Tcl Capacities in Crisis for EDA

By

Phil Brooks - Mentor Graphics Corporation

&

Sridhar Srinivasan - Mentor Graphics Corporation

Presented at the 23nd annual Tcl/Tk conference, Houston Texas, November 2016

Mentor Graphics Corporation
8005 Boeckman Road
Wilsonville, Oregon
97070
phil_brooks@mentor.com

sridhar_srinivasan@mentor.com

Abstract: Tcl has been a significant component in many EDA applications for many years and it still remains so today. It has worked well, despite the various limits on data structure sizes, because previously, in cases where there were large data sets, native C++ data structures have been used while Tcl worked around the edges, dealing with reasonable small subsets of data, control files, data structures, and meta-data. With the latest integrated circuit manufacturing technologies, even these previously limited data sets are frequently exceeding Tcl's internal size limits like the maximum size of a list, maximum size of string, etc. As EDA goes, the rest of the industry follows, so the number of applications that can reasonably expect to create very large lists, arrays, strings etc. is quickly growing.

The body of the paper discusses several real cases cases where what used to be reasonable subsets of design data that could easily be expected to be limited to within a fairly small size have suddenly grown to exceed the limits on size of a Tcl list, array, or string. We

will look at recent trends in EDA analysis for both the aggregate data (things like the number of devices or nets in a chip design) as well as trends in the size of what used to be 'safe' data sets for Tcl - like the number of devices in a cell on a net, or the number of polygons in a cell on a net, or the length of a verification program deck, the number of cells in a design etc. and show that even reasonably small subsets of data are frequently growing to exceed the limits imposed by Tcl. In addition, customers are requiring more ad hoc exploration and analysis of their designs - these sorts of analysis are ideally suited toward analysis using Tcl.

It is critical today for the EDA industry today that we start moving toward a Tcl 9 implementation that removes all of the historical 32 bit size limitations.

Mentor Graphics Calibre Verification Tool

The Calibre design verification tool is used by chip designers to programmatically verify their chip designs using several different analysis applications. These applications share a common database and a set of geometric analysis algorithms that are used to perform the following functions, among others:

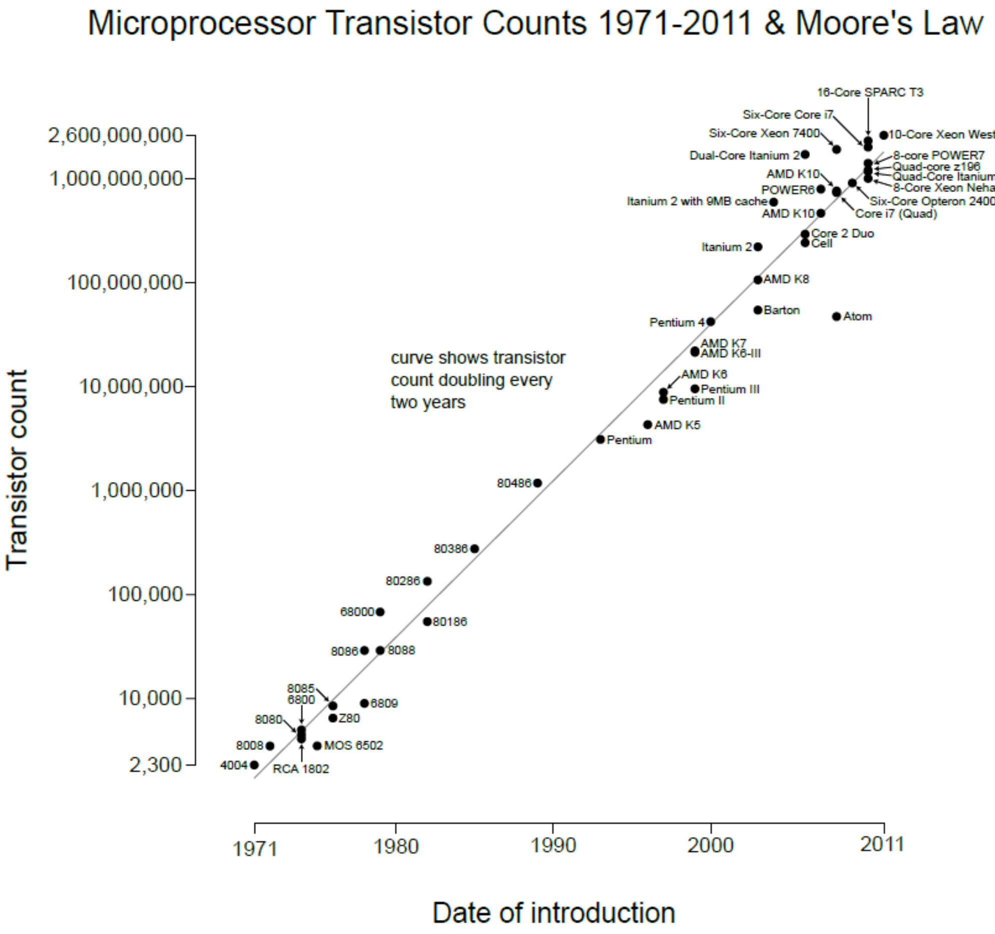
- Design Rule Checking (DRC) – Measures design features and checks these measurements against predetermined constraints of a semiconductor manufacturing technology.
- Layout vs. Schematic comparison (LVS) – Uses descriptions of signal connectivity and device construction in a design to turn purely geometric layout database input into a full device schematic. Then that device schematic is compared to the engineering schematic (source netlist) to see if they are electrically equivalent. Tcl is used extensively within the LVS application for rule deck construction and property calculations and comparisons.
- Parasitic Extraction (PEX) – Calculate resistance and capacitance of design components and add these to the intentionally constructed device schematics derived during LVS.
- Programmable Electrical Rule Checking (PERC) – Programmatically accesses the LVS netlists using a Tcl based command language to check electrical constraint conditions.

Tcl is used in Calibre to construct rule check decks as well as for certain property calculation operations. Calibre Perc uses Tcl extensively as a command language. The following case studies show several situations, encountered over the last year, that illustrate the explosion is size of some of the data sets that have been amenable to Tcl in the past, but that are now nearing or going beyond the hard limits on Tcl data structures and causing performance problems or runtime product crashes. We also explore some of the ways that small problems can turn into large ones when large data structures are copied inadvertently within a Tcl program.

EDA Trends

Several trends are causing an explosion of design data required for typical EDA designs that go beyond the simple increase in the number of devices and nets involved in the design:

- Feature size – This is the expected change in manufacturing technology feature size from the familiar march of Moore's law. Reducing feature size allows creation of more devices on a chip. Here is some information on manufacturing feature sizes that have been made available over the past years:

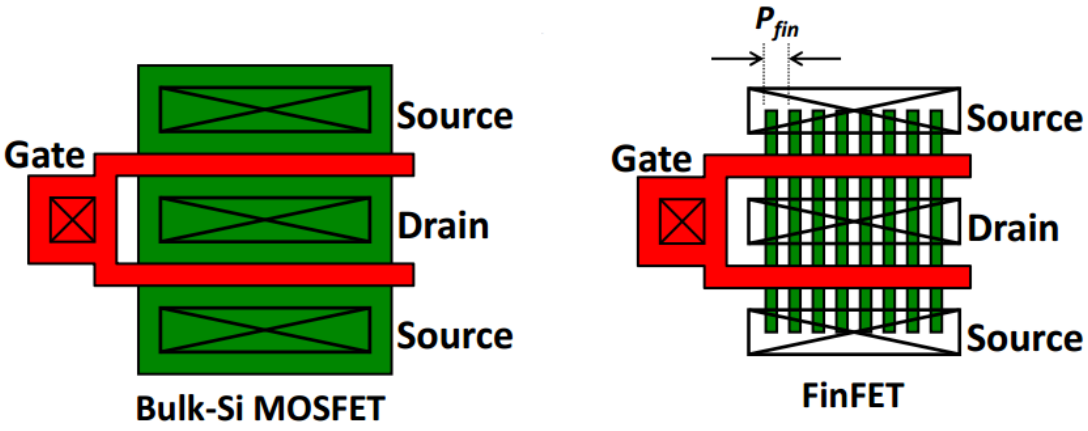


top	Tech	Transistors	Cell Count	Max Nets/Cell	Device Types	ESD Check Pin Pairs
2005	65mn	250 million				
2007	45nm	500 million				
2009	32nm	1 billion	200,000	3 million	100	400,000
2011	22nm	2 billion				
2013	16nm	4 billion	500,000	6 million	200	
2016	10nm	7 billion		4 billion		6 million

2017	7nm	9 billion				19 million

Transistor counts are on the leading edge implementations and typically grow over the life of a particular technology. More complete information is available at https://en.wikipedia.org/wiki/Transistor_count.

- Increased device complexity – The newest device technology, being used at 16nm and below, are called FINFET devices. These devices are much more complex than previous generation devices and there are special construction artifacts that are required in the design files to construct these. In the end, this means that there is a lot more polygon data required to create a single device or group of devices.



Diagrams from: https://people.eecs.berkeley.edu/~tking/presentations/KingLiu_2012VLSI-Tshortcourse

- Increasingly complex POWER delivery network – In order to manage power in modern devices, a great deal of complexity has been added to the way modern chips are powered. More available voltages and more switchable power domains offer a mechanism to provide different parts of the chip with different voltages and to turn parts of the chip off at different times rather than the old one size fits all approach. These also lead to a big increase in the number of types if devices that designers must use to manage power efficiently.
- Increasingly complex planarity requirements – Semiconductor manufacturing processes require that certain design layers, like metal and polysilicon, have fill polygons present to meet density and

planarity manufacturing constraints. These don’t serve a function on the chip, but they are required for manufacturing processes like chemical-mechanical planarization or polishing so that the polishing results in even planar surfaces without dishing or bumping as a result of mixed materials on the surface being polished. Requirements for fill are much more complex now than in past designs. This often results in design flattening – where repeated design entities that used to be represented in a hierarchical fashion are now represented in a more flat repeated fashion. That results in larger datasets and

- Increased complexity of Electro Static Discharge (ESD) protection circuits – During the manufacturing process, static electricity can be discharged onto the chip. If this shock carries high enough voltages to sensitive circuitry on the chip, it can overload it and destroy that portion of the chip. To protect against this, ESD protection circuits are constructed around the I/O pads on the chip. While the feature size of the devices on the chips goes down, the size of potential ESD shocks remains fairly constant. As a result, it takes a lot more protection circuitry to absorb a similarly sized discharge. Protection circuits that used to consist of a few dozen diodes and transistors in 65mn chips have grown to hundreds of thousands of diodes and transistors in 28nm chips.

EDA Case Study – Moderately large string becomes too large for Tcl

One common technique in our Perc Electrical Rule Checking (PERC) check Tcl programs is to perform voltage dependent spacing checks that require placing properties like potential voltages on specific nets in a design and then performing analysis on those nets. Initially the number of nets that were marked with these voltage markers were fairly limited. As a result, the following algorithm was used:

- Identify specific locations for voltage markers in the design
- Create markers using a command syntax for creating markers at a specific location having a specific voltage range. This command syntax looked like this:

```
Marker_layer = DFM CREATE LAYER POLYGON NODAL [
    2 831.234 234.290 831.236 234.292 cell CELL1 net 1
```

```
            2 431.234 34.290 431.236 34.292cell CELL2 net 2
                   ....
       ]
```

- Feed the marker string to a Tcl command that would create the markers in our internal database.
- Analysis was performed on the markers and results were presented.

For the original 65nm designs that used this check, several hundred thousand of these markers might typically be generated.   This technique is capable of working with the hierarchy of a design, so that a single marker can be placed on a net that is replicated throughout the design very efficiently by simply creating one marker on the un-replicated net. Recently, in a 20nm design (2014 technology) a customer was found instead to be placing a large number of voltage markers at a high level in the design in order to account for minor voltage differences in a few specific areas. For this case, several hundred million markers were created. As a result, the command file expanded to well beyond the 2GB Tcl string limit and a crash resulted.

Mitigation

C++ interfaces were developed to break up the groups of nets by their voltage ranges in order to handle nets in smaller groups. The marker commands were written to a file on disk and then that file was opened and the markers were created one at a time.

EDA Case Study – Rule Deck Size

The first case follows ramifications of the growth in size of the Calibre rule deck. The rule deck is essentially a program that tells a Calibre application how to go about its analysis of a customer design.   Rule decks have been growing substantially as new technologies require more and more in depth analysis, devices become more complex, and process design rules become more stringent.   Rule decks have grown from .5-1 k bytes 20 years ago, to 2-4 megabytes bytes now, but recently additional application data was added into the rule deck resulting in rule files that weighed in at 1GB in size. As a result, Tcl code that could previously easily examine a rule deck, copy it, generate some new rules etc. would not run into any problems. Once the new 1GB rule decks started going through the same code, customers saw severe performance issues that weren't easy to correct.

For example, one rule deck analysis function looked roughly like this:

```
# Turn the entire rule file text into a Tcl string
set db_rules [dfm::get_svrf_data $db_rules_obj -freeze]

# Now, take the single string and split it into a list of lines
foreach line [split $db_rules '\n'] {
   ...
}
```

The net effect of the above is to take the 1GB in memory copy of the rulefile, copy it into a Tcl string (+1GB) and then split that into a list of single line strings (+1.7GB). As a result of making several passes through that analysis, our extra 1GB of file input data turned into over 20GB of process size. In this case, Tcl handled everything fine, because no single piece of data was never beyond 1GB.

Mitigation

One possible way to avoid some additional memory usage might be to use the string package to avoid splitting the big returned string into a list. So, for example, the following code might be used to look at the big returned string line by line, but avoid splitting it into a list by using '*string first*':

```
    set end [ string length $bigstring ]
    set curr 0
    while { $curr < $end } {
        set last [ string first "\n" $bigstring $curr ]
        if { $last == -1 } {
            set last $end
        }
        set line [ string range $bigstring $curr $last ]
        puts -nonewline $outfile "$line"
        set curr $last
        incr curr
    }
```

We found that this, counterintuitively, also requires significant additional memory. After a query on comp.lang.tcl, Don Porter found that this additional memory occurs because the string is converted into a Unicode string by the *'string first'* command. Don believes that an optimization that avoids conversion to Unicode is possible in this case.

Another work around that currently avoids the conversion to Unicode is possible with *'string index*':

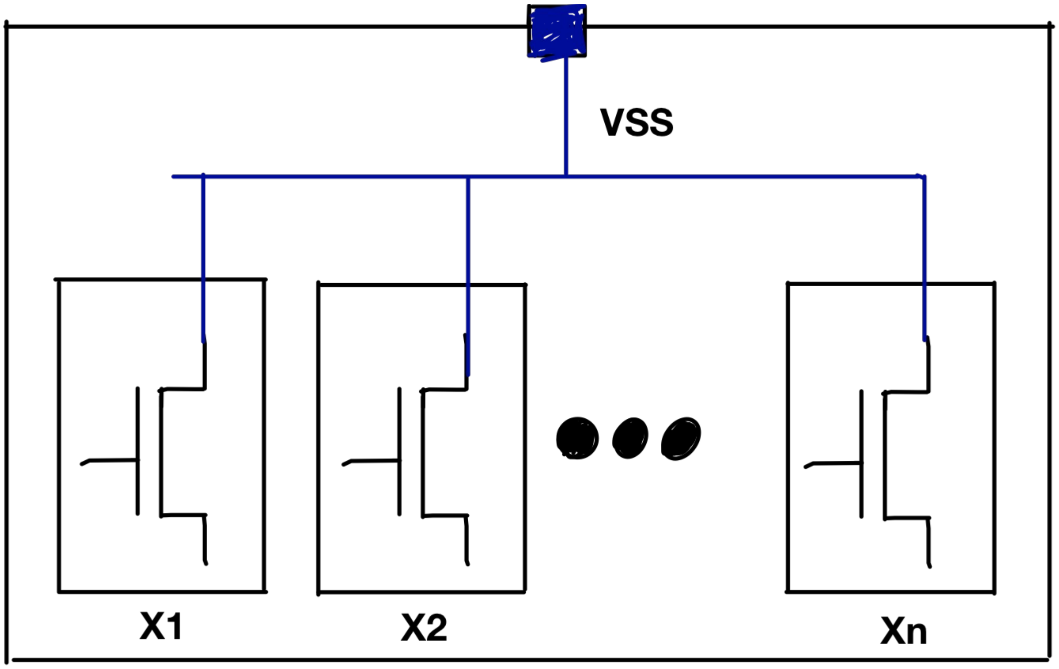
```
set end [ string length $bigstring ]
set curr 0
set last 0
while { $curr < $end } {
    set c [ string index $bigstring $curr ]
    if { [ string match $c "\n" ] } {
        set line [ string range $bigstring $last $curr ]
        puts -nonewline $outfile "$line"
        set last $curr
        incr last
    }
    incr curr
}
```

This approach is significantly slower than the previous approaches – likely due to the bytecode execution of the loop vs. 'C' execution in *'string first'*.

This particular problem was solved by avoiding making the initial copy of the rule text string. C++ interfaces were developed to provide access to the parsed rulefile structures that were needed, so the call to dfm::get_svrf_data $db_rules_obj –freeze was removed along with the list based look through each line of the rule deck.

EDA Case Study – Number of devices on a single net

While the complexity of the power supply on a modern chip keeps the number of devices on a single power net from getting too high, many modern chips are using a single ground net for the entire chip. As a result, the number of devices on that single ground net has grown very significantly.



set result [perc::count -net $net -list]

In modern designs many devices have direct connections to Ground, leading to huge number of devices

An application that created a list of all devices on the ground net in our Perc application looks like this:

```
set result_list [ perc::count –net $net –list ]
```

Since most of the devices on the chip have a connection to ground, the resulting list length now frequently surpasses the length of a Tcl list.

Mitigation

This problem was solved by returning an “iterator” – which is a Tcl command object that allows you to access what would have been items in the list one by one. So, instead of writing:

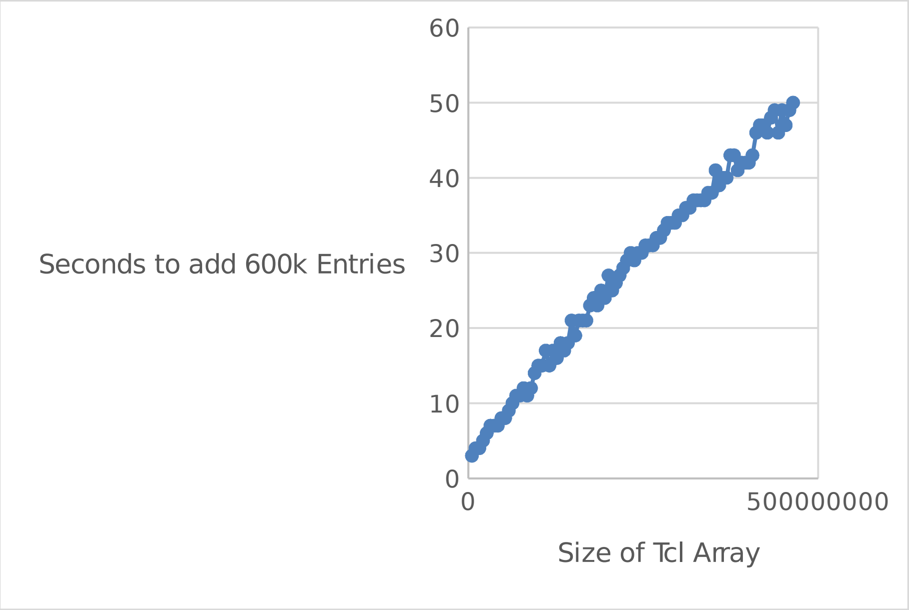
```
set result_list [ perc::count –net $net –list ]

foreach item $result_result_list {
        ...

    }
```

Now you write:

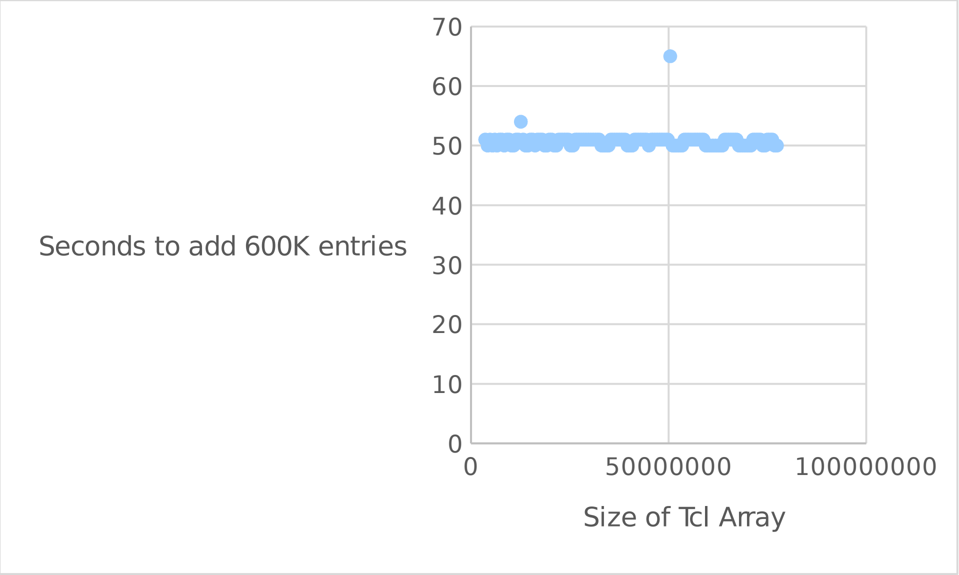
```
    set result_iter [ perc::count –net $net –iter ]

    while { $result_iter } {

        ...

        dfm::incr result_iter

    }
```

EDA Case Study – Net Voltage checks

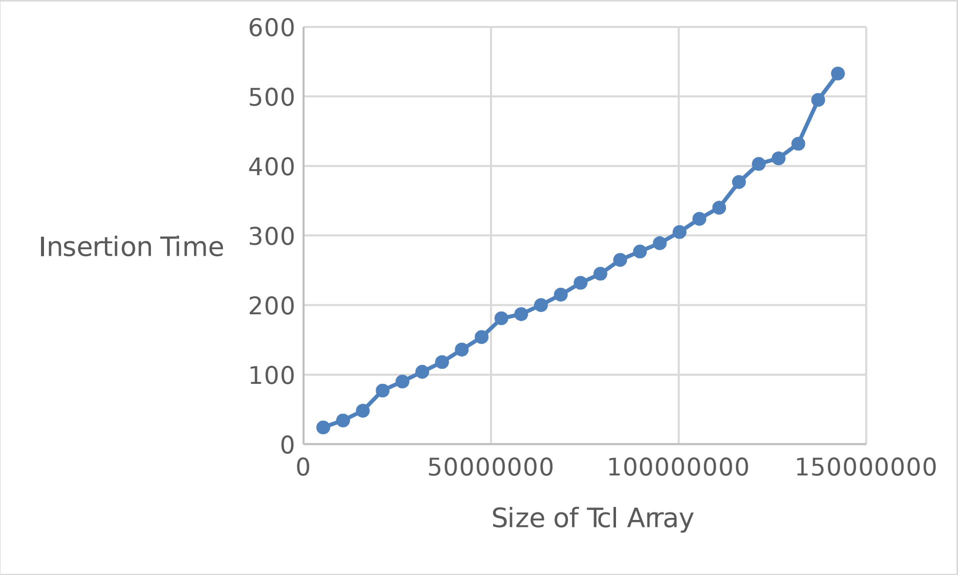
One ramification of the complex power networks that drive chips today is that there are different voltages running in different parts of the chip. This leads to the need for design rules where net spacing depends upon voltage present on the net. That leads to a need to check, for each electrical net on the chip, to see if it is too close to incompatible voltages.   Our solution for doing this involved creating a large Tcl array that used Cell name and Net name as a key and contained a complex nested list of a couple dozen short numbers and strings as the array values.   As the designs grew, we found that adding net specific data to the large array slowed significantly as the size of the size of the array grew:



Reproducing this slowdown in a standalone Tcl script proved tricky. Simply creating a similarly sized list using similar keys and similar data payload showed very flat results as the size of the array grew:



Varying the key length and payload size did nothing to alter the performance curve. In the end, the native application data was all written to disk and converted into straight Tcl so as to avoid any dependency on the Calibre application and to figure out exactly what was causing the significant slowdown. In the end, the slowdown was called by the instrumentation code. It appears that frequent calls to the '*array size'* command can have a severe adverse performance impact on insertions into the array being called:



2015 WIP Revisited

In the 2015 conference, I presented a Work-In-Progress presentation that foreshadows this presentation – several data capacity issues were presented. This section revisits those limits and progress made on solving them in the 8.6.4 release.

Sourcing a large (>2gb) file:

This script will generate a very large file:

```
set outfile [ open "big.tcl" w ]
set count 120000000
set current 0
while { $current != $count } {
    puts $outfile "# a comment $count"
    incr current
}

puts $outfile "puts \"hello world\""
```

This still results in a core dump in Tcl 8.6.4, though a nice message is printed now. The size of the script that can be executed is now larger.

```
$ /usr/local/ActiveTcl-8.6/bin/tclsh big.tcl
max size for a Tcl value (2147483647 bytes) exceeded
Aborted (core dumped)
```

This creates such a problem for some of our customers that we have overloaded the source command with a line-by-line readin that doesn't have to create a string first.

Large List behavior

The following script used to slowly bog down as the limit on the size of a list was approached:

```
# Create a list with a billion elements

set i 0
set init_size 200000000

# Start with a fixed size allocation
set my_list [ lrepeat $init_size "" ]

set count 0
set t0 [ clock seconds ]

# Populate fixed size with data
while {$i < $init_size} {
       lset my_list $i "List element $i"
       incr i
}

# Now grow that list.
while {$i < 1000000000} {
       lappend my_list "List element $i"
       incr count
       if { [expr {fmod($count, 100000000)} ] == 0.0} {
          # Show progress every 100 million elements
          set tnow [ clock seconds ]
          puts "elapsed time: [ expr { $tnow - $t0 } ] seconds"
          puts [llength $my_list]
       } elseif { $count > 403000000 } {
          # Show progress more incrementally
          if { [expr {fmod($count, 1000)} ] == 0.0} {
              set tnow [ clock seconds ]
              puts "elapsed time: [ expr { $tnow - $t0 } ]
seconds"
              puts [llength $my_list]
          }
       }
       incr i
}
```

The allocations now go up to the limit without slowing down:

```
$ /usr/local/ActiveTcl-8.6/bin/tclsh list_limit_2.tcl
elapsed time: 455 seconds
300000000
elapsed time: 647 seconds
400000000
elapsed time: 838 seconds
500000000
max length of a Tcl list (536870909 elements) exceeded
    while executing
"lappend my_list "List element $i""
    ("while" body line 2)
    invoked from within
"while {$i < 1000000000} {
       lappend my_list "List element $i"
       incr count
       if { [expr {fmod($count, 100000000)} ] == 0.0} {
         ..."
    (file "list_limit_2.tcl" line 20)
```

Conclusion

Tcl has served the EDA industry well for many years and it is still very widely used in many applications. However, the capacity limitations that were initially nearly irrelevant and in recent years have become first minor annoyances and then critical problems are on the verge of making Tcl unusable for many EDA tasks. It is critical that we develop a fully scalable set of data structures for Tcl 9.0 before this happens!   Also, Tcl provides an ideal scripting language for end user exploration and analysis in EDA design. These end users are frequently completely unaware of the Tcl data size limits and they have no ‘C’ or ‘C++’ recourse if their data structures blow up.