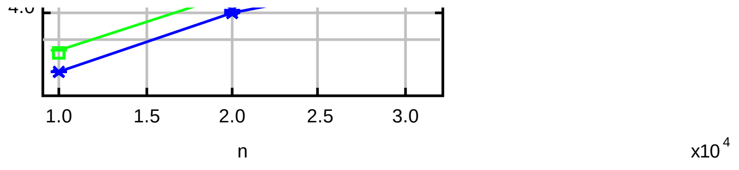
As for the string test, Tcl is faster than Java, because its I/O libraries are coded in C, whereas Java's are coded in Java. The factor−of−two performance hit of using native threads in Tcl Blend is apparent in this and the following two plots.



wc

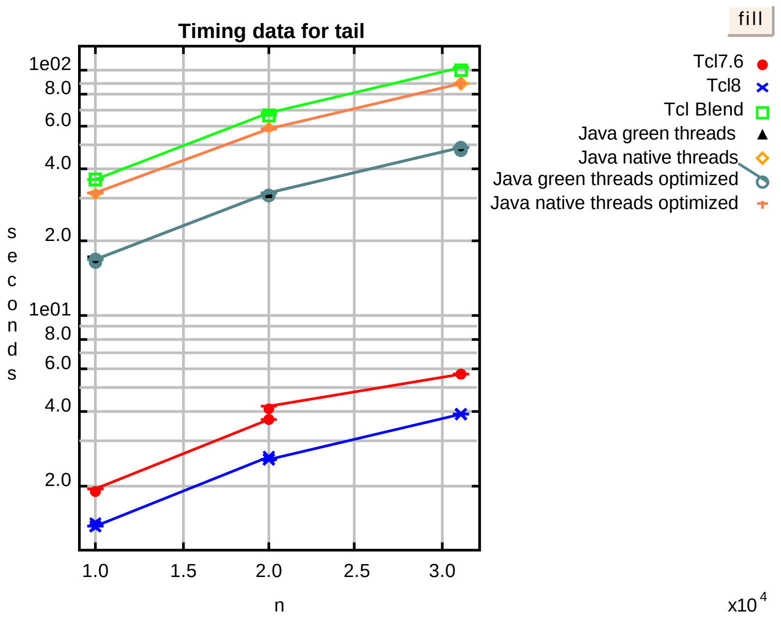
This test counts the words in a file. The Tcl version of the test reads a line at a time and simply counts the number of words in each line using the Tcl `llength` function. The Java version scans the file a character at a time looking for spaces. As for the previous test, the pure Tcl solutions are substantially faster than the Java solutions.





tail

This test reads the entire file into an array and prints the lines out in reverse order. Again, the pure Tcl solutions were substantially faster than the Java solutions. Jacl did not produce any results, as it ran out of memory after about 400 seconds on a 10000 line test file.

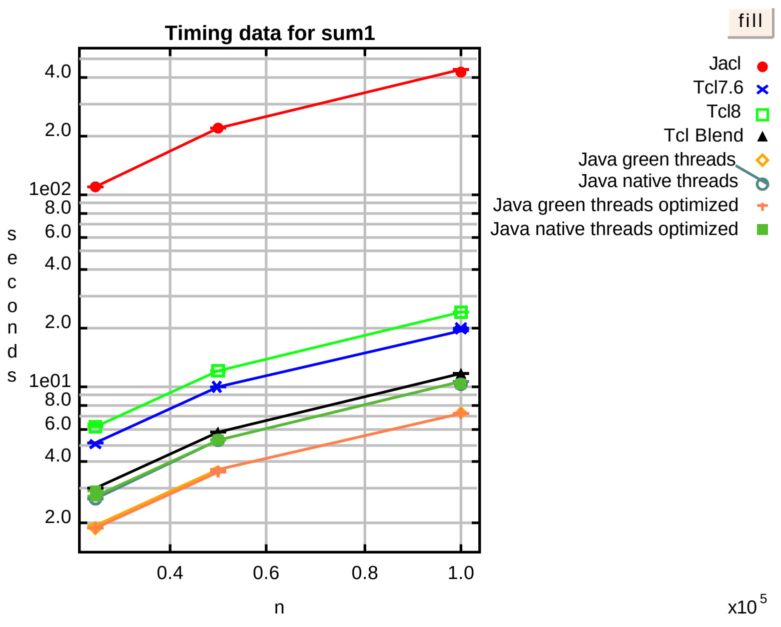


sum1

This test reads a file, and produces the sum of the first field on each line. It is a combination of input and numeric processing.

Kernighan and Van Wyk's Tcl code is below:

```
set s 0 set tcl_precision 17
foreach f $argv {
    set fd [open $f]
    while {[gets $fd line] >= 0} {
        set s [expr $s + [lindex $line 0]]
    }
    close $fd
}
puts $s
```



Discussion and Conclusions

The tests contained a lot of results that we expected, either from the tests performed by Kernighan and Van Wyk, or from our own intuitions about the relative performance the various implementations. They also contained many we didn't expect!

For numeric and control processing, Java is fastest. However, string processing and file I/O is better done in Tcl. The use of Tcl Blend to call Java had no measurable effect in performance −− except for the factor−of−two drop present with native threads on Solaris!

The Tcl byte−code compiler usually, but not always, improves performance. In particular, tests involving I/O tend to run slower in the byte−code interpreter. Jacl, not surprisingly, is always slowest. We expected this, but were unprepared for the degree in difference for some of the tests (two orders of magnitude in some cases).

To be fair, the Jacl release we tested was the first release, and Jacl is a much newer program than Tcl. To get a better handle on a real life Jacl application, we ran both Jacl and Tcl Blend on the Jacl test suite −− in this context, Jacl was 6 to 7 times slower than Tcl Blend. There were, however, a number of discrepancies in the test results, most common being that Tcl Blend failed some tests because of minor differences in return values. We noted that the test suite calls a lot of Java code, where Jacl and Tcl Blend will run at similar speeds −− it could be that the test suite is not executing much Tcl code, hiding some of Jacl's poor interpretation performance.

Still, the Jacl test suite results are heartening, since it seems that overall Jacl is not as slow as some of the timing tests indicate. Even so, its overall performance is poor enough that we can only consider it to be an interesting experiment at the moment.

The JavaScript tests are unlike the other tests, since JavaScript runs inside a Web browser. The JavaScript times are probably higher than they would be if it were possible to run them as a stand−alone program. Comparisons of the JavaScript times to the other times should be taken with a very large grain of salt. It would be interesting to compare performance of JavaScript and Java running in the same browser −− some preliminary measurements indicate that Java code running in a browser runs at about half the speed as when stand−alone.

In summary, combining scripting languages with Java offers new opportunities for software development, but with some interesting and non−obvious performance implications.

References

Papers

(Cunningham 97)
: Douglas Cunningham, Eswaran Subrahmanian, Arthur Westerberg, *User−Centered Evolutionary Software Development Using Python and Java*, Proceedings of the 6th International Python Conference, http://www.python.org/workshops/1997−10/proceedings/cunningham.html.

(Hugunin 97)
: Jim Hugunin, Eswaran Subrahmanian, Arthur Westerberg, *Python and Java − The Best of Both Worlds*, Proceedings of the 6th International Python Conference, http://www.python.org/workshops/1997−10/proceedings/hugunin.ps.

(Johnson 98)
: Raymond Johnson, "*Tcl and Java Integration*," White paper, Sun Microsystems Laboratories, http://sunscript.sun.com/java/tcljava.ps.

(Kernighan and Van Wyk 97)
: Brian W. Kernighan and Christopher J. Van Wyk, "*Timing Trials, or, the Trials of Timing: Experiments with Scripting and User−Interface Languages*," Bell Labs, Murray Hill NJ, http://cm.bell−labs.com/cm/cs/who/bwk/interps/pap.html

(Lam and Smith 97)
: Ioi K. Lam, Brian Smith, "*Jacl: A Tcl Implementation in Java*," Proc. of the USENIX 1996 Tcl/Tk Workshop, Boston, MA July 1997.

(Lewis 96)
: Brian Lewis, "*An On−the−fly Bytecode Compiler for Tcl*," Proc. of the USENIX 1996 Tcl/Tk Workshop, Monterey, CA, July 1996,

(JPL 98)
: O'Reilly Software, *JPL − A Java−Perl Interface*, http://www.oreilly.com/catalog/prkunix/info/more_jpl.html

(Ousterhout 97)
: John K Ousterhout, *Scripting: Higher Level Programming for the 21st Century*, White paper, Scriptics Corporation, http://www.scriptics.com/people/john.ousterhout/scripting.html

(Stanton and Corey 96)
: Scott Stanton, Ken Corey, "*TclJava: Toward Portable Extensions*," Tcl/Tk conference, Monterey, CA, 1996.

Websites

Scriptics Jacl and Tcl Blend page
: http://www.scriptics.com

Netscape's JavaScript documentation
: http://developer.netscape.com/library/documentation/javascript.html

Brian Kernighan's web site − includes links to the code for the tests.
: http://cm.bell−labs.com/cm/cs/who/bwk/

Scripting Performance Test Bed Details
: http://ptolemy.eecs.berkeley.edu/~cxh/java/tclblend/scriptperf/driver/index.html

Christopher Hylands' Tcl Blend page.
: Includes patches and sources for this paper.
http://ptolemy.eecs.berkeley.edu/~cxh/java/tclblend/index.html

---

Back to Tcl Blend

Last updated: 09/01/98