Pulling Out All the Stops

Implementation of a High Performance Tcl Extention in the Calibre IC Design VerificationTtool.

by Phil Brooks - software engineer, Mentor Graphics Corporation

12th Annual Tcl/Tk Conference
Portland, Oregon
October, 2005

ABSTRACT

The Calibre IC Design Verification environment has used Tcl for a number of years in a traditional scripted GUI architecture where Tcl drives underlying high performance C++ application code. An existing high performance limited functionality built-in scripting language provides users with custom calculation capabilities in some performance critical areas. When users needed greater flexibility and extensibility for some low level scripted calculations, I turned to Tcl to provide them. The catch was that these calculations are at the heart of one of Calibre's performance sensitive, multi-threaded C++ analysis modules. User provided Tcl procs will be called millions of times from within threaded C++ application code in a performance critical area. The existing build-in language will provide the underlying application framework for Tcl user procs. I examine the architecture of this system that provides maximum user provided Tcl proc call performance from inside a performance sensitive C++ application.

The Calibre Application

The Calibre Verification tool does geometric analysis on integrated circuit designs. The geometric data manipulations are programmed by the user in a specialized geometric manipulation language called Standard Verification Rule Format (SVRF). Using SVRF, users can perform various functions that help them to discover geometric properties of the design that can in turn be used in verification analysis. This paper concerns use of Calibre for analysis of devices (i.e. transistors, resistors, etc) and calculation of properties related to those devices. Here is an example of a transistor layout with a GATE (RED) polygon that is in contact with two PIN polygons (GREEN) connected to metal leads (BLUE) polygons that are attached by contacts (GREY) polygons.

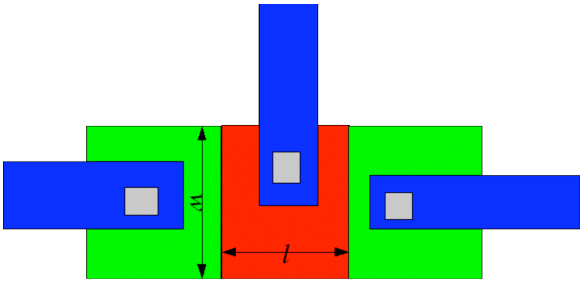
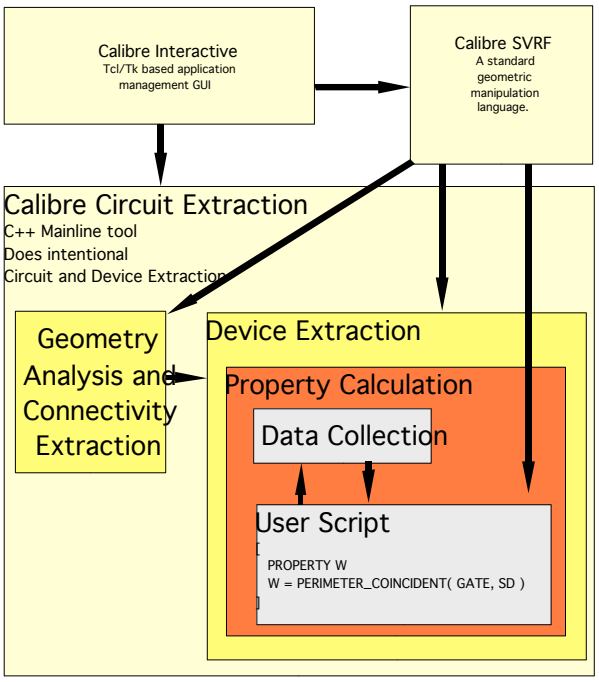


Figure 1 - Transistor Layout

Device Recognition

The device recognition facility in Calibre allows the user to search a layout database for intentional device structures (i.e. transistors, resistors, etc.) and calculate various properties concerning these structures for later use in applications like Layout vs. Schematic Comparison (LVS), and Circuit Simulation. This is an architectural overview of the Device Recognition portion of the application:



Calibre Built-in Scripting Engine

Calibre's simple user script functionality provides user defined device calculations based on geometric measurements made by Calibre.

These scripts are parsed and byte compiled before the device recognizer runs. After parsing, the user scripting module tailors the device recognizer's data collector to gather only the information that will be needed during the script's run time.

This built-in scripting language is extremely fast, but also has very limited capabilities (no looping constructs, no user defined function calls, only numeric and string data types, no dynamic allocation, etc). As integrated circuit technology advances, customers are requiring the ability to measure and calculate more and more complex device properties. Rather than making piecemeal additions to the existing scripting engine, these requirements could be handled by a general purpose scripting language like Tcl.

The Builtin Language

The property statement declares properties that are being calculated. Data collection functions provide raw data collected by the device recognizer. Conditions and arithmetic expressions allow final calculations of desired properties.

```
DEVICE MP PGATE PGATE(G) SD(S) SD(D) DIFF
[
  property w, l
  w = 0.5 * ( perim_co( S, G )
              + perim_in( S, G )
             + perim_co( D, G )
             + perim_in( D, G ) )
  l = area( G ) / w
  if( ( bends( G ) != 0 ))
  {
    if( w > l )
      w = w – 0.5 * bends( G ) * l
    else l = l – 0.5 * bends( G ) * w
  }
]
```

Code Example 1

One of the properties that makes the Calibre scripting language perform so well is the fact that it doesn't allow any dynamic allocation during the individual device calculation. Rather, each device is evaluated by a byte compiled program that has all potential variables and results preallocated at the device call time. Here is an example of a device property calculation program using the built-in language:

When Calibre runs, the script is parsed and the functions (perim_co, perim_in, area, and bends) are used to set up data collection for the device recognizer. As shapes are scanned, and results of these functions are stored in memory along with space for the temporary variables, property results, and the compiled byte code for the program that is going to run. After data is collected, the byte-code from the built-in script runs using the pre-allocated variables in the variable table. Conceptually, it looks like this:

Variable Name	Value
w	0.0
l	0.0
temp1	0.0
temp2	0.0
temp3	0.0
perim_co(S,G)	4
perim_in(S,G)	6
perim_co(D,G)	4
perim_in(D,G)	6
area(G)	12
bends(G)	2

Byte Code
ADD temp3 perim_co(S,G) perim_in(S,G)
ADD temp2 temp3 perim_co(D,G)
ADD temp1 temp2 perim_in(D,G)
MUL w 0.5 temp1
DIV l area(G) w
NE bends(G) 0 [31] [70]
GT w l [37] [55]
MUL temp2 0.5 bends(G)
MUL temp1 temp2 l
SUB w w temp1
GOTO [70]
MUL temp2 0.5 bends(G)
MUL temp1 temp2 w
SUB l l temp1

Recently a new device property data collection function was introduced called **ENCLOSURE_PERPENDICULAR** (ENC_PER for short). Unlike previously implemented functions that each return a single measurement value, ENC_PER returns an array of measurement triplets. The array returned has a length that is dependent upon actual shapes encountered by the device recognizer. This function required new fundamental capabilities in the built-in scripting language since it didn't have any looping or indexing constructs. Project constraints at the time would not allow for extensive additions to the scripting language. We thought about adding a more fully developed scripting language to the device recognizer at that time, but didn't have time for that either. Instead, the project team

decided to add a new SUM() function to our built-in scripting language. The SUM function created a summation capability that would allow users to describe operation expressions on the individual array slices and sum the results of that expression across the slices. Our previous example might look like the following once the customer adds use of the ENC_PER and SUM functions:

```
DEVICE MP PGATE PGATE(G) SD(S) SD(D) DIFF
[
  property W, L, SEFFA
  W = 0.5 * ( perim_co(S,G) +
     perim_in(S,G) + perim_co(D,G)
     + perim_in(D,G) )
  L = area(G) / W
  if( ( bends(G) != 0 ))
  {
    if( W > L )
      W = W - 0.5 * bends(G) * L
    else L = L - 0.5 * bends(G) * W
  }

  S = ENC_PER (PGATE, DIFF, SD, 25)
  SEFFA = W /
   SUM(S::W / (S::A + 0.5*L) ) - 0.5*L
]
```

Code Example 2

This program produces a byte code about twice as long as the previous one.

The SUM function satisfied the immediate requirements, but couldn't do other things customers later wanted, like calculate MIN, MAX, or other arbitrary calculations on members of the returned array. Later on, customers requested even more complex measurements that were implemented in another new data collection function called **ENCLOSURE_PERPENDICULAR_MULTI-FINGER**, (ENC_PER_MUL) that produced an array of arrays of measurement triplets. This array of arrays also has a length that is dependent upon actual shapes encountered by the device recognizer.

Given these new requirements, I considered the alternatives of adding multi-dimensional looping, allocation of temporary data, etc to our built-in language and adding a general purpose scripting language capability.

To use an external scripting language, I will need the following:

- A simple way to *describe the data collection required before it is needed*. In the existing language, this is done by simply parsing the script and looking for calls to data retrieval functions. The present system is very convenient for customers since use of a data collection function anywhere in the script is easily identified by the parser before the data collection routines have started. If I replace the existing scripting language with a language like Tcl, this will not be so easy since Tcl code can not be readily examined to find out what data collection functions might be called.
- I was also hoping to *leverage much of the existing scripting engine.* Customers have invested heavily in SVRF files that use the old scripting language. They understand how it works and don't want to have to rewrite any of their scripts just to use the new functionality presented by another scripting language. Also, I don't want to rewrite all of the engine's functionality from within a new script environment. 31 data collection routines are currently available in the existing scripting language. They all need to be available in the new environment.
- I need a way to *get data collected from Calibre to the script*.
- I need a way to *get calculation results from the script back into Calibre*.
- Overall performance can't slow things down too much.

Why Tcl?

At this point, I should discuss the detailed analysis of various scripting languages that I undertook, along with philosophical musings and the finer points of language grammar, ideology, geometry, and theology as it pertains to available scripting languages. All of this musing would, in this ideal version of this paper, lead to the inevitability of choosing Tcl as the one true scripting language for this application (this is, after all, a Tcl conference). In reality, I must admit that I chose Tcl for one reason, and one reason alone. It was already there. Tcl is available to Calibre customers for several other purposes inside the SVRF geometrical programming language. Shoveling another unrelated scripting language

into our application for use inside SVRF would have been confusing.

How Tcl?

Now the question is how to do this? I added a new function in our built-in scripting language that allows the user to call out to Tcl. This function is called **TVF_NUMERIC_FUNCTION** (or TVF_NUM_FUN) It looks like this:

```
DEVICE MP PGATE PGATE(G) SD(S) SD(D) DIFF
[
  property W, L, SEFFA
  W = 0.5 * ( perim_co(S,G) +
     perim_in(S,G) + perim_co(D,G)
     + perim_in(D,G) )
  L = area(G) / W
  if( ( bends(G) != 0 ))
  {
    if( W > L )
      W = W - 0.5 * bends(G) * L
    else L = L - 0.5 * bends(G) * W
  }

  S = ENC_PER (PGATE, DIFF, SD, 25)
  SEFFA = TVF_NUM_FUN (
          "calc_eff_a",
          "device_func ", S, W, L
  )
]
```

Code Example 3

The new function is designed to work just like the other built-in language functions except that it calls user specified Tcl code. The user's Tcl code is also stored in the SVRF rule file in a TVF FUNCTION block (TVF is our special name for Tcl inside of SVRF)

```
TVF FUNCTION device_func [/*
  proc calc_eff_a { enc W L } {
     ... tcl code ...
  }
*/]
```

Code Example 4

Lets see how this solution stacks up against our requirements:

- *describe the data collection required before it is needed* – Data collection needed is still defined by the scripting language – the Tcl code can't obscure any up front data collection requirements since its only interface is through call parameters.
- *leverage much of the existing scripting engine* – The example above shows clearly that existing customer code is not impacted, a simple one line addition to the existing script allows use of the Tcl functionality without impacting other calculations that the script performs.
- *get data collected from Calibre to the script* – I do this by passing parameters in to Tcl explicitly in the script.
- *get calculation results from the script back into Calibre*. – this will be done with the return value from the Tcl function.
- Overall performance can't slow things down too much – no way to tell this until I write some code. One nice thing about having the SUM function in the built-in language is that it allows us to compare runtimes for doing similar calculations in the two environments with one another.

The remainder of the paper looks at the implementations used to provide this feature and at performance of the resulting system.

Performance Testcase

To test the performance of this interface, I created two test circuits. The first contains 500,000 devices that have to be run through the device property calculation script. I used a device program that was extensive enough to be representative of customer device calculation scripts. The *SVRF Example Code* section in the code appendix shows the performance test scripts that were used. For each device processed, 3 Tcl procs are called, each returns a numeric result that can be stored as a property or used in additional calculations. The 3 Tcl procs call back into Calibre a total of 10 times for each device to retrieve data from the argument objects passed into the 3 functions. That gives us a total of 1.5 million Tcl proc calls and 5 million callbacks on each run of the testcase. In addition, the testcase is set up so that it provides good thread parallelization opportunities. During the run, the devices are broken up into 18 partitions that can run in parallel threads. The testcase has been artificially constructed to make the actual C++ device recognition algorithm run very quickly and therefore exaggerate the importance of the scripting component. The testcase only takes a few minutes to run, so evaluating performance impacts of our change is fairly simple.

Efficiently Calling Tcl Code

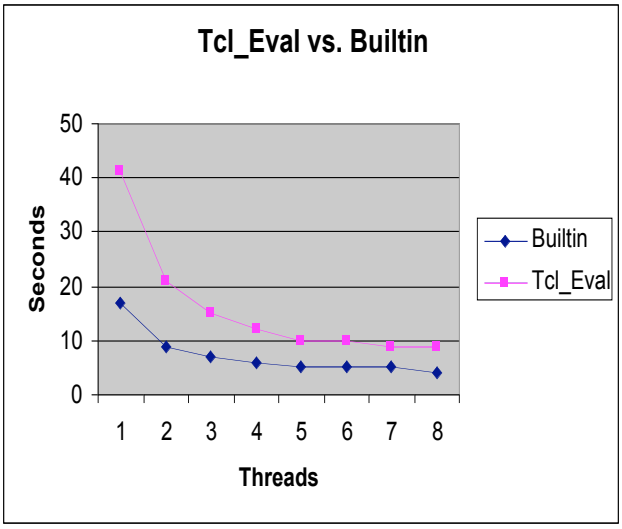
The first step is getting from the high performance, multi-threaded C++ code into the

Tcl interpreter efficiently. What is the difference in performance for various Tcl interfaces? I have choices betlen Tcl_Eval, Tcl_EvalEx, Tcl_EvalObjEx, and Tcl_EvalObjv as alternate calling mechanisms in our quest for ultimate call performance to the user proc.

To start with, I will use the simplest interface, Tcl_Eval. From reading the Tcl documentation, I expect that this will not be a good choice since it causes reinterpretation of the string to a byte code on each invocation and doesn't allow use of the TCL_EVAL_GLOBAL flag. The example code appendix provides excerpts of code from this implementation entitled *Tcl_Eval Example Code*.

Tcl_Eval Performance

Now we get to see our first performance numbers. The test machine we have is an 8 way AMD Athlon server, so I will try runs with 1-8 threads. Let's compare Tcl_Eval against the built-in scripting language. Both implementations are performing the same calculations. The built-in language is using our SUM function and Tcl is using a for loop to cycle through the array data.

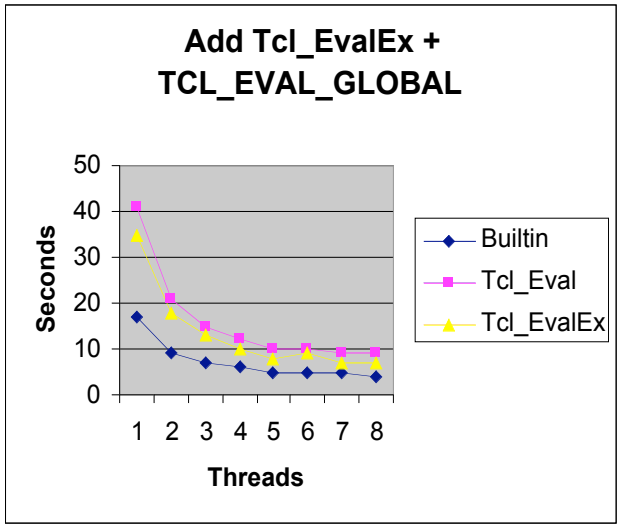


Not bad for a first cut that we know isn't an optimal implementation – our overall system performance with the Tcl script is consistently running a little over 2x the runtime of the built-in scripting language. It also scales similarly with multi-threaded runs. Keep in mind that I have artificially constructed our test case to make the C++ device recognition part run very quickly and easily. In real life, the device recognition code would have to do a lot more work and the 2x difference here would be much smaller.

For the first change to this baseline, I will simply use Tcl_EvalEx with the TCL_EVAL_GLOBAL flag. This will still re-evaluate the script with each execution, but it adds TCL_EVAL_GLOBAL. This is a pretty straight forward two line change to our execution code that is also presented in the code appendix.

Tcl_EvalEx Performance

Here are the performance numbers with the new call:

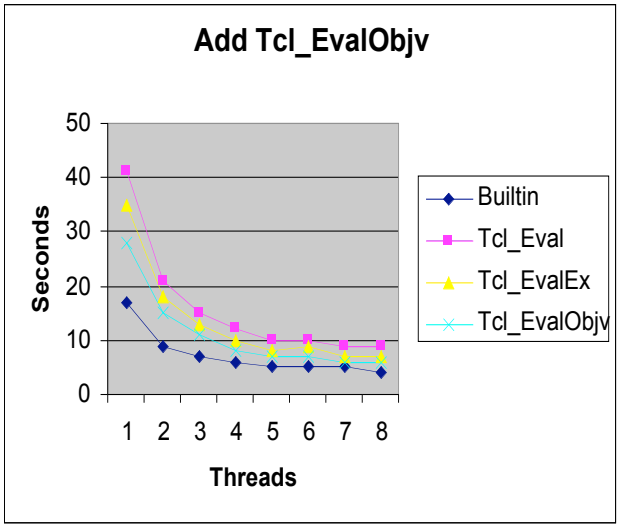


We see that the documentation's advice for using TCL_EVAL_GLOBAL really is worthwhile. It shaves about 15% of from the Tcl_Eval runtimes and brings us a bit closer to the built-in language.

Next I want to enable the byte compiler. Our previous two experiments required string evaluation on the calling string invocation for each of our 1.5 million Tcl proc calls, now I will use Tcl_EvalObjv to turn on Tcl's byte compiler in hopes of cutting the difference between Tcl and the built-in language even more. The code changes a bit more this time, so I will repeat the main parts again in another example in the code appendix entitled *Tcl_EvalObjv Code Example*.

Tcl_EvalObjv Performance

Now, how did we do?



This time, we see a 20% improvement in performance! I did some additional experiments that used pitted Tcl_EvalObjv against Tcl_EvalObjEx, but found that they provide similar performance.

Getting data to Tcl Efficiently

Successful use of Tcl in this environment required efficient transfer of per-device call information to the user's Tcl proc. One key to the performance of this system is in the setup. I create as much of the script calling environment as possible up front. That way, when a device is recognized in the compute intensive code, there isn't much setup to do. Careful avoidance of per device call setup creation for arguments helps to achieve high efficiency calls from C++ to Tcl. I chose a solution in which each argument to the Tcl Proc is represented as a Tcl object command. Each of these object commands has a number of data access commands available for use inside the Tcl code. The argument structures are reused for each device recognized by a particular thread so that there is just a pointer to update before calling Tcl_EvalObjv. This minimizes need for creation argument objects or lists that need to be created for each device specific call. It also allows us to handle the multi-dimensional data that is available from functions like ENC_PER. The code entitled *Tcl_CreateObjCommand* example in the code appendix shows how the command objects are created up front and then how they are accessed during the device specific property call.

Getting data back from C++ Efficiently

Getting data back from the C++ callback efficiently for use inside the Tcl script is also an important consideration in the overall performance of this system. Our Tcl call interface only provides functions that return a double right now, so data is passed back to Tcl through a Tcl_DoubleObj that is set to the value desired and returned as a result object to the interpreter.

I looked at the creation of Tcl_Obj objects that are used to return results from the callback to the Tcl script. I was worried about quickly creating and deleting Tcl_DoubleObj objects used to pass results back. So I tried some experiments with caching the result objects and reusing them (after checking Is_Shared()) if they could be reused.

My experiments, however, proved this caching was not useful. Tcl seems to do a good job of avoiding object allocation thrashing when repeatedly creating and destroying similar objects in repeated callbacks.

Writing Efficient Tcl Code

One area I don't have much control over is how the users will write these extensions. For example, simply removing the {} from the expr commands in our Tcl script resulted in total execution time growing by nearly 300%. I need to make sure our documentation encourages users to write their scripts efficiently.

One way I can encourage users to write efficient code is to provide interfaces for our object that encourage high performance idioms. I suspect that users will generally work with these objects with a for loop similar to the one presented below. Profiler runs show that the code is spending a lot of time in the Tcl interpreter simply running the script. I decided to look at providing some sort of internal iteration and indexing capability as a part of our command object. In our first example, the Tcl code used a for loop:

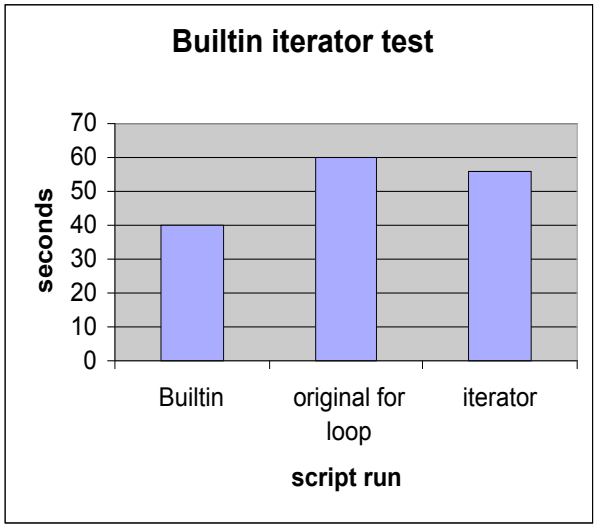
```
set slice_count [ $enc slice_count ]
for { set i 0 } { $i<$slice_count } {
  incr i } {
  # slice specific code here
  # uses index $i to access data
  # from the $enc command object
  # by passing it in as an argument
}
```

Our new object command will try to remove the need for some of the loop bookkeeping and

passing around the index variable $i above. I now use a while loop that looks like this:

```
set i [ $enc first_slice ]
while { $i > 0 }  {
  # slice specific code here
  # index is implicit $enc will
  # access data from the current
  # array slice.
  set i [ $enc next_slice ]
}
```

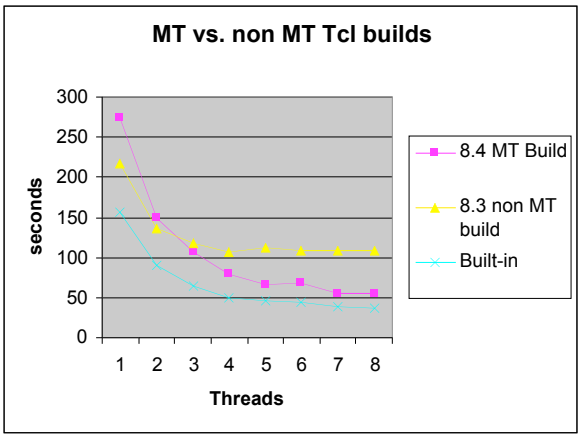
Trying this internal index approach provides some promising results. I ran this test on the second test data set that has more data slices in each test device (thus requiring more indexing when the script runs) and I get the following results:



This very simple change gave us a respectable 6% gain in device recognition performance. It is easy to imagine adding other capabilities to the command object that might result in similar gains. Perhaps we could improve performance even more with some simple variables and expression handling and ... Wait!, that sounds like another interpreted compiler, exactly what we're trying to get out of doing in the first place!

Threading vs. Non-Threading build

The first implementation of this functionality used a non-MT version of Tcl 8.3. As shown in the following performance comparison, the non-MT code runs faster, but limits concurrency. In this non-MT implementation, our device recognizer still runs multi-threaded, but access to the Tcl interpreter is serialized.



Conclusions

Tcl is often used as a top down scripting environment interface. The Device Property calculation engine for the Calibre design verification tool shows that it can also successfully be used to provide a high performance user extension interfaces even in performance critical applications where the Tcl procs will be called millions of times in multi-threaded code. In these situations, it is important to *pull out all the stops* to gain maximum performance when calling the script. This is done by:

- making use of TCL_EVAL_GLOBAL
- making use of the byte compiled interfaces like Tcl_EvalObjv
- taking care of as much call setup as possible before the performance critical repeated calls are made
- providing command object interfaces that allow your command object to carry out data and compute intensive tasks.
- making sure that your Tcl script is written to perform well (remember eval { ... } !).

Running this device calculation on a real chip design using 8 CPUs only resulted in no increase in device recognition time.

Appendix

Test environments used for experiments -

AMD Athlon machine
jmachine info:
OS system:
Linux
OS release:
2.6.9-11.ELsmp
OS version:
#1 SMP Fri May 20 18:25:30 EDT 2005
CPU:
4300
CPU Speed:
2194 MHz Athlon
Number of CPUs: 8
Vendor Release Version:
RedHat Enterprise 4 Update 1
Memory: 63927 Mbytes
Tcl Version: 8.4 – MT build
Tcl Version: 8.3 – non MT build