Experiences with modularizing the Tcl core.

Andreas Kupries ActiveState Corp 580 Granville Vancouver, BC CA
andreask@ActiveState.com

ABSTRACT

As part of a contract with Cisco for their migration of the Tcl interpreter embedded in IOS, their router OS, from version 7.6 to version 8.3.4, ActiveState was tasked with reducing the static and stack memory requirements of the Tcl core. To this end, it modularized the 8.3.4 Tcl core so that features not required by Cisco could be compiled out of the core. This reduced the static size of the core, and reduced the load on the C stack by moving big structures to heap. Here we describe our experiences with performing these tasks, the techniques we used, the difficulties we had, and our conclusions regarding the future work on these tasks.

1. OVERVIEW

ActiveState was given the task of reducing both the static and stack memory requirements of the Tcl core. A previous attempt by Karl Lehenbauer to fit Tcl into smaller machines [1]A used an older and smaller versions of Tcl, version 6.7 as its base. In contrast to this our customer explicitly requested the usage of the 8.3.4 core, as part of an upgrade of his systems from the 7.x series to 8.3.4.

It is not known which version of Tcl was the base for the even earlier attempt of using Tcl in the Mars Pathfinder [2]A, except that it definitely was a version released before the introduction of bytecode compilation. Their approach was even more radical than ours, by reducing the interpreter to an absolutely minimal core of parser and basic supporting facilities (tclParse.c, tclBasic.c), and then adding facilities as required by their scripts.

Another project of interest is *Rivendell* by John Hall, sometimes called `PalmTcl` [3]A. This project is based upon Tcl 7.4 and ports the language to PalmOS-based PDA's. The changes to the base 7.4 core `ANSI`fy the code base, changed the memory allocation, and split the code into sections, essentially a number of interdependent dynamic libraries, to deal with limits placed upon the size of such libraries (They have to be smaller than 64 K). Although the description claims that the code *has been enhanced with reduced stack usage* no such changes were found.

Similar to Rivendell is Ashok Nadkarni's Palm Tcl, again a port of Tcl to the Palm OS, based on Tcl 7.6 [4]A. This project was not investigated.

We on the other hand have now modularized the Tcl core (version 8.3.4) so that the features not required by our customer can be compiled out of the core, at the discretion of the user of the modified core.

We also reduced the load on the C stack through shrinking big structures on the stack, or moving them to the heap wholesale where shrinking was not possible. In this context it is interesting to know that Karl Lehenbauer noted in his paper [1]A the stack requirements of the Tcl interpreter as the major problem he encountered when doing the port.

The remainder of the paper is structured as follows. In chapter 2 we explain which features were selected for modularization and how the excisable code was determined. In the next chapter, 3 we present the results of the modularization and compare the static size of the core with various features removed.

Chapter 4 discusses the second major topic, the reduction of the usage of the C stack by the Tcl interpreter to allow its use in environments with limited stack space. The results of our changes are presented chapter 5.

At last, chapter 6 discusses our conclusions, and chapter 7 possible future work in this area.

2. FEATURES OF THE CORE

Our first task was to reduce the overall size of a statically build interpreter executable (`tclsh`) and/or interpreter library (`libtcl.a`). This was realized by making a number of features provided by the standard interpreter optional, i.e removable at will during compile time. With this an end-user of the modularized core is free to remove any features which are not needed by his application and thus to shrink the space required for the executable or library.

The overall set of features which were initially made removable were determined by the requirements of our customer. In other words, we did not spent any time on features possibly giving us a high gain for the effort, but only on features our customer deemed irrelevant for his environment. The details however, i.e. the division of the features named by our customer into the actual removable features, were guided by the results of the discussions the community had had on the newsgroup `comp.lang.tcl`about this topic [5]A.

The chosen features are listed in Table 1, together with

Tcl '2002 Vancouver, BC CA
.

the C preprocessor macros whose existence will activate the removal of a particular feature. By default none of these macros are defined. Setting the macro `MODULAR_TCL`will activate all of these macros and thus remove the whole set of features.

As Source Navigator crashed when processing the Tcl C sources at that time we decided to conditionalize the code by hand with support by the C compiler. This means that it is possible that we missed code which could have been made optional but currently is not. This may happen especially for a function F which is shared by removable features, and only such. In that case removing all features requiring F will **not** remove F. This deficiency has to be addressed in the future.

Starting at the topmost function for a feature, usually the function implementing the command to remove we followed the sources and determined the public and non-public non-static functions reachable from this point. After excluding them from the sources through ifdef's a complete build was performed and the compiler log analyzed to find all functions which are actually required by other parts of the core. These functions were made unconditional again. Additional runs of the compiler then allowed us to find all static functions required only by the excluded functionality. These were excluded as well.

3. COMPARISON OF CORE SIZES

Compiling the modified Tcl core for a number of different flags combinations yielded Table 2. It shows us how much of the object code in a static interpreter is used by the various features.

It is obvious from looking at this table that most of the chosen features reduce the overall size of the static interpreter by only very small amounts of space. Only the file system code and the code dealing with the management of channels beyond the standard channels reduce the size by moderately significant amounts of space.

There is obviously room for improvement here. The question is, where are the features whose removal will reduce the library by significant amounts of space ? Our best answer so far is Table 7. This table lists the 36 largest object files and thus points us to the main areas we should look into.

Based on that data several possibilities for additional features to be removed are discussed in more detail later, in chapter 7, about our future work.

4. REDUCTION OF STACK USAGE

The second task was to reduce the load placed on the C stack by the interpreter to enable the interpreter to run in environments with limited stack space. In the case of our customer this would be a router.

To this end three different techniques were used:

1. Identification of variable sized data types with a large default size, instances of which are placed on the stack.

   For such types we reduced their default size either through an existing preprocessor macro determining their size, or by introducing such a macro and proceeding as before.

   An example of such a type is `Tcl_DString`.Its base size is $3 * sizeof(long)$ on typical 32bit systems, i.e. 12 bytes. However it also allocates a static character buffer as part of the structure to handle small strings without having to allocate space on the heap. The macro controlling the size of this buffer is `TCL_DSTRING_STATIC_SIZE`.Its default value is 200, driving the size of the whole structure to 212 bytes.

   Table 3 shows the macros introduced in this way.

2. Identification of large variables on the stack which cannot be shrunk. For these we devised a set of macros for the declaration and use of such variables whose implementation can be switched between the allocation of such variables on the stack and their allocation on the heap.

   An example of the usage of these macros can be found in the file `generic/tclCompCmds.c`,see the function `TclCompileIncrCmd`:

```
int
TclCompileIncrCmd(interp, parsePtr, envPtr)
    Tcl_Interp *interp;
    Tcl_Parse *parsePtr;
    CompileEnv *envPtr;
{
    Tcl_Token *varTokenPtr, *incrTokenPtr;
    TEMP (Tcl_Parse) elemParse;
    ...
    STRING (160, buffer);

    NEWTEMP (Tcl_Parse,elemParse);
    NEWSTR (160, buffer);

    envPtr->maxStackDepth = 0;
    if ((parsePtr->numWords != 2) &&
(parsePtr->numWords != 3)) {
        ...
        RELTEMP(elemParse);
        return TCL_ERROR;
    }
    ...
}
```

   Figure 4 shows side by side the two implementations for allocating temporary variables on either the stack or the heap. If the macro `TCL_STRUCT_ON_HEAP`is defined during compilation temporary variables will be allocated on the heap.

3. Miguel Sofer[1] provided us with a modified engine for the execution of bytecode. This engine does **not** invoke itself recursively when calling a byte-compiled procedure from other byte-compiled code. It rather jumps directly back into its own main loop, reusing the state variables on the C stack. The actual state of the engine is saved to and restored from the Tcl evaluation stack, which is managed on the heap. This means that calling Tcl procedures inside of Tcl procedures, and especially recursion of Tcl procedures does not consume any C stack anymore.

   This new executor was named **NRE**, for “non-recursive engine”.

[1]Mail: <mig@utdt.edu>

Table 1: **Removable features**

Macro	Removed feature
`TCL_NO_SOCKETS`	Channel driver "tcp"
`TCL_NO_TTY`	Channel driver "tty"
`TCL_NO_PIPES`	Channel driver "pipe"
`TCL_NO_PIDCMD`	Command `[pid]`
`TCL_NO_NONSTDCHAN`	Creation of channels beyond stdin, stdout and stderr
`TCL_NO_CHANNELCOPY`	Channel copying, C/Tcl, `[fcopy]`
`TCL_NO_CHANNEL_READ`	Command `[read]` and "Tcl_ReadChars"
`TCL_NO_CHANNEL_EOF`	Command `[eof]`
`TCL_NO_CHANNEL_CONFIG`	Command `[fconfigure]` and Tcl_GetChannelOption
`TCL_NO_CHANNEL_BLOCKED`	`[fblocked]`
`TCL_NO_FILEEVENTS`	`[fileevent]` and underlying APIs
`TCL_NO_FILESYSTEM`	Everything related to the file system
`TCL_NO_LOADCMD`	`[load]` and machinery below
`TCL_NO_SLAVEINTERP`	Slave interpreters
`TCL_NO_CMDALIASES`	Command aliases

Table 2: Size of `libtcl.a` for selected variants

Flags	lib/libtcl8.3.a Size, absolute	Size, relative	Shrinkage, relative
	490124	100.00	0.00
`TCL_NO_CHANNEL_EOF`	489964	99.97	0.03
`TCL_NO_CHANNEL_BLOCKED`	489940	99.96	0.04
`TCL_NO_PIDCMD`	489824	99.94	0.06
`TCL_NO_CHANNEL_READ`	489420	99.86	0.14
`TCL_NO_TTY`	488268	99.62	0.38
`TCL_NO_CHANNELCOPY`	488032	99.57	0.43
`TCL_NO_LOADCMD`	487264	99.42	0.58
`TCL_NO_CMDALIASES`	486576	99.28	0.72
`TCL_NO_SLAVEINTERP`	485080	98.97	1.03
`TCL_NO_CHANNEL_CONFIG`	484932	98.94	1.06
`TCL_NO_SOCKETS`	484428	98.84	1.16
`TCL_NO_FILEEVENTS`	484376	98.83	1.17
`TCL_NO_SLAVEINTERP` `TCL_NO_CMDALIASES`	480624	98.06	1.94
`TCL_NO_PIPES`	480136	97.96	2.04
`TCL_NO_NONSTDCHAN`	463544	94.58	5.42
`TCL_NO_FILESYSTEM`	440940	89.96	10.04
`MODULAR_TCL`	403228	82.27	17.73

Table 3: Adaptable default sizes

Macro	Default size	Meaning, Location of usage
`TCL_DSTRING_STATIC_SIZE`	200	Default static buffer in Tcl_DString
`TCL_FMT_STATIC_FLOATBUFFER_SZ`	320	`generic/tclCmdAH.c`,line 1978
`TCL_FMT_STATIC_VALIDATE_LIST`	16	`generic/tclScan.c`,line 266
`TCL_FOREACH_STATIC_ARGS`	9	`generic/tclCmdAH.c`,line 1735
`TCL_FOREACH_STATIC_LIST_SZ`	4	`generic/tclCmdAH.c`,line 1739
`TCL_FOREACH_STATIC_VARLIST_SZ`	5	`generic/tclCompCmds.c`,line 621
`TCL_RESULT_APPEND_STATIC_LIST_SZ`	16	`generic/tclResult.c`,line 458 `generic/tclStringObj.c`,line 1199
`TCL_MERGE_STATIC_LIST_SZ`	20	`generic/tclListObj.c`,line 1009 `generic/tclUtil.c`,line 831
`TCL_PROC_STATIC_CLOCALS`	20	`generic/tclExecute.c`,line 6272
`TCL_PROC_STATIC_ARGS`	20	`generic/tclProc.c`,line 751
`TCL_INVOKE_STATIC_ARGS`	20	`generic/tclBasic.c`,line 1782 `generic/tclBasic.c`,line 1857 `generic/tclBasic.c`,line 2951 `generic/tclParse.c`,line 1332
`TCL_EVAL_STATIC_VARCHARS`	30	`generic/tclParse.c`,line 1166
`TCL_STATS_COUNTERS`	10	`generic/tclHash.c`,line 306 `generic/tclLiteral.c`,line 898
`TCL_LSORT_STATIC_MERGE_BUCKETS`	30	`generic/tclCmdIL.c`,line 2680

Table 4: Switchable temporary variables

Macro	Heap	Stack
TEMP(t)	t *	t
ITEM(var,item)	var → item	var . item
REF(var)	(var)	&(var)
NEWTEMP(t,var)	(var) = (t *) ckalloc(sizeof(t))	
RELTEMP(var)	ckfree((void*)(var))	
STRING(n,var)	char* var	char var [n]
NEWSTR(n,var)	(var) = (char *) ckalloc(n)	

To identify the hot spots of stack usage, and from there the variables and data structures causing them, we semi-automatically instrumented a copy of the base sources with code monitoring the entrance to and returns from all functions in the interpreter and also recording the the location of the stack pointer for each call.

This instrumentation was only semi-automatic because the script used to insert the code was very simple and did not recognize all possible C syntax for return statements and function calls. About 25 % of the instrumented code had to be reworked manually to get the instrumentation code to work properly in them. There were two big offenders in this regard.

- The regular expression engine. It hid lots of return statements in C preprocessor macros and also used a number of unbraced if statements, i.e.

  ```
  if (...)
     return X;
  ```

  causing the instrumentation script to change the semantics of the instrumented code.

- Tail-calls scattered throughout the code, i.e.

  ```
  foo ()
  {
      ...
      return bar ();
  }
  ```

  These were rewritten to

  ```
  foo ()
  {
      ...
      X = bar ();
      return X;
  }
  ```

  allowing the insertion of the instrumentation code between the called function and the return of the caller itself.

The inserted code manages a logical duplicate of the C call stack on the heap and uses that to record both the names of the active functions and the locations of the cpu stack pointer for them. By comparing the location of the stack pointer before the call of a function F and its location inside, it was possible for us to deduce the amount of C stack space consumed by F.

This method also records the amount of stack required for function linkage, i.e. stack management done by the C compiler. This is no true disadvantage as this amount is usually small, and more important, constant. Because of the latter this does not disturb the ranking of functions when sorting them by the amount of stack they require.

As the execution of the inserted code causes a noticeable slowdown of the interpreter additional processing beyond the comparison of the stack pointer is not done. Instead we write the information about the association between functions and required stack directly to a log file. After each run a number of external scripts is used to postprocess the logged data. The result is a sorted list where the functions consuming the most stack are shown at the top. Thus identifying the hot spots to look at.

Instead of trying to devise a Tcl application which exercises all parts of the interpreter we simply executed the testsuite that comes with the Tcl core to generate the stack traces.

5. STACK REDUCTION RESULTS

To test the effectiveness of our changes to the core in reducing usage of stack we used the testsuite again, but with a twist. Using a special script we ran each of the 7879 tests multiple times, with an ever-shrinking stack, until it failed. The size of the stack for which a test failed first was recorded. The initial size of the stack was 48 K and reduced by 1 K per iteration.

Running this with unmodified and modified cores then allowed us to compare how the consumption of stack was changed by our modifications. The results of these comparisons for selected build variants are shown in the Table 5.

The first three columns in each these two tables list the minimal amount of stack required by any test during its execution, the maximal amount required by at least one test during its execution, and the average amount of stack required to execute a test.

For the values in the last three columns each configuration (except for the first) was compared to the configuration preceding it in the table. To facilitate this the system computed the difference in stack consumption for each test and then recorded minimal, maximal and average difference. A positive difference means that the configuration required less stack than the one it was compared to.

The line "Base" in Table 5 refers to the unmodified 8.3.4 core. It was compiled with basic compiler optimizations (-O). The other three lines refer to the modular core with its non-recursive byte code engine. It was always compiled with basic compiler optimizations (-O), but different settings for the switches affecting the usage of the C stack. Which switches were activated is listed in Table 6. An entry of "—" means that the switch was not activated and set to its default value.

We can see that the basic overhead of stack required by the test framework amounts to 31 K for the unmodified core. There are two tests breaking the initial limit of 48 K stack, these are "interp-29.1" and "interp-29.2". Both are testing the handling of the recursion limit of the Tcl core.

The introduction of the non-recursive engine alone (configuration "Stack (Zero)" reduces the average consumption of stack by nearly 4 K. Especially noteworthy is that now no test is breaking the initial limit of 48 K anymore, not even the two tests checking the recursion limit mentioned before. Actually these two tests now require nothing more than the new minimum overhead of 27 K stack, accounting for the reported maximal reduction of 21 K.

Adding the basic set of switches (configuration "Stack (Basic)") to shrink variables on the stack, or move to them to the heap reduces the consumption of stack by another near half K, bringing the reduction up to slightly more than 4 K on average. So these switches do have an effect, but not as much as the new byte code engine.

Setting all possible switches (configuration "Stack (All)")

Table 5: Stack consumption changes

	Min	Max	Avg	d/Min	d/Max	d/Avg
Base	31	48	31.227			
Stack (Zero)	27	39	27.449	0	21	3.779
Stack (Basic)	27	35	27.065	0	8	0.384
Stack (All)	27	35	27.065	-3	2	-0.001

Table 6: Stack reduction switches

Macro	No settings	Basic setting	Full setting
`TCL_DSTRING_STATIC_SIZE`	—	1	1
`TCL_FMT_STATIC_FLOATBUFFER_SZ`	—	1	1
`TCL_FMT_STATIC_VALIDATE_LIST`	—	—	1
`TCL_FOREACH_STATIC_ARGS`	—	1	1
`TCL_FOREACH_STATIC_LIST_SZ`	—	1	1
`TCL_FOREACH_STATIC_VARLIST_SZ`	—	—	1
`TCL_RESULT_APPEND_STATIC_LIST_SZ`	—	1	1
`TCL_MERGE_STATIC_LIST_SZ`	—	—	1
`TCL_PROC_STATIC_CLOCALS`	—	1	1
`TCL_PROC_STATIC_ARGS`	—	—	1
`TCL_INVOKE_STATIC_ARGS`	—	1	1
`TCL_EVAL_STATIC_VARCHARS`	—	—	1
`TCL_STATS_COUNTERS`	—	—	1
`TCL_LSORT_STATIC_MERGE_BUCKETS`	—	—	1
`TCL_STRUCT_ON_HEAP`	n	y	y

we find that this is actually slightly worse than using only the basic set. It is unclear however if this a measuring problem or a true worsening. It is also unclear if the expected further reduction is missing because the testsuite is not exercising these parts of the core, or if the affected variables cause a reduction which is too small to be measured at all.

6. CONCLUSIONS

It is possible to reduce the size of the static interpreter, however the exact reduction is highly dependent on the features chosen for removal.

In contrast reducing the amount of C stack consumed by the interpreter was a definite success, with Miguel Sofer's new byte code engine accomplishing most in this area. It is highly recommended to port this engine to Tcl 8.4. The other changes also accomplish their goal, but not as much as the NRE.

7. FUTURE WORK

The work on the modularized core is currently on hiatus. However, before that happened, we briefly investigated the following possibilities for additional optional features, with estimates for amount of work and possible gain. It should also be noted that the sources of the modularized core are available through the Tcl CVS at SourceForge, under the branch-tag `mod-8-3-4-branch`. The license is the same as for the unmodified Tcl core itself.

The estimates are given in both Lines Of Code (LOC) for the sources, and a percentage of the total size of the static library. The lines of codes were counted using "wc -l". Nothing was done to take comments into account. This means that the percentages given below can be seriously off (overestimation) given the extensive commenting of the Tcl core code. The percentages are based on the contents of Table 7 and Table 8, which list the sizes of the various object files.

The whole interpreter (115 files, matching the glob pattern `tcl/{generic,unix}/*.c`)[2] comes in at 3214256 LOC and 486648 Byte. This is 100 %.

1. Removal of the event system at large (Commands `[after]`, `[vwait]`, and `[update]`).

File	Touched	
tclInt.h	1 line	
tclTimer.c	1129 lines (all)	
tclBasic.c	3 lines	
tclEvent.c	547 lines (about half[3])	
tclNotify.c	1081 lines (all)	
tclUnixNotfy.c	1050 lines (all) [4]	
tclUnixEvent.c	77 lines (all) [4]	
tclWinNotify.c	522 lines (all) [4]	
tclMacNotify.c	581 lines (all) [4]	
	4991 lines	0.15 %
	binary	2.30 %

2. Removal of the handling of binary data (Command `[binary]`).

File	Touched	
tclBasic.c	1 lines	
tclInt.h	2 lines	
tclBinary.c	1552 lines (all)	
	1555 lines	0.04 %
	binary	1.60 %

Alternative: Leave Tcl_ObjType "tclByteArray" in.

[2] And without files supporting the testsuite
[3] The file contains code exit handlers too, which have to stay.
[4] These files are not relevant to the IOS port, but will still have to be deactivated in the standard core.

Table 7: Object sizes I

Object file	#byte	% of Total
regcomp.o	40368	8.30
tclExecute.o	26540	5.45
tclIO.o	25296	5.20
tclCmdMZ.o	17160	3.53
tclBasic.o	16656	3.42
tclVar.o	16584	3.41
tclCompCmds.o	16256	3.34
tclCompile.o	16024	3.29
tclCmdAH.o	14228	2.92
tclNamesp.o	13712	2.82
tclUtf.o	13396	2.75
tclDate.o	12752	2.62
tclCmdIL.o	12660	2.60
tclFileName.o	12020	2.47
tclPosixStr.o	10972	2.25
tclInterp.o	10196	2.10
tclEncoding.o	10008	2.06
regexec.o	9296	1.91
tclParse.o	9252	1.90
tclUnixChan.o	9196	1.89
tclUtil.o	8600	1.77
tclIOCmd.o	7828	1.61
tclBinary.o	7804	1.60
tclScan.o	7380	1.52
tclProc.o	7288	1.50
tclUnixFCmd.o	6604	1.36
tclParseExpr.o	6452	1.33
tclUnixInit.o	6040	1.24
tclPipe.o	6040	1.24
tclPkg.o	5544	1.14
tclObj.o	5520	1.13
tclCompExpr.o	5512	1.13
tclFCmd.o	5272	1.08
tclStringObj.o	4728	0.97
tclUnixPipe.o	4196	0.86
tclTimer.o	4088	0.84
tclEvent.o	4036	0.83
Total	486648	100.00

Table 8: Object sizes II

Object file	#byte	% of Total
tclListObj.o	3760	0.77
tclIOGT.o	3760	0.77
tclRegexp.o	3700	0.76
tclResult.o	3608	0.74
tclLoad.o	3556	0.73
tclUnixFile.o	3324	0.68
tclIOUtil.o	3300	0.68
tclMain.o	3296	0.68
tclHash.o	3296	0.68
tclStubInit.o	2960	0.61
tclLiteral.o	2784	0.57
tclNotify.o	2596	0.53
tclEnv.o	2396	0.49
tclClock.o	2304	0.47
tclLink.o	2240	0.46
tclGet.o	2164	0.44
regerror.o	1972	0.41
tclUnixNotfy.o	1784	0.37
tclIndexObj.o	1668	0.34
tclPreserve.o	1500	0.31
tclResolve.o	1228	0.25
tclThread.o	1216	0.25
tclUnixTime.o	1160	0.24
tclCkalloc.o	1116	0.23
tclLoadDl.o	1044	0.21
tclAsync.o	1028	0.21
tclIOSock.o	992	0.20
tclStubLib.o	980	0.20
tclHistory.o	920	0.19
tclPanic.o	876	0.18
tclAppInit.o	776	0.16
tclUnixSock.o	760	0.16
tclUnixEvent.o	752	0.15
tclAlloc.o	620	0.13
tclMtherr.o	616	0.13
regfree.o	560	0.12
tclUnixThrd.o	532	0.11
Total	486648	100.00

File	Touched	
tclBinary.c	1027 lines (2/3 of file)	
	1030 lines	0.03 %
	binary	1.06 %

3. Removal of the handling of times and dates (Command `[clock]`).

File	Touched	
tclBasic.c	1 lines	
tclInt.h	2 lines	
tclClock.c	377 lines (all)	
tclDate.c	1873 lines (all)	
	2253 lines	0.07 %
	binary	3.09 %

4. Removal of the package system (Command `[package]`).

File	Touched	
tclBasic.c	1 line	
tclInt.h	2 lines	
tclPkg.c	979 lines	
	982 lines	0.03 %
	binary	1.14 %

5. Removal of new string manipulation functionality (Command `[string]`).

File	Touched	
tclCmdMZ.c	1331 lines[5]	
	Lines at most	0.04 %
	binary	1.58 %

6. Reverting `[lsort]` to a quicksort based implementation, cutting out our own mergesort-based implementation.

File	Touched	
tclCmdIL.c	678 lines (lsort)	
	Lines at most	0.02 %
	binary	0.54 %

7. Removal of the bytecode compiler.

File	Touched	
tclCompCmds.c	2043 lines (all)	
tclCompExpr.c	1051 lines (all)	
tclCompile.c	3414 lines (all)	
Entrypoints ...	300 lines (estim.)	
	6808 lines	0.21 %
	binary	7.76 %

8. Remove of the bytecode executor. This implies the removal of the bytecode compiler. Without execution of bytecodes its compilation makes no sense.

File	Touched	
tclCompCmds.c	2043 lines (all)	
tclCompExpr.c	1051 lines (all)	
tclCompile.c	3414 lines (all)	
tclExecute.c	6412 lines (all)	
Entrypoints ...	300 lines (estim.)	
	13220 lines	0.41 %
	binary	13.21 %

9. Removal of regular expressions.

File	Touched	
regc_color.c	17775 lines	
regc_cvec.c	5094 lines	
regc_lex.c	24495 lines	
regc_locale.c	34453 lines	
regc_nfa.c	36234 lines	
regcomp.c	59492 lines	
rege_dfa.c	17820 lines	
regerror.c	3515 lines	
regexec.c	28360 lines	
regfree.c	2086 lines	
regfronts.c	2394 lines	
tclRegexp.c	1029 lines	
	232747 lines	7.24 %
	binary	11.5 %

10. Removal of namespaces (Command `[namespace]`).

File	Touched
tclNamepace.c	3916 lines (mostly)[6]
tclVar.c	4813 lines
tclParse.c	2357 lines
tclParseExpr.c	1870 lines
various (set, proc)	1000 lines

A simple cut-out of this feature is not possible, we will rather have to rewrite parts of the parser, and of commands like `[set]` and `[proc]` to remove the special handling of the colon (:), the namespace separator character, from the system.

About 13956 LOC have to be touched for this, which is about 0.43 %. Circa 2.82 % of the binary are definitely removed.

11. Removal of encodings (Command `[encoding]`). We did not count the size of the then irrelevant encoding files, as we did not count them as part of the whole code base either.

File	Touched
tclEncoding.c	2871 lines

How much is removed from the file above depends on the chosen model, of which we have two:

(a) Deactivating encodings completely may remove about 80-90 % of that file. This are circa 1.06 % of the binary.

Not everything might be removed because we believe that the best way to remove UTF8 completely is to rewrite the Utf ¡-¿ External converter functions and throw away the rest. That way we don't have to think about all the other places which do UTF8.

If we remove this completely we have to touch many more places throughout the whole code, most notably the channel system. The latter would bring the number of LOCs removed or rewritten up, but also takes much longer. We currently have no good LOC estimate for this scenario.

As a first approximation we grep'ped the sources for "Tcl_Utf", which gives us 314 locations in 40 files. We guesstimate that each location translates into 1-4 lines of code touched directly. And

[5] Just `Tcl_StringObjCmd`

[6] There are some routines which deal which the call stack, these have to stay.

depending on context maybe 5-20 others around each location which have to change too. That would be between 314 and 6280 LOC changed, i.e. replaced with different, non-UTF, code.

The number of 5-20 other lines depending on context might be an underestimation for the channel system. This part of the core will very likely need a complete reorganization to allow usage both with and without encodings. This would be 8389 lines changed in tclIO.c. Changed, not cut!

But also note the fact that `tclIO.c`is with 5.20 % of the binary also the third-largest file right now.

Summary: About 17540 LOC have to be touched for this, which is about 0.54 % of the whole sources.

(b) whereas just deactivating the loading of external encodings may remove 60-70 % of that file.

12. The generic part of the I/O system.

    While this subsystem is with 8.46 % of the binary code the third-largest part of the core after regular expressions and the engine for the execution of bytecodes, it also a tangled web and in our opinion at least very difficult to unravel.

    Especially as it is heavily influenced by the choice of whether to use encodings or not, and also if it has to support the notifier or not, i.e. file events.

    No estimates were made for this part of the core.

It was noted before that Source Navigator crashed when processing the Tcl sources. This not the case for the newest version, 5.1. This means that our ability to determine which parts of the code have to be made conditional, or are dependent on more than one feature is greatly enhanced. Of course, we will have to write special scripts which mine the dependency database for the information we need. This however is less difficult than searching through the sources by ourselves, and less error-prone.

Such help is especially important for a future up-port of the modularization changes to 8.4. The internal organization of the code has changed so much that the patches we could generate from the comparison of an unmodified versus an modularized 8.3.4 core are essentially useless. The only parts which can be lifted over relatively easily will be the changes to reduce the consumption of stack space.

APPENDIX

A. REFERENCES

[1] Karl Lehenbauer
A Tcl-Powered Handheld Computer For Telecommunications Test Automation.
OSCON 2001, San Diego, July 2001

[2] David E. Smyth
Tcl And Concurrent Object-Oriented Flight Software: Tcl on Mars.
2nd Tcl/Tk Workshop, New Orleans, 1994.

[3] John Hall
Rivendell, aka PalmTcl
`http://sourceforge.net/projects/palmtcl`
A more current version is available at `http://rivendell.sourcefubar.net`

[4] Ashok Nadkarni
Palm Tcl (note the space)
`http://palm-tcl.sourceforge.net/`

[5] Alexandre Ferrieux
Call For Ideas: How to modularize ?
`http://groups.google.ca/groups?th=ca2f3925760d67a`
`news:comp.lang.tcl`, 1998