

VCS optimization techniques for Multi-Chip simulations

Debashis Biswas

CISCO
Bangalore, India

www.cisco.com

ABSTRACT

Discovering system performance issues late in the product development cycle can be catastrophic to project schedules and product competitiveness, causing failure in the market. To improve the overall project schedule, system-level simulation and performance analysis must be done earlier in the product design cycle to accelerate innovation. System-level simulation involving multiple chips has its own challenges and pushes the tool's capabilities. The daunting complexity of design containing multiple chips with huge functionality results in a database size that takes several gigabytes of memory, and takes long hours to perform any operation. This necessitates using systems with big memory and using different VCS optimization techniques to improve the compile and run time performance. To address these issues we used some of the optimization techniques natively present in VCS. Each technique is described in detail in this paper along with simulation results that we got in our system-level simulations.

Table of Contents

| | |
|--|----|
| 1. Introduction | 4 |
| 2. Challenges with Multi-Chip simulations | 4 |
| 3. VCS techniques to improve COMPILE TIME performance | 7 |
| 3.1 Parallel compilation | |
| 3.2 Fast compilation | |
| 3.3 Incremental compilation | |
| 3.4 Partition compile | |
| 4. VCS techniques to improve RUN TIME performance | 25 |
| 4.1 Radiant technology | |
| 4.2 Save & Restore | |
| 4.3 Interactive rewind | |
| 4.4 VCS Multicore Technology | |
| 4.5 Dump scope control | |
| 5. VCS profiler utility to find compilation and simulation bottlenecks..... | 36 |
| 5.1 <i>Signal duming was consuming lot of simulation time & machine memory</i> | |
| 5.2 <i>A performance bug in our generic memory model was exposed</i> | |
| 6. VCS 3-step flow | 38 |
| 7. Results | 40 |
| 8. Conclusions and Looking Forward | 42 |
| 9. Acknowledgements | 43 |
| 10. References | 44 |

Table of Figures

| | |
|--|----|
| Figure 1 - System Config 1 (SC1) | 5 |
| Figure 2 - System Config 2 (SC2) | 5 |
| Figure 3 - System Config 3 (SC3) | 6 |
| Figure 4 - The Architecture of Incremental Compilation in VCS | 12 |
| Figure 5 - Partition compile Example Module hierarchy | 16 |
| Figure 6 - Partition Compile v2k configuration file | 17 |
| Figure 7 - PARTITION TIME – Compile time comparison (in mins) in System Config 1 | 18 |
| Figure 8 - PARTITION TIME – Compile time comparison (in mins) in System Config 2 | 19 |
| Figure 9 - PARTITION TIME – Compile time comparison (in mins) in System Config 3 | 20 |
| Figure 10 - Disk space savings through sharing of partitions..... | 22 |
| Figure 11 - Typical Simulation flow..... | 27 |

| | |
|---|----|
| Figure 12 - Traditional simulation flow | 28 |
| Figure 13 - Save & Restore (How does it help??) | 29 |
| Figure 14- SAVE & RESTORE – Simulation time comparison | 31 |
| Figure 15 - Interactive Rewind (An illustration) | 32 |
| Figure 16 - Simprofile time report | 37 |
| Figure 17 - Simprofile mem report | 37 |
| Figure 18 - Simprofile mem report (Time Module View) | 38 |
| Figure 19 - Comparison of Total TAT in the optimized flow vs. bare-bone approach (without any optimizations) →1st ITERATION | 41 |
| Figure 20 - Comparison of Total TAT in the optimized flow vs. bare-bone approach (without any optimizations) →2 nd ITERATION | 41 |

Table of Tables

| | |
|---|----|
| Table 1 - Improvement in compilation time using Parallel compilation | 9 |
| Table 2 - Compile time reduction using <i>-fastcomp=0</i> | 10 |
| Table 3 - Compile time reduction using <i>-fastcomp=1</i> | 10 |
| Table 4 - Compile time reduction using Incremental compilation | 14 |
| Table 5 - PARTITION TIME – Compile time comparison in System Config 1 | 18 |
| Table 6 - PARTITION TIME – Compile time comparison in System Config 2..... | 19 |
| Table 7 - PARTITION TIME – Compile time comparison in System Config 3..... | 20 |
| Table 8 - Disk space savings through sharing of partitions in Partition Compile | 21 |
| Table 9 - Improvement in Simulation time using radiant technology | 26 |
| Table 10 - SAVE & RESTORE – Run time comparison | 30 |
| Table 11 - Improvement in simulation time (when ALP is turned ON for VPD dump) | 35 |
| Table 12 - Saving in Disk Space and Simulation time using Dump scope control | 36 |
| Table 13 - Reduction in TAT using the optimized flow (1st ITERATION) | 42 |
| Table 14 - Reduction in TAT using the optimized flow (2nd ITERATION)..... | 42 |

1. Introduction

As the overwhelming appetite for network bandwidth continues the complexity of networking ASICs is increasing day by day. Because of this the port density and port speed on the chip is increasing. Cisco being the world leader in Networking has always been on the forefront to address this ever-increasing appetite for bandwidth. As a part of this initiative we are designing ASICs that will power the next generation of enterprise, campus, service provider, storage networks and data center switches. These ASICs will support Unified Access, bring-your-own-device (BYOD) trend, mobility, and the Internet of Things (IoT) “*changing the Way We Live, Work, Play and Learn.*”

These ASICs are very complex with a single ASIC having the capability to perform the complete switching function. Winning in the market place requires system development teams to put better product to the market ahead of competition. At the same time cost of the product needs to be reduced. Integration of more and more functionality into a single chip is becoming very important to achieve this objective. One of the key enablers in reducing time to market is to ensure that ASICs used in the System are designed on-time and integration and interoperability of these ASICs in the system as a whole is seamless so that the product is delivered on time.

Simulating multiple chips together helps in validating that the current chip will work with any legacy chips already developed and tested by the team. Also it helps in validating any assumptions made between the chips. Each ASIC may work on its own, but may not be connected consistently with the system specs at the higher level. So the focus of system simulation should be on the operation of the system as a whole, rather than the individual ASICs.

A complete system-level test for a system design will include many milliseconds of simulation time and millions of lines of code with a database size of several gigabytes of memory. This will involve its own challenges in terms of getting the whole design compiled together and simulated in a reasonable amount of time.

In this paper we present our experience with some of the challenges faced while developing a complete SystemDV infrastructure to simulate multiple ASICs together to mimic their behaviour in the complete system as a whole. Using partition compile, incremental compile, save & restore and other VCS optimization techniques helped us in utilizing our available engineering, VCS licenses and LSF resources in an efficient and optimal way *without requiring investment in any additional hardware or software resources (in terms of license cost)*. The results obtained from all these techniques are described in detail along with the detailed description of how to use them in any verification environment.

2. Challenges with Multi-Chip simulations

Here is the summary of the ASICs being designed for our next-generation products.

ASIC X (design in progress)

- This is a 28M+ multiport Ethernet switch ASIC, which includes the MACs, forwarding logic, buffering and queuing logic/memory.

ASIC Y (design in progress)

- This is a 59M+ multiport Ethernet switch ASIC, which includes the MACs, forwarding logic, buffering and queuing logic/memory.

ASIC Z (legacy chip, already working in field)

- This is a 5M+ IEEE 802.1ae MacSec enabled Ethernet network ports aggregation chip. This has to interoperate with ASIC X & Y in the new systems.

Different systems are planned using a combination of these ASICs. Three of them for which verification is in progress (with the SystemDV infrastructure that we developed) are as follows-

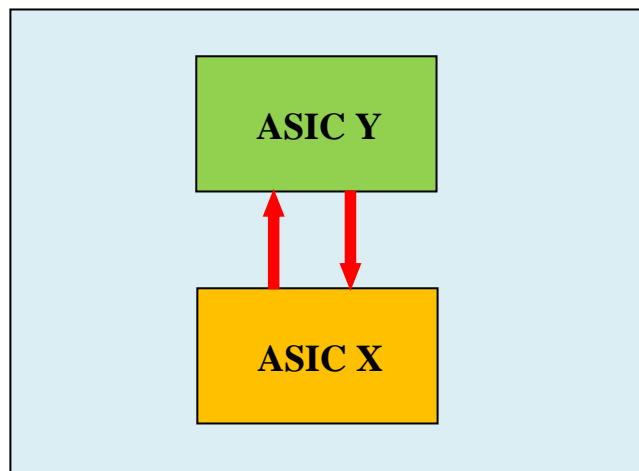


Figure 1. System Config 1 (SC1)

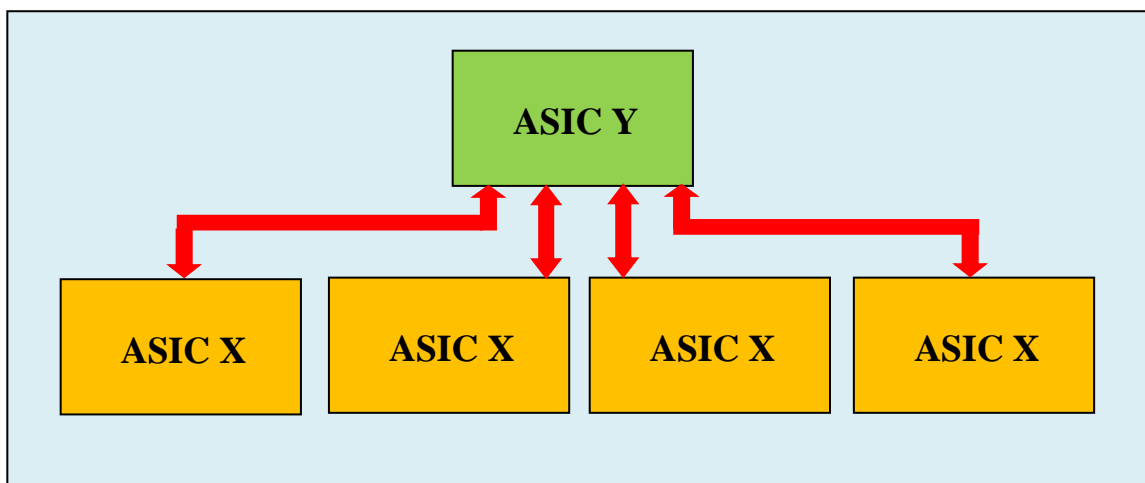


Figure 2. System Config 2 (SC2)

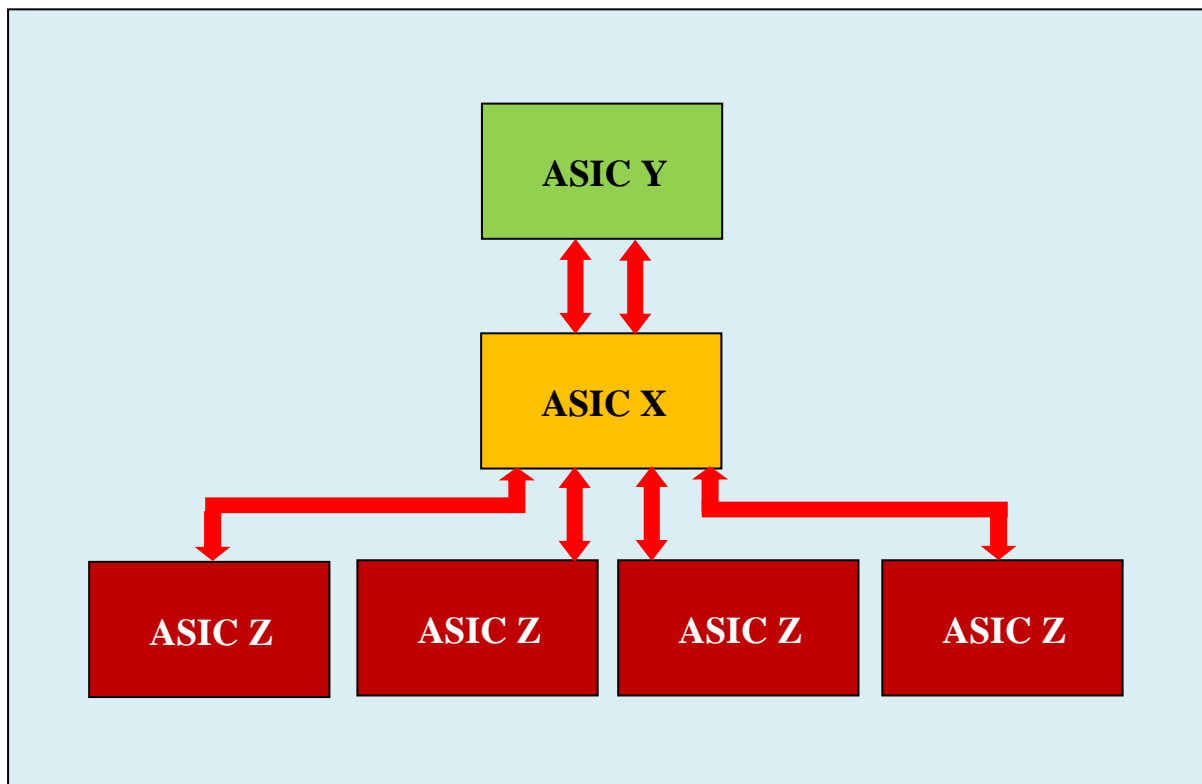


Figure 3. System Config 3 (SC3)

The challenge is to compile all the files (of individual ASICs) together, with different vendor technology files, and duplicate module definitions. Needless to say, simulation is one of the key concerns since gate count can get typically very high ~250-300 million, which means that compile time and run time is going to be a big bottleneck. Also duplicate module definitions (same module name getting used in different ASICs and their module definitions being different) is going to give module name conflicts.

These challenges forced us to explore new switches and optimization techniques in VCS which can

- Give us better insight into our Testbench by identifying simulation bottlenecks (VCS – *simprofile* switch)
- Give us better turnaround time in case of very small incremental changes done during the Testbench development cycle (incremental compile, partition compile etc.)
- Give us better run time performance while running regressions by avoiding reset sequence and some common init routines which are common across many simulations (VCS save & restore, interactive rewind etc.)
- Avoid module name conflicts by following a methodology which logically separates all the ASICs into different namespaces during the compilation process (VCS 3-step flow)

Why not FPGA prototyping and Emulation??

When the question of System validation comes obviously the question comes as to why we are not using FPGA prototyping and emulation.

In our case, we had a handful of scenarios to be tested at system level, so using FPGA prototyping for system level validation could not be justified. Also couple of these testcases were performance related for which FPGA prototyping was definitely not a choice because of core frequency limitations in FPGAs (Virtex 7 max core frequency is ~500MHz). In addition to this there are some additional considerations to be kept in mind while using FPGA prototyping and emulation.

- FPGA based prototypes are fast, but involves a lot of development time in porting both the design and testbench.
- HVL based testbench can't be directly ported to FPGA. It needs to be synthesizable.
- Debugging in FPGA is difficult, because of limited visibility (Test pin multiplexing, Chipscope tool in Xilinx, Signal Tap Analyzer in Altera are some workarounds but still gives very limited visibility and uses the existing logic and memory resources in FPGA).
- FPGA based prototypes are expensive.
- Performance testing can't be done in FPGA. FPGA prototyping is plagued by the “Three Laws of prototyping” (*Courtesy : Doug Amos in his book FPMM*).
 - SoCs are larger than FPGAs.
 - SoCs are *faster* than FPGAs.
 - SoC designs are FPGA-hostile.

Since we were leveraging the optimization techniques natively present in VCS the turnaround time of the SystemDV infrastructure development was drastically reduced. The whole infrastructure was developed within 2 months' time frame starting from scratch *without requiring investment in any additional hardware or software resources (in terms of license cost)*.

3. VCS techniques to improve COMPILE TIME performance

Compile-time performance plays a very important role when we are in the *initial phase of our design* development cycle. In this phase, we may want to modify and recompile the design to observe the behavior. Since, this phase involves a lot many recompiling cycles, achieving a faster compilation time is important.

In our setup, the testbench was having several lines of code changes (going from one run to other) during initial environment bringup. This was affecting the productivity of our verification engineers during the testbench development phase and initial bringup of our SystemDV environment. Multiple iterations of simulations have to go through compilation again even for small changes and each compilation was taking around 1hour. Any reduction in the compile time here was a definite gain for us.

3.1 PARALLEL COMPILATION

You can improve the compile-time performance by specifying the number of parallel processes VCS MX can launch for the native code generation phase of the elaboration. You should specify this using the compile-time option `-j[no_of_processes]`, as shown below:

```
% vcs -j[no_of_processes] [options] top_entity/module/config
```

Note:

Parallel compilation applies only for the Verilog portion of the design.

For example, the following command line will fork off two parallel processes to generate a binary executable:

```
% vcs -j2 top
```

This is an infrastructure based optimization.

**Multi core CPU is the requirement. Ideally,
Number of cores in CPU = No. of threads**

How VCS implements PARALLEL COMPILATION?

Simulator algorithms are designed to compile the individual segments of the design on different cores of the CPU, thus speeding up compile time.

What is the optimum choice of “N” in “-jN”

VCS does not limit the number of cores the technology can be run on. But the optimum choice of “N” is dictated by following two factors-

- 1) VCS resorts to normal compilation mode when it cannot find semaphore available i.e. when the operating system on your machine hits the semaphore limitation. In that case VCS will simply ignore the "-j" option with the following warning message about semaphore limitation:

```
-----  
Warning-[VCS_DISPARAL] Parallel code-gen disabled  
VCS disabled parallel code generation(-j) due to semaphore limitation in  
current system.  
Will continue the compilation with normal mode  
-----
```

- 2) The parallel compile processes with "-j" option does not work if the machine does not have enough memory to support all the processes. This can be due to smaller machine or if there are too many jobs running on this machine. In that case, VCS will put the following warning in the VCS compilation log:

```
-----  
Warning-[PCNCMP2] Cannot create more process  
Insufficient storage space to create more process for parallel compile. Only  
processes were forked for parallel compile, while processes were requested to '-j'.  
VCS will continue to compile with processes including the  
main process.  
-----
```

Memory usage overhead for parallel compilation in vcs with -j switch

Memory usage overhead is to be expected as N processes will be forked off during code generation stage of vcs compilation.

Currently, the overhead is N times as “N” vcs processes will be forked off and each process has the same memory foot print as the original process. This may be an issue if the design is quite large and the total memory usage of the N processes exceeds the virtual memory space on the machine or even the operating system limit causing the simulation to abort.

“-j4” was an optimum choice for our design

-j4 was an optimum choice for our design, -j8 and above was causing simulation crash since our design was quite large and the total memory usage was shooting up linearly (even though we were using 8-core/16-core CPU).

Table 1. shows improvement seen using parallel compilation for the three system configurations.

| System Configuration | Compilation time | | Reduction in compile time |
|----------------------|------------------------------|---|---------------------------|
| | Without parallel compilation | With parallel compilation (-j4 on a 4-Core CPU) | |
| SC 1 | 50 mins | 45 mins | 10% |
| SC 2 | 160 mins | 145 mins | 9.38% |
| SC 3 | 180 mins | 160 mins | 11.11% |

Table 1. Improvement in compilation time using Parallel compilation

CONCLUSION

Overall we are getting ~10% reduction in our setup using parallel compilation

3.2 FAST COMPILATION

VCS has new compile-time performance optimizations called Fast Compilation that one can use to reduce compile-time for your design (and therefore overall turnaround time). It can be enabled by using the *-fastcomp* compile-time option.

There are two levels of Fast Compilation, specified by the *-fastcomp=0* and *-fastcomp=1* compile-time options.

-fastcomp=0 is the same as ***-fastcomp*** (without the argument) and is the generally preferred option.

-fastcomp=1 applies more aggressive compile-time performance optimizations.

How VCS reduces compilation time in Fast compilation?

Fast compilation reduces the elaboration time, by moving some of the simulator operations to runtime.

Table 2. shows improvement seen using ***-fastcomp=0*** for the three system configurations.

| System Configuration | Compilation time | | Reduction in compile time |
|----------------------|-------------------------|-----------------|---------------------------|
| | Without fastcomp switch | With fastcomp=0 | |
| SC 1 | 45 mins | 42 mins | 6.67% |
| SC 2 | 145 mins | 135 mins | 6.89% |
| SC 3 | 160 mins | 150 mins | 6.25% |

Table 2. Compile time reduction using *-fastcomp=0*

Table 3. shows improvement seen using ***-fastcomp=1*** for the three system configurations.

| System Configuration | Compilation time | | Reduction in compile time |
|----------------------|-------------------------|-----------------|---------------------------|
| | Without fastcomp switch | With fastcomp=1 | |
| SC 1 | 45 mins | 40 mins | 11.11% |
| SC 2 | 145 mins | 130 mins | 10.34% |
| SC 3 | 160 mins | 145 mins | 9.38% |

Table 3. Compile time reduction using *-fastcomp=1*

CONCLUSION

Overall we are getting ~6-10% reduction in our setup using the two switches for fast compilation

WORD OF CAUTION!!!

Reduction in compile time may incur 5-7% increase in run-time in some designs...

3.3 INCREMENTAL COMPILATION

During elaboration, VCS MX builds the design hierarchy. By default, when you recompile the design, VCS MX compiles only those design units that have changed since the last elaboration. This is called incremental compilation.

The incremental compilation feature is the default in VCS MX. It triggers recompilation of design units under the following conditions:

- Changes in the command-line options.
- Change in the target of a hierarchical reference.
- Change in the ports of a design unit.
- Change in the functional behavior of the design.
- Change in a compile-time constant such as a parameter/generic.

The following conditions do not cause VCS MX to recompile a module:

- Change of time stamp of any source file.
- Change in file name or grouping of modules in any source file.
- Unrelated change in the same source file.
- Nonfunctional changes such as comments or white space.

3.3.1 The Architecture of Incremental Compilation in VCS

Figure 4. gives an overview of the VCS architecture, with focus on incremental compilation

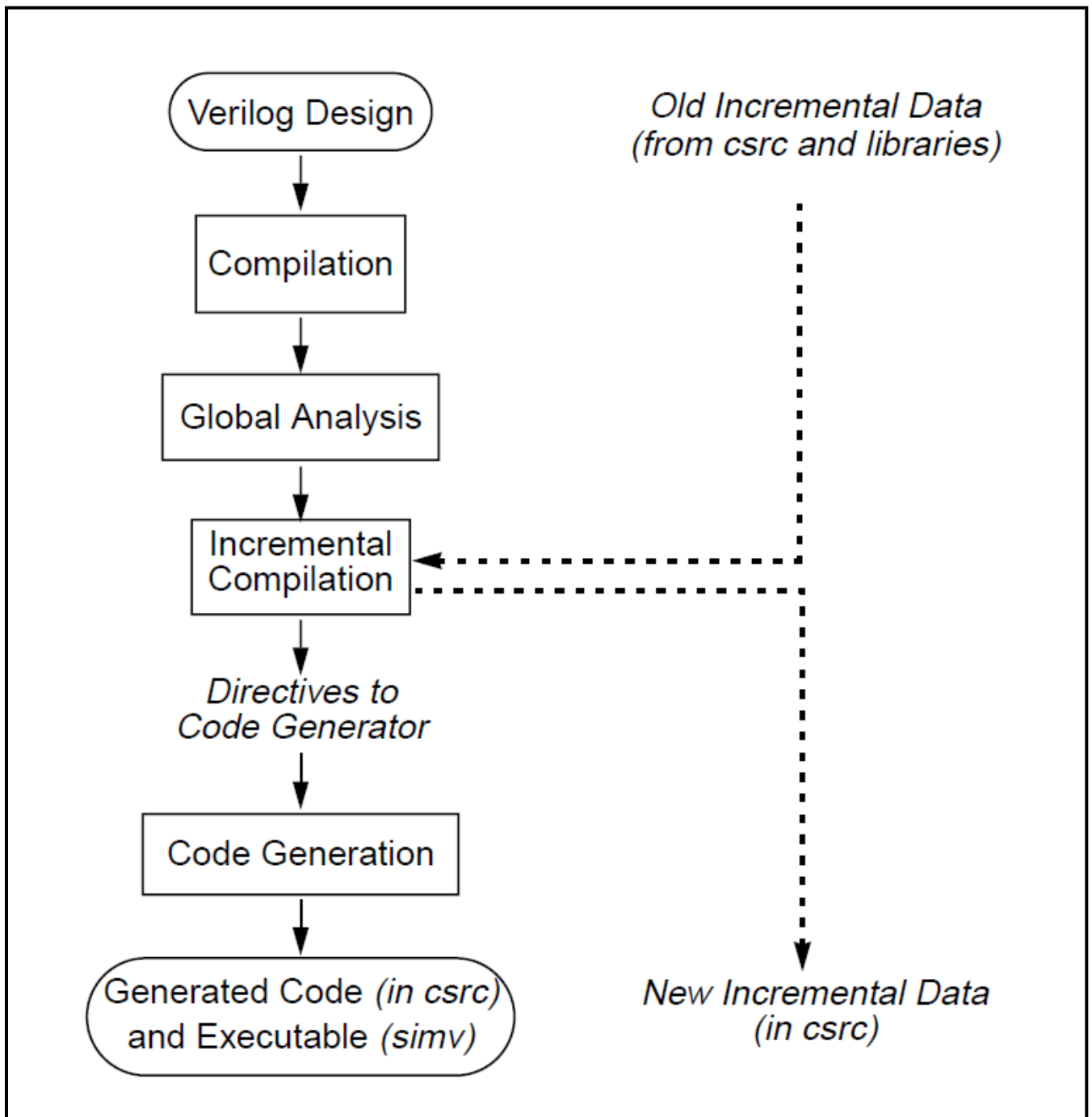


Figure 4. The Architecture of Incremental Compilation in VCS

- VCS first performs a preliminary compilation on the Verilog source.
- Based on this, a Global Analysis (GA) of the design is conducted to resolve inter-module dependencies, and to perform global optimizations.
- At the end of GA, the Incremental Compilation phase analyzes incremental-specific data from the previous run.
- First, relevant token changes in the Verilog source are detected. Next, inter-module dependencies (determined through GA) are used to propagate the effect of the changes to other modules in the design hierarchy.
- This results in exact determination of the modules for which code needs to be re-generated in the current run.

Inter-module dependencies

When a module is modified, it naturally gets re-compiled. Certain changes to modules can also cause other modules in the design to be re-compiled. The following three types of inter-module interactions can cause this:

- ***Parent-child port relationships:*** A change in the port specification can cause re-compilation of both the parent and child modules.
- ***Hierarchical references:*** A new hierarchical reference in a module will cause both the referer and referee modules to be re-compiled.
- ***Module inlining:*** In certain situations, VCS performs an optimization that inlines child instantiations into the parent's body. This introduces a special type of inter-module interaction that affects incremental compilation. This optimization is transparent to the user. But since such inlining alters the design hierarchy for code generation, re-compilation decisions for an inlined child module affect those of the non-inlined ancestor module.

Global dependencies

Several global dependencies also come into play in influencing the re-compilation decisions of incremental compilation.

- ***Command-line options:*** Changes in command-line options can result in changes to generated code. For instance, global flags such as ‘+nospecify’ will affect a whole suite of modules sharing the relevant property (the occurrence of ‘specify’ blocks, in this case).
- ***Compiler directives:*** Altering a compiler directive can affect the code generated for several modules. For instance, changing the *timescale/precision* for the design will affect all modules within the scope of that change. Incremental compilation has to handle such changes.
- ***Platform dependence:*** VCS supports Verilog simulation on many platforms (such as SunOS, Solaris, HP, NT, *etc.*). Moving between platforms will naturally result in re-compilation of the entire design.
- ***PLI/ACC/Debug specifications:*** VCS allows the PLI/ACC capabilities of individual modules to be specified through a ‘.tab’ file. If these capabilities are altered from run to run, this will necessitate re-compilation of the concerned modules.

3.3.2 Options for Incremental Compilation

-Mdirectory=directory

Specifies the incremental compile directory. The default name for this directory is *csrc*, and its default location is your current directory. You can substitute the shorter *-Mdir* for *-Mdirectory*.

-Mlib=dir

This option provides VCS MX with a central place to look for the descriptor information before it compiles a module and a central place to get the object files when it links together the executable. This option allows you to use the parts of a design that have been already tested and debugged by other members of your team without recompiling the modules for these parts of the design.

You can specify more than one place for VCS MX to look for descriptor information and object files by providing multiple arguments with this option.

Example:

```
vcs design.v -Mlib=/design/dir1 -Mlib=/design/dir2
```

Or, you can specify more than one directory with this option, using a colon (:) as a delimiter between them, as shown below:

```
vcs design.v -Mlib=/design/dir1:/design/dir2
```

Table 4. shows improvement seen using incremental compilation for the three system configurations.

| System Configuration | Compilation time | | Reduction in compile time |
|----------------------|------------------|---------------------|---------------------------|
| | Scratch compile | Incremental compile | |
| SC 1 | 45 mins | 28 mins | 37.78% |
| SC 2 | 145 mins | 90 mins | 37.93% |
| SC 3 | 160 mins | 105 mins | 34.38% |

Table 4. Compile time reduction using Incremental compilation

CONCLUSION

Overall we are getting 30-35% reduction in our setup using incremental compilation

3.4 PARTITION COMPILE

Partition Compile is a VCS feature that allows you to compile portions of the design and get significantly faster turnaround time during the iterative process of compile and recompile.

The Partition Compile feature requires the VCS MX installation

Partition Compile is enabled with the *-partcomp* compile-time option.

With Partition Compile one specifies partitions in your design that one expects to revise and recompile often. VCS MX recompiles only the modified partitions.

These partitions can be specified in the following ways:

- Using Autopartitioning
- Specifying the partitions manually

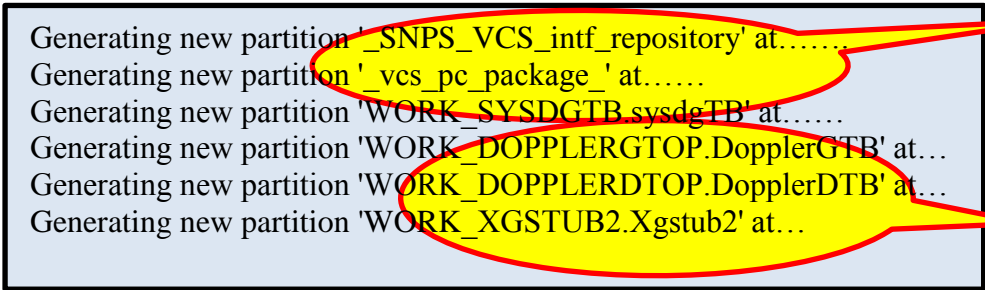
Using Autopartitioning

It is recommended to use autopartitioning, particularly when using Partition Compile for the first time on your design and then try to increase performance.

Based on our experience usually the Synopsys created auto-partitions are good to go in most of the cases. In our case we were using VCS 3-step flow and analyzing our three designs ASIC X,Y and Z into three different work directories. VCS created six partitions for our design.

Please note that in addition to the partitions you specify, VCS MX creates internal or default partitions for the following parts of the design:

- a partition for the the top-level module
- a partition for all SystemVerilog packages
- a partition for all SystemVerilog interfaces



Generating new partition '_SNPS_VCS_intf_repository' at.....
Generating new partition '_vcs_pc_package_' at.....
Generating new partition 'WORK_SYSDGTB.sysdgTB' at.....
Generating new partition 'WORK_DOPPLERGTOP.DopplerGTB' at...
Generating new partition 'WORK_DOPPLERDTP.DopplerDTB' at...
Generating new partition 'WORK_XGSTUB2.Xgstub2' at...

The screenshot shows a list of partitions generated by VCS. Two yellow callout boxes with red borders point to specific lines. The first callout points to the first three lines, and the second callout points to the last three lines.

Three separate partitions created for
(a) Top-level module
(b) SV packages
(c) SV interfaces

Separate partitions created for
(a) ASIC X
(b) ASIC Y
(c) ASIC Z
which were analysed separately in VLOGAN

Using autopartdbg option along with -partcomp (*-partcomp=autopartdbg*) creates the *vcs_partition_config.file* file, which contains the design partitioning information. Once the partition config file is generated, it can be modified to add/delete any partitions based on user requirements. This partition config file can then be passed at the vcs step to pick the modified partitioning.

When in doubt we can profile our partition compilation as explained in the section **3.4.3 Profiling Partition Compilation**.

Specifying the partitions manually in one of the following ways:

- In an +optconfigfile configuration file (for Verilog/SV designs)
- In a V2K or SV configuration (for Verilog/SV/VHDL/SystemC and MX designs)

The Partition Compile use model is very similar to the existing VCS MX regular compile use model. No changes are required to the source code of the design and testbench when migrating to Partition Compile and runtime performance is not compromised when compared to the regular VCS MX compilation use model.

With Partition Compile, you designate separate partitions for various parts of the design and testbench, making sure to designate a partition or partitions for the part that you frequently revise and compile. You then see faster turnaround times from recompiling only some partitions and not needing to recompile other partitions.

For the 1st iteration there is some overhead because some time is spent in creating the partitionlib database.

Specifying partitions using v2k configuration file

Let's consider the following Example hierarchy

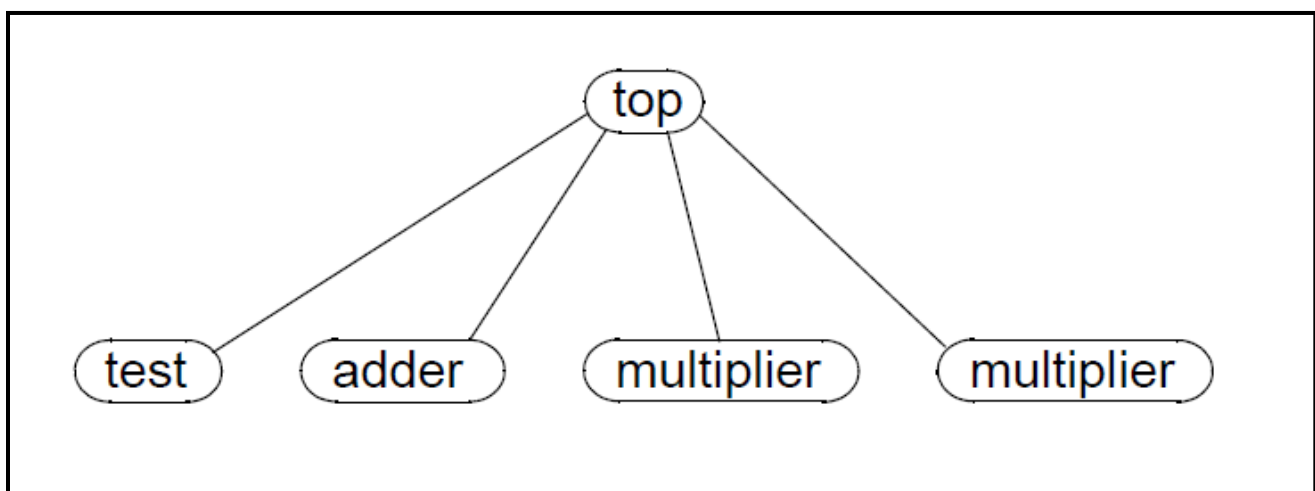


Figure 5. Partition compile Example Module hierarchy


```

// topcfg.v
config topcfg;
design top; // top-level module
partition instance top.t1; // partition for program test
instance top.m1 use multiplier;
instance top.m2 use multiplier;
partition instance top.m1; // partition for multiplier instance m1
partition instance top.m2; // partition for multiplier instance m2
endconfig

```

Figure 6. Partition Compile v2k configuration file

The above configuration specifies:

- one partition for program block test
- one partition for the m1 instance
- one partition for the m2 instance.

The default and unspecified partition contains the top-level module top and module adder.

Commands for Partition Compile

Add the *topcfg.v* file and *-top config_name* option to the vcs command line, for example:

```
% vcs -partcomp -top topcfg topcfg.v top.v test.v add_mult.v [other options]
```

Partition compile Results in our SystemDV environment for the three System Configs

| Sytem Config1 (1ASIC X+1ASIC Y) | Compilation time | | Improvement in compilation time with partition compile |
|------------------------------------|------------------------------|---------------------------|---|
| | Without partition compile | With partition compile | |
| Fresh compilation | 45 mins | 65 mins | -44.44% |
| Small change in testcase | 45 mins | 3 mins | 93.33% |
| Small change in TB | 45 mins | 5 mins | 88.89% |
| Small change in ASIC X | 45 mins | 10 mins | 77.78% |
| Small change in ASIC Y | 45 mins | 12 mins | 73.33% |

Table 5. PARTITION TIME – Compile time comparison in System Config 1

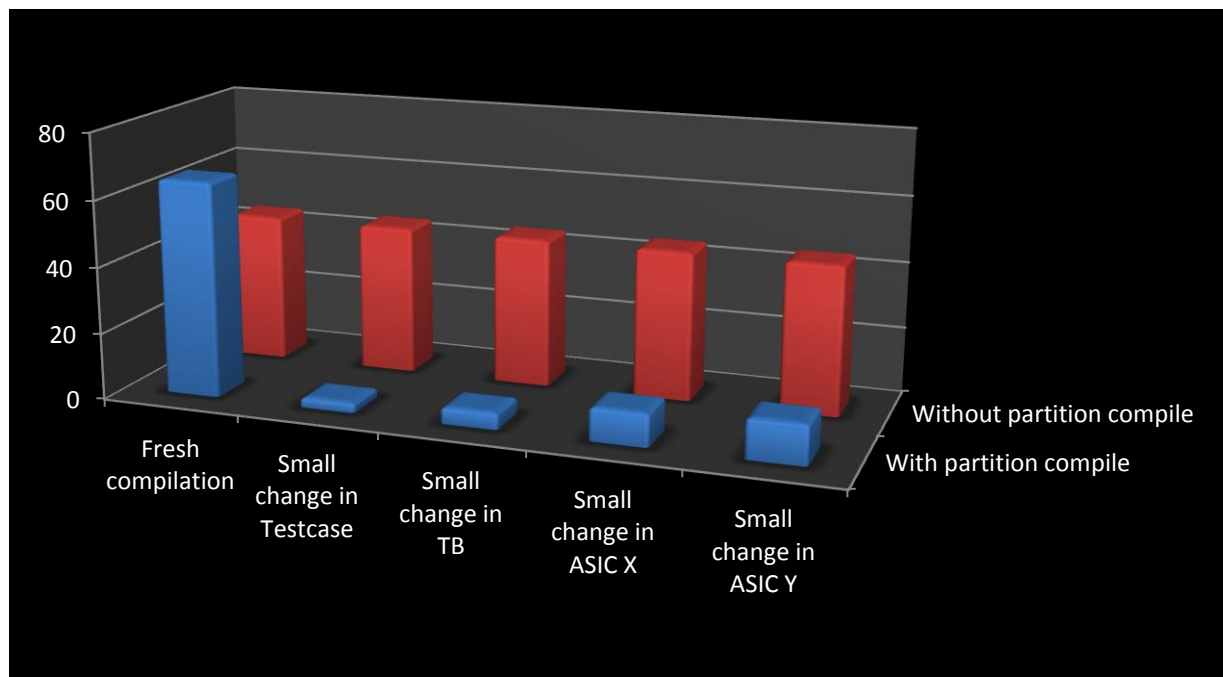


Figure 7. PARTITION TIME – Compile time comparison (in mins) in System Config 1

| System Config2 (4 ASIC X+1ASIC Y) | Compilation time | | Improvement in compilation time with partition compile |
|--------------------------------------|------------------------------|---------------------------|---|
| | Without partition compile | With partition compile | |
| Fresh compilation | 145 mins | 200 mins | -37.93% |
| Small change in testcase | 145 mins | 8 mins | 94.48% |
| Small change in TB | 145 mins | 10 mins | 93.10% |
| Small change in ASIC X | 145 mins | 12 mins | 91.72% |
| Small change in ASIC Y | 145 mins | 15 mins | 89.66% |

Table 6. PARTITION TIME – Compile time comparison in System Config 2

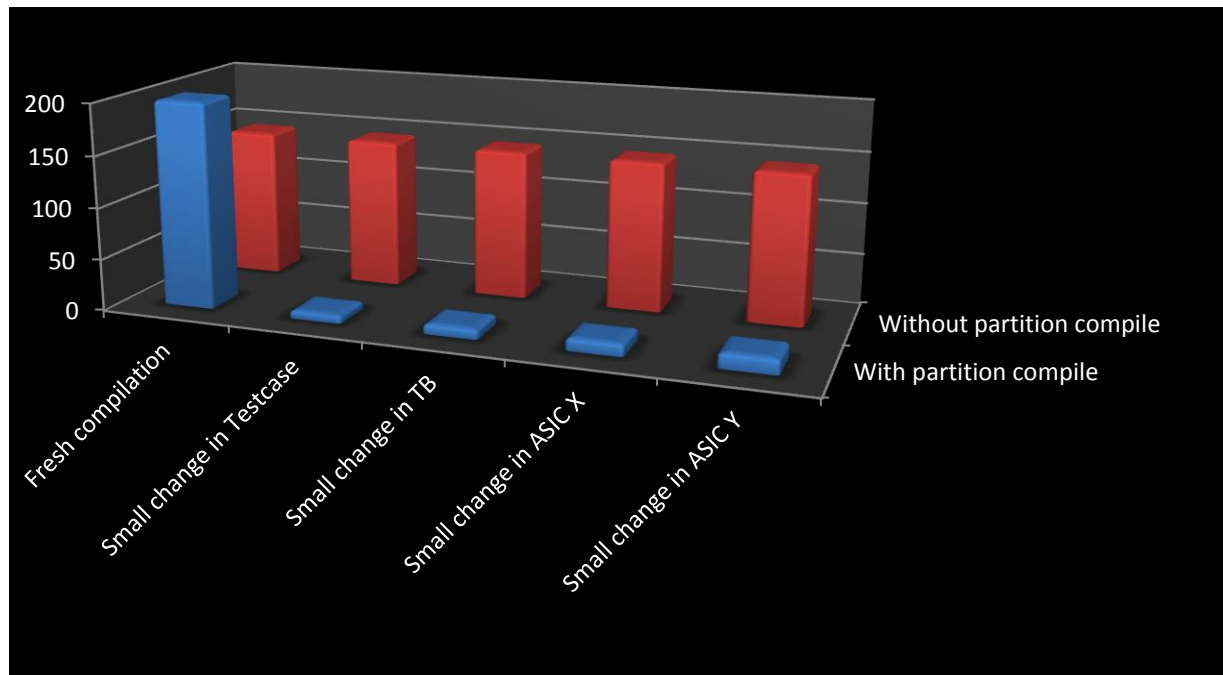


Figure 8. PARTITION TIME – Compile time comparison (in mins) in System Config 2

| System Config3 (4 ASIC Z + 4 ASIC X+1ASIC Y) | Compilation time | | Improvement in compilation time with partition compile |
|---|---------------------------|------------------------|--|
| | Without partition compile | With partition compile | |
| Fresh compilation | 160 mins | 220 mins | -37.5% |
| Small change in testcase | 160 mins | 8 mins | 95% |
| Small change in TB | 160 mins | 10 mins | 93.75% |
| Small change in ASIC X | 160 mins | 12 mins | 92.5% |
| Small change in ASIC Y | 160 mins | 15 mins | 90.63% |
| Small change in ASIC Z | 160 mins | 10 mins | 93.75% |

Table 7. PARTITION TIME – Compile time comparison in System Config 3

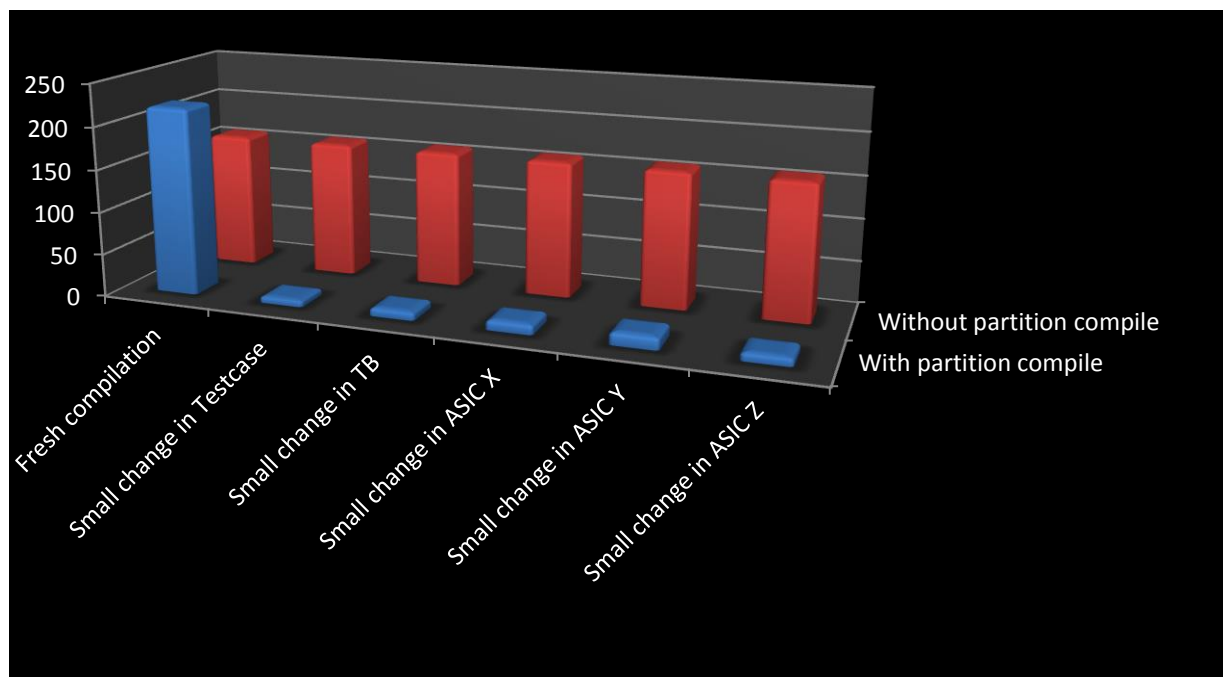


Figure 9. PARTITION TIME – Compile time comparison (in mins) in System Config 3

Note- Partition Compile has slight runtime penalty (~5% observed in our design)

3.4.1 Big disk space savings through sharing of partitions

Different tests can share same DUT partitions. This can save disk space by sharing compilation data. This can be done by generating the partitions by running the first test and then reusing the common partition data for the remaining tests running in parallel with the `-partcomp_sharedlib=dir_path` option.

Use Model illustrated

The vcs command line for test1 is:

```
% vcs -partcomp top.v test.v add_mult.v \  
-partcomp_dir=./PARTCOMP_test1 [other options]
```

The command line for test2 is:

```
% vcs -partcomp top.v test.v add_mult.v \  
-partcomp_dir=./PARTCOMP_test2 \  
-partcomp_sharedlib=./PARTCOMP_test1 [other options]
```

VCS MX reuses the common DUT partitions from the PARTCOMP_test1 directory and issues messages that tell you the partitions being reused in the current compilation.

Use the option `-partcomp_sharedlib=<dir_path>` to refer to shared partition directory

Use the `-partcomp_dir=<dir_path>` option to specify a directory for the partition data.

| Components | Partition Compile | Single Compile | Comparison |
|------------------|-------------------|----------------|--|
| simv | 6.6M | 525M | simv and crsc in partition compile is very small when compared to single compile |
| simv.daidir | 625M | 547M | |
| crsc | 9.6M | 1.1G | |
| Local partition | 20M | 0 | Partition database reused for all 25 testcases in partition compile |
| Global Partition | 2.8G | 0 | |

Table 8. Disk space savings through sharing of partitions in Partition Compile

We had 25 testcases in our testplan.

Disk space consumption (with partition compile)

$2.8G$ (Global partition) + $25 * \{6.6M$ (simv) + $625M$ (simv.daidir) + $9.6M$ (crsc) + $20M$ (local partition) $\} = 19.33G$

Disk space consumption (with single compile)

$25 * \{525M$ (simv) + $547M$ (simv.daidir) + 1.1 (crsc) $\} = 54.3G$

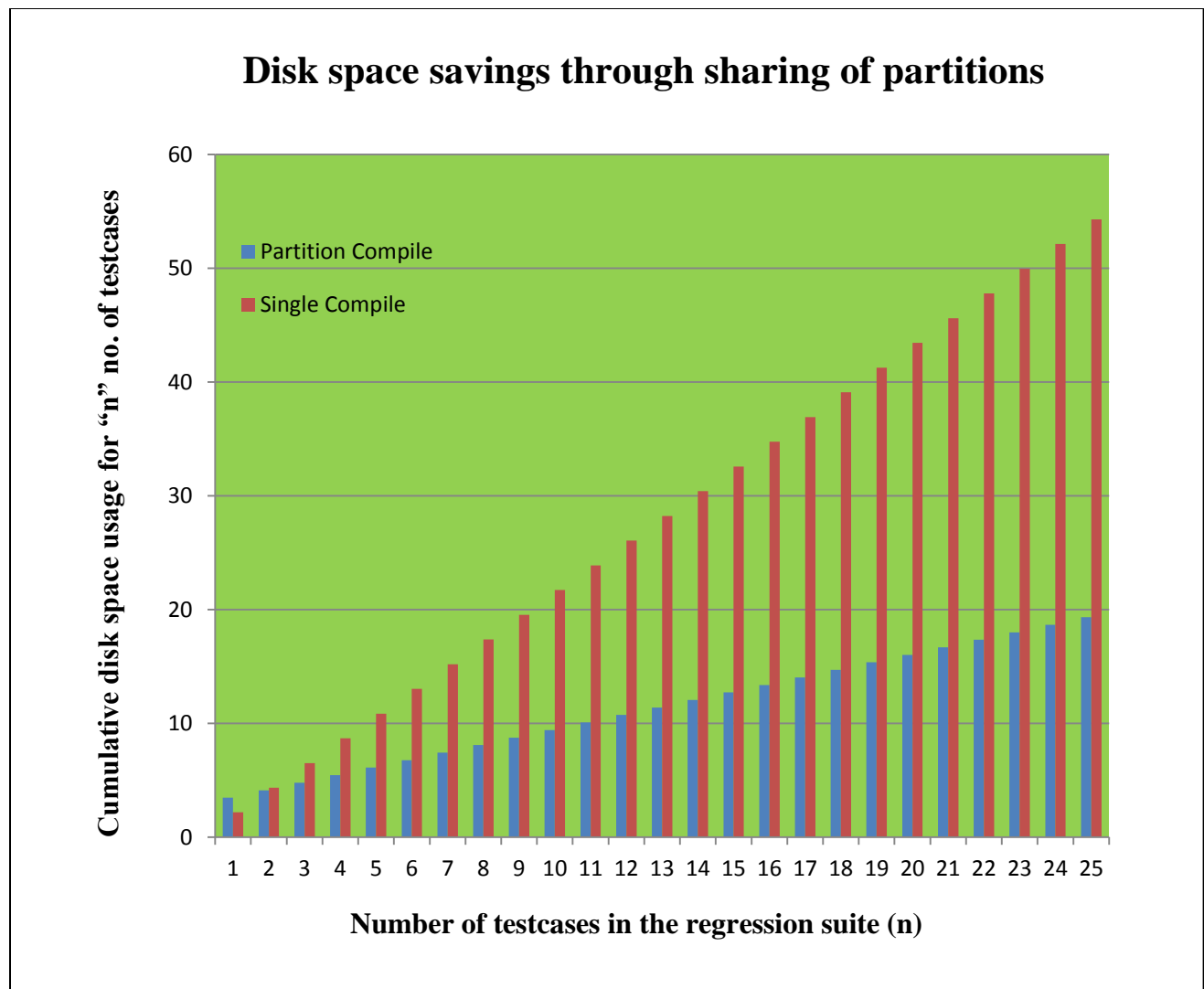


Figure 10. Disk space savings through sharing of partitions

CONCLUSION-

~65 % savings in disk space with partition compile as against single compile.

3.4.2 Parallel Compilation of the partitions (using multiple cores)

When scratch compile times are too long, partition the DUT and compile the partitions in parallel. Parallel compilation of partitions can significantly improve scratch compile times.

To enable the parallel compilation of partitions, use the *-fastpartcomp=j* option on the vcs command line.

3.4.3 Profiling Partition Compilation

Compile time for each partition should be similar. Parallel compile of partitions will not be useful if the compile time of each partition varies significantly.

"*-pctimeprof*" option can be used to get the compile time of each partition at "vcs" step.

```
MAKEPROF@work_DopplerGTop/partitionlib/WORK_DOPPLERGTOP_DopplerGTB_oKdA  
O/libvcspc_WORK_DOPPLERGTOP_DopplerGTB_oKdAO.so, Real: 1069.56, User: 940.24,  
Sys: 18.49  
MAKEPROF@work_DopplerDTop/partitionlib/WORK_DOPPLERDTOP_DopplerDTB_6TaV  
Pd/libvcspc_WORK_DOPPLERDTOP_DopplerDTB_6TaVPd.so, Real: 1396.25, User:  
1249.21, Sys: 19.71  
MAKEPROF@work_sysdgTB/partitionlib/WORK_SYSDGTB_sysdgTB_MFFunc/libvcspc_W  
ORK_SYSDGTB_sysdgTB_MFFunc.so, Real: 7.59, User: 5.66, Sys: 0.69
```

3.4.4 Scenarios Causing a Recompilation of a Partition

The following cases cause a recompilation of a partition:

- Change of a module definition in a partition, either instance- or cell-based.
- Change of a vcs command-line option.
- Change in the *synopsys_sim.setup* or *+optconfigfile* or *+v2k* config file.
- Global design property changes like adding *\$dumpvars* to a partition, in which case all partitions need to be recompiled.
- Changes to the ports or parameters that affect the ports of a partition. If there is a parent and child partition, and this change is only to the child partition, the parent need not be recompiled.
- Changes to *\$unit* scope. Any change (addition of new signal, type, module, or interface) to a *\$unit* causes recompilation of all the partitions associated with the *\$unit*.
- Changes to an interface require the recompilation of all modules connected to that interface.
- Changes to a shared package. In this case all the partitions sharing this package require recompilation.

- Reanalysis of source code (*vlogan* or *vhdlan*) with different command-line options causes recompilation of the source code and all partitions that share the logical library.

3.4.5 Partition Compile Limitations

The following technologies are not supported with Partition Compile:

- The timing optimizer (*+timopt*).
- Searching for the *PATHPULSE\$* specparam in specify blocks with *+pathpulse*.
- Multicore DLP (DLP is an LCA feature)
- AMS is not supported, but mixed-signal simulation with FastSPICE or XA is supported.
- Using the *+optconfigfile* or *+v2k config file* method of specifying partitions is only enabled for Verilog and SystemVerilog code.

3.4.6 High level guidelines for partition compile flow

- 1) The first compile time will be long if there are too many partitions.
 - Do not create unnecessary partitions.
 - Create one or more partitions of the required components which are going to change frequently.
 - Create one or more partitions of components which have a long compile time.
 - Create one or more partitions for SV packages which are going to change frequently.
 - Do not create more than 8-10 partitions in general.
- 2) The first compile and recompile times of very large partitions will be longer than smaller partitions.
 - Ensure that the partitions you create are balanced in terms of compile time.
 - If the compile time is very long for some partitions and very short for other partitions then the overall compile time will still be long.
- 3) XMRs across partitions without having an *XMR config file* or *-partcomp=noxmrcfg* option will lead to multiple partitions being recompiled if the XMRs change during recompilation.
 - No care to be taken when XMRs are not changing during recompilation.
- 4) External IP/VIP modules can be made separate partitions as these don't need to be recompiled.

4. VCS techniques to improve RUN TIME performance

Unlike compile time performance which plays a key role in the *initial phase of the design* development cycle, runtime performance is important in regression phase or in the *final phase of the design* development cycle.

4.1 Radiant Technology

VCS MX has built-in Radiant technology which can dramatically boost the simulation runs while running long regressions. VCS MX Radiant Technology applies performance optimizations to the Verilog portion of your design while VCS MX compiles your Verilog source code. These Radiant optimizations improve the simulation performance of all types of designs from behavioral, RTL to gate-level designs. Radiant Technology particularly improves the performance of functional simulations where there are no timing specifications or when delays are distributed to gates and assignment statements.

Radiant Technology brings in certain code based optimizations where it internally generates Meta code for certain constructs to bring runtime improvement.

Some of these optimizations include

- Function inlining
- Task inlining
- Loop unrolling
- Common subexpression elimination
- Dead code (unreachable code) elimination

Compiling With Radiant Technology

Radiant Technology optimizations are not enabled by default. You enable them using the compile-time options:

+rad

Specifies using Radiant Technology

+optconfigfile

Optional. Specifies applying Radiant Technology optimizations to part of the design using a configuration file.

Table 9. shows improvement in run time in our SystemDV simulations using radiant technology

| System Configuration | Simulation time | | Reduction in simulation time |
|----------------------|----------------------------|--------------------------------|------------------------------|
| | Without radiant technology | With radiant technology (+rad) | |
| SC 1 | 60 mins | 50 mins | 16.67% |
| SC 2 | 80 mins | 65 mins | 18.75% |
| SC 3 | 120 mins | 95 mins | 20.83% |

Table 9. Improvement in Simulation time using radiant technology

CONCLUSION

Overall we are getting ~18% reduction in our simulation time using radiant technology

KNOWN LIMITATIONS OF RADIANT TECHNOLOGY

- Avoid dumping VCD files while using rad optimizations as it causes a performance hit.
- Back-annotation of SDF files is not supported.
- Can lead to race-conditions in the design (Random stability not guaranteed).
- Can change code-coverage results going from run to run.
- Incremental compile times are longer with this (~5% increase observed in our design) but still shorter than a full recompilation of the design).

4.2 Save & Restore

Save & Restore is a novel approach to reduce the overall regression run time of testcases by avoiding the redundant execution of some common init routines which are common for all the simulations in a particular regression suite.

A typical simulation flow consists of the following steps-

- Reset sequence
- Init sequence (programming registers)
- Run phase (send traffic)
- EOTC (End of Test checks)

Reset and some part of init routines are common for all simulations and usually takes lot of simulation cycles. Run phase is the phase where the real dynamics of simulation is there because

this is the phase which actually stresses the DUT, most of the corner case bugs are exposed here, and most of the activity in terms of RTL coverage is seen here .

Now the question is can we avoid the time consuming and repetitive reset and init routines for all the simulations. This is where VCS save and restore feature comes to our rescue, where we execute the reset and common init routines (common for all the simulations) only once, create a checkpoint at that point, and reuse that checkpoint for all the subsequent simulations.

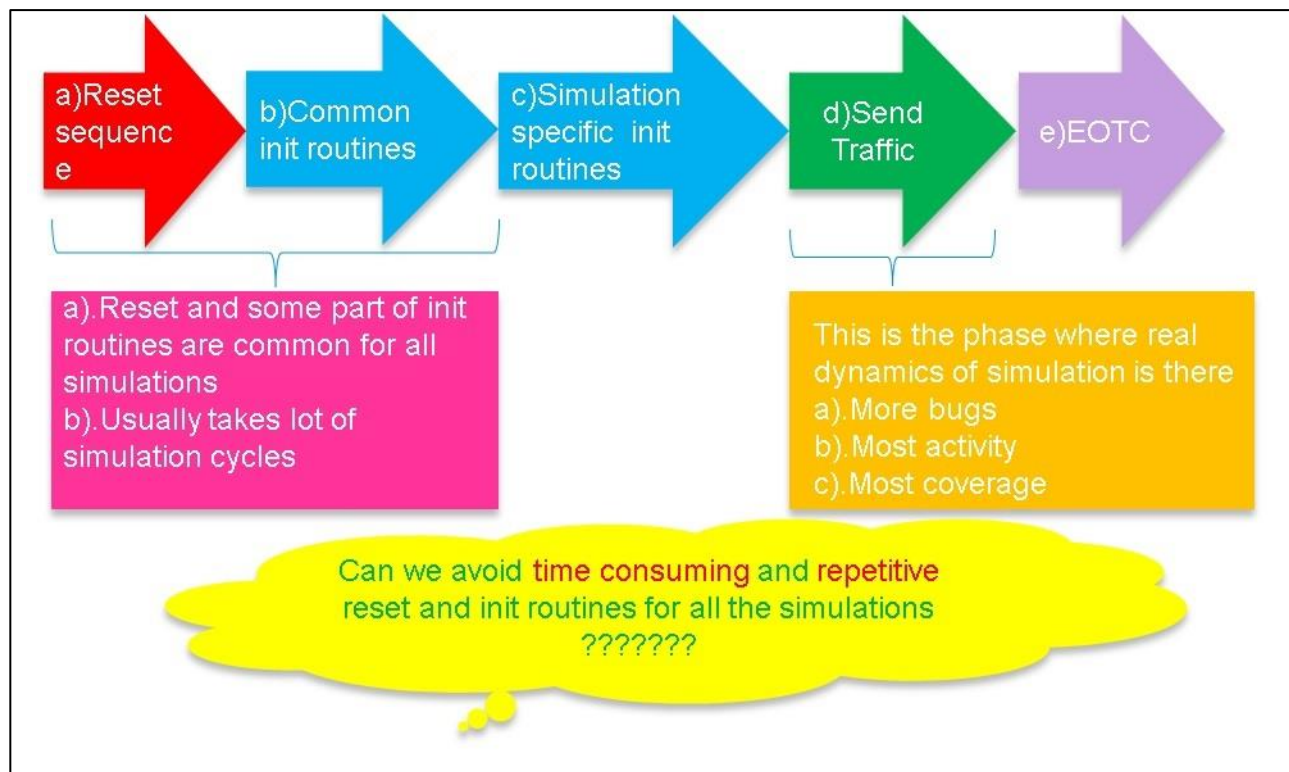


Figure 11. Typical Simulation flow

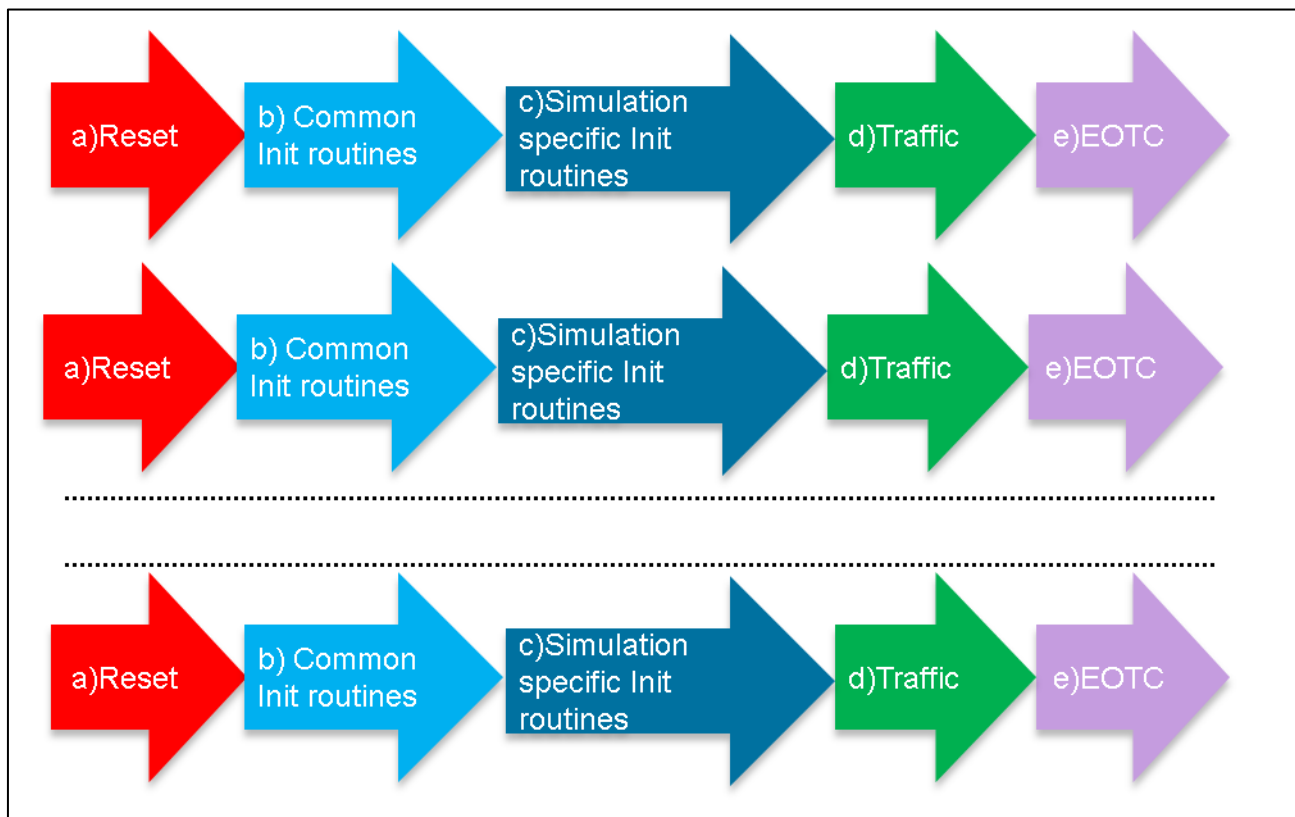


Figure 12. Traditional simulation flow

In Save & Restore, we run a single simulation and create a checkpoint save_and_restore.chk. Checkpoint includes the DUT configuration which is common for all the simulations i.e. configuration from time $t=0$ to time $t=T$ (t being simulation time), where T is the time when all the common init sequence and init routines are complete. Idea behind creating a checkpoint is to do the init routine only once and reuse it for all the simulations.

**Total saving in terms of CPU time = (Time for init sequence till checkpoint generation)*
(Total number of simulations).**

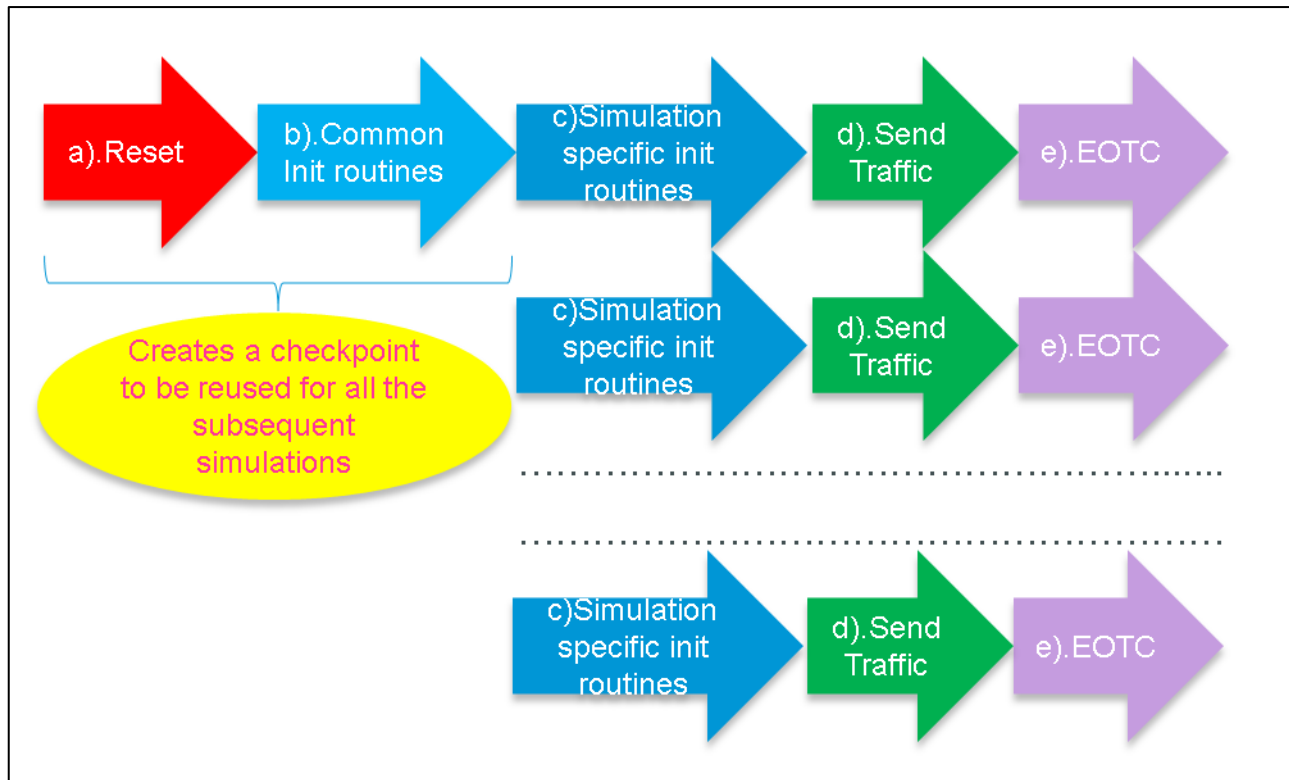


Figure 13. Save & Restore (How does it help??)

How to implement SAVE and RESTORE to optimize regression run time???

Use the two session management commands - *\$save* and *\$restore*.

1) \$save

Use this command to store the current simulation snapshot in a specified file.

This command saves the entire simulation state including breakpoints set at the time of saving the simulation.

2) \$restore

Use this command to restore the saved simulation state from a specified file.

This command restores the entire simulation state including breakpoints set at the time of saving the simulation.

A simulation can be restored multiple times by using different (or same) simulation snapshots (of same tool).

SAVE and RESTORE (An illustration)

```
% cat test.v
module simple_restart;
initial begin
    #10
    $display("one");
    $display("two");
    $save("test.chk");
    $display("three");
    #10
    $display("four");
end
endmodule
```

Now compile the example source file:

```
% vcs test.v
```

Now run the simulation:

```
% simv
```

VCS displays the following:

```
one
two
$save: Creating test.chk from current state of simv...
three
four
```

To restart the simulation from the state saved in the check file, enter:

```
% simv -r test.chk
```

VCS displays the following:

```
Restart of a saved simulation
three
four
```

SAVE & RESTORE (Simulation Results)

Save & Restore was used for running performance simulation in ProjectX (a 35 million gate ASIC)

| Scenario | Total CPU time (in hours) | Observations |
|------------------------|---------------------------|---|
| Without save & restore | 63,000 hours | Total savings of 36,000 hours of CPU time |
| With save & restore | 27,000 hours | |

Table 10. SAVE & RESTORE – Run time comparison

Time for common init routines ~ 4 hours
Total number of simulations ~ 9000
Total savings in terms of CPU time = 4 x 9000 hours
 = 36,000 hours.

Assuming that we are getting 100 LSF slots for ProjectX (dictated by LSF scheduler)
Total savings = 36,000/100 hours
 = 360 hours (15days)

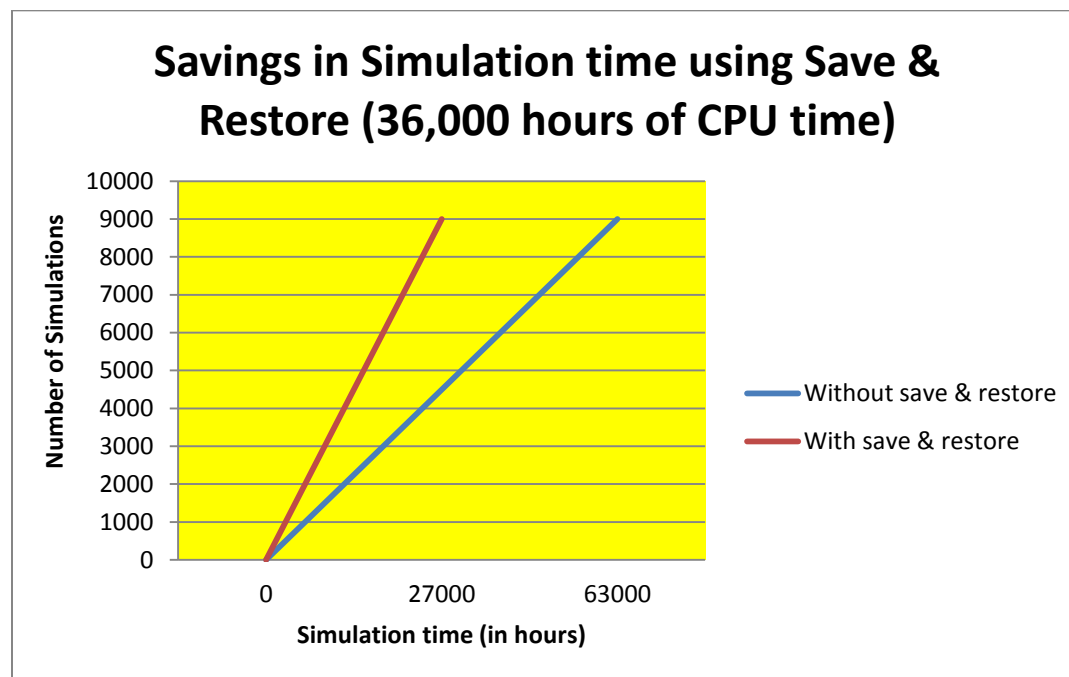


Figure 14. SAVE & RESTORE – Simulation time comparison

4.3 Interactive Rewind

This is an extension to save & restore. We can create multiple simulation snapshots using the UCLI "Checkpoint" feature during an interactive debug session. In the same debug session, we can go back to any of those previous snapshots, by using the UCLI "Rewind" feature and do 'What if' analysis. When we create multiple checkpoints, say at times "t1, t2, t3, ...tn", and you want to rewind from your current simulation time to a previous simulation time say t2, then all the checkpoints that follows t2 (t3, t4 etc.) gets deleted. This is intentional, because when we go back to history using the rewind operation, you are given an option to force/release the signal values and continue with a different simulation path until you get the desired results.

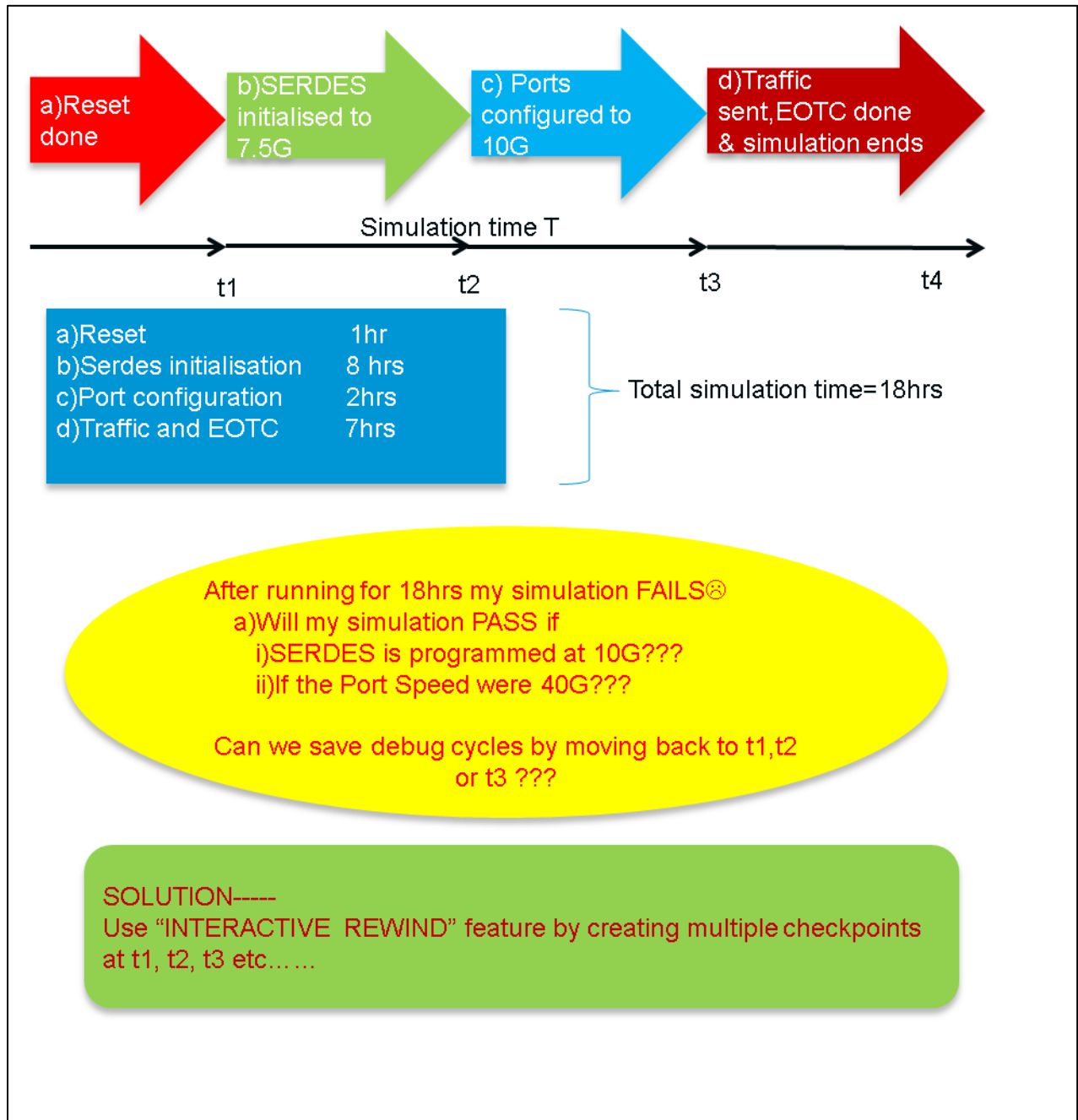


Figure 15. Interactive Rewind (An illustration)

INTERACTIVE REWIND (Advantages)

- We can do "What if" analysis.
- This way, you need not restart your simulation from time zero and you save time.

4.4 VCS Multicore Technology

VCS Multicore Technology takes advantage of the computing power of multiple processors in one machine to improve simulation turnaround time. Design Level Parallelism is an LCA feature. VCS Multicore Technology feature, includes both design level parallelism (DLP) and the GA Application Level Parallelism (ALP).

Candidates for DLP should be a long running simulation. Short running simulations of an hour or less may not show much value to user even if DLP can show some gain. It's the long running testcase typically several hours/days where you will see most value if DLP can demonstrate any gain. Its recommended to do design profiling before using DLP.

ALP can be used in the following scenarios:

- Assertion simulation
- Toggle coverage
- Multicore functional coverage
- VPD dumping
- SAIF dumping

VCS Multicore Technology Options

You use the VCS -parallel option to invoke parallel compilation. The syntax is:

```
vcs filename(s).v -parallel [+multicore_option(s)]  
[-parallel+show_features][-o multicore_executable_name]  
[vcs-options]
```

These options and properties are as follows:

-parallel

When used without VCS Multicore options, -parallel enables all VCS Multicore Technology options, except for design level parallelism. When used with VCS Multicore options, -parallel enables only those option specified.

This option is available at compile-time only.

+design=FILENAME

Enables design level parallelism and specifies the name of the partition configuration file. Note: this option is available at compile-time only.

+fc[=NCORES]

This compile-time option, enables Multicore Functional Coverage, and with *NCORES* specifies the number of cores or PFC consumers. *NCORES* can be changed at run time.

```
vcs -parallel+fc ...  
vcs -parallel+fc=3 ...
```

+profile

Enables Multicore design and application level profiling

+profile value

Enables value-based design level profiling.

+sva[=*NCORES*]

This compile-time option enables multicore SVA, and with *NCORES* specifies the number of cores or multicore SVA consumers. *NCORES* can be changed at run time.

+tgl[=*NCORES*]

Enables multicore Toggle Coverage, and specifies the number of multicore toggle coverage consumers. To enable the use of the same executable for both serial and parallel runs, use this option at runtime. *NCORES* specifies the number of multicore SVA consumers. For ALP, *NCORES* can be changed at run time.

+vpd[=*NCORES*]

Enables multicore VCD+ Dumping, and specifies the number of multicore VCD+ consumers. To enable the use of the same executable for both serial and parallel runs, use this option at runtime. *NCORES* specifies the number of multicore SVA consumers. For ALP, *NCORES* can be changed at run time.

[-o *multicore executable name*]

Using the VCS -o option to specify the simulation executable binary filename allows work on multiple simultaneous VCS Multicore compiles and runs. VCS Multicore-specific data is stored in a directory *executable_name*.pdaidir. The default path name is simv.pdaidir.

Note: If [*NCORES*] is not specified, the default is 1 client.

-parallel+show features

Displays enabled VCS Multicore features. Note that you must enter the -parallel option with +show_features

Examples:

-parallel+vpd is equal to -parallel+vpd=1

-parallel+tgl is equal to -parallel+tgl=1

VCS Multicore option examples:

vcs -parallel+design=part.cfg

vcs -parallel+fc -o psimv

vcs -parallel+vpd+fc -parallel+tgl -o par_simv

vcs -parallel+design=part.cfg+sva

| System Configuration | Simulation time | | Reduction in simulation time |
|----------------------|-----------------|--|------------------------------|
| | Without ALP | With ALP ON (-parallel+vpd=3 on a 4-Core CPU) | |
| SC 1 | 230 mins | 150 mins | 34.78% |
| SC 2 | 270 mins | 180 mins | 33.33% |
| SC 3 | 330 mins | 220 mins | 33.33% |

Table 11. Improvement in simulation time (when ALP is turned ON for VPD dump)

CONCLUSION

Overall we got ~35% in our simulation time while running our simulations with ALP ON for VPD dumping.

4.5 Dump scope control

Dumping simulation signals and debugging the simulation results from the dump are very essential for verification of ASICs. This consumes a lot of simulation time and memory, especially for large designs. In this sub-section we discuss enhancements we brought into our verification environment to improve dumping efficiency.

VCS provides users great flexibility controlling many dumping options at runtime. VCS provides system tasks like *\$vcdpluson*, *\$vcdplusoff*, *\$vcdplusfile*, *\$vcdplusmemon*, *\$vcdplusmemoff* to dump simulation results in VPD format.

Controlling what design scope(s) to dump improves the simulation time and also reduces the size of the VPD created. The improvement seen on simulation time and size of the VPD when we dump all the signals vs. when we dump only signals in specified scope is captured in Table 12. In our case “*dump only signals in specified scope*” means dumping signals only for the modules which are in boundaries between two ASICs which helped in debugging any System Level issues arising because of interoperability issues in between two ASICs

| System Configuration | Scenario | Simulation Time | VPD size | Observations |
|----------------------|---|-----------------|----------|---|
| SC1 | Start simulation at time 0 and dump all the signals | 150 mins | 9 GB | Simulation time was improved by 40% and size was reduced by 88% |
| | Start simulation at time 0 and dump only signals in specified scope | 90 mins | 1 GB | |
| SC2 | Start simulation at time 0 and dump all the signals | 180 mins | 11 GB | Simulation time was improved by 38% and size was reduced by 89% |
| | Start simulation at time 0 and dump only signals in specified scope | 110 mins | 1.2 GB | |
| SC3 | Start simulation at time 0 and dump all the signals | 220 mins | 12 GB | Simulation time was improved by 31% and size was reduced by 87% |
| | Start simulation at time 0 and dump only signals in specified scope | 150 mins | 1.5GB | |

Table 12. Savings in Disk Space and Simulation time using Dump scope control

5 Use of VCS Simulation profiler to identify Simulation bottlenecks

VCS simulation profiler reports the amount of CPU time and machine memory used by the design and testbench components. The profiling information can be reported in different views related to time and memory, and in different levels of granularity : per module definition, module instance, based on constructs such as always, initial blocks, functions, tasks etc.

To enable the simulation profiling capability, add the compile-time option `-simprofile` to the compilation command. Then at run-time, add the option `-simprofile <keyword>`. The additional sub-option `<keyword>` specifies the type of data VCS collects during the simulation: `time` specifies collecting data on the CPU times, and `mem` specifies collecting data on the machine memory.

5.1 Signal dumping was consuming lot of simulation time & machine memory

Simulation profiling reports showed large percentage of simulation time and machine memory was spent in dumping signals.

Figure 16 & 17 shows summary of the VCS profiling result. It can be seen that *10% of the simulation time and 36% of the machine memory* was spent on VPD dumping.

| Time Summary | | |
|----------------------|-----------------|-------------|
| Component | Time | Percentage |
| VERILOG | 1050.2 s | 70.50% |
| HSIM | 218.3 s | 14.12% |
| Value Change Dumping | 153.9 s | 10.33% |
| KERNEL | 57.8 s | 3.88% |
| DEBUG | 8.2 s | 0.55% |
| PROFILE | 5.0 s | 0.33% |
| PLI/DPI/DirectC | 4.1 s | 0.28% |
| ASSERTION_KERNEL | 63.5 ms | 0.00% |
| total | 1489.6 s | 100% |

Figure 16. Simprofile time report

| Memory Summary | | |
|----------------------|--------------------|-------------|
| Component | Size | Percentage |
| VERILOG | 7176.12 MB | 43.21% |
| Value Change Dumping | 5979.61 MB | 36.00% |
| PLI/DPI/DirectC | 1807.13 MB | 11.42% |
| HSIM | 1369.92 MB | 8.25% |
| KERNEL | 161.34 MB | 0.97% |
| ASSERTION_KERNEL | 23.91 MB | 0.14% |
| total | 16608.33 MB | 100% |

| dynamic Summary | | |
|-------------------------|------------------|-------------|
| Dynamic Object | Size | Percentage |
| SmartQueue | 360.39 KB | 48.37% |
| Class | 101.70 KB | 13.65% |
| String | 97.92 KB | 13.14% |
| Event | 73.88 KB | 9.91% |
| AssociativeArray | 67.59 KB | 9.07% |
| AssociativeArrayElement | 25.15 KB | 3.38% |
| SmartQueueElement | 18.13 KB | 2.43% |
| DynamicArray | 384 B | 0.05% |
| total | 745.13 KB | 100% |

Figure 17. Simprofile mem report

5.2 A performance bug in our generic memory model was exposed

VCS profiler helped us in finding a performance issue in our EDRAM memory model. Profiling of our SystemDV simulations showed that simulation time of our generic EDRAM models was consuming ~25% of the total simulation time in the simulations. This was obviously impacting all of our simulations since these models are common.

| Time Module View | | | | |
|------------------------------|----------------|------------|----------------|------------|
| Module | Inclusive Time | Percentage | Exclusive Time | Percentage |
| ▶ CEP_EDRAMWRAP_1R1W | -- | -- | 231.13 s | 14.40 % |
| ▶ CEP_EDRAMWRAP_1RW | -- | -- | 92.66 s | 5.77 % |
| ▶ PhcEdram | 275.91 s | 17.19 % | 43.25 s | 2.69 % |
| ▶ CEP_RAMWRAP_1RW | -- | -- | 30.33 s | 1.89 % |
| ▶ C8T28S0I_LL_SDFPRQX5_P16 | 17.30 s | 1.08 % | 17.30 s | 1.08 % |
| ▶ PcsSM | 16.08 s | 1.00 % | 16.08 s | 1.00 % |
| ▶ PcsSM | 15.63 s | 0.97 % | 15.63 s | 0.97 % |
| ▶ sysdgTB | 1125.61 s | 70.14 % | 13.83 s | 0.86 % |
| ▶ synchronizer | -- | -- | 13.08 s | 0.82 % |
| ▶ NifGallantAsyncFifoControl | -- | -- | 10.26 s | 0.64 % |
| ▶ NifTxDataBuffer | -- | -- | 8.00 s | 0.50 % |
| ▶ GalSyncOneBit | -- | -- | 7.46 s | 0.46 % |
| ▶ CEP_RAMWRAP_2RW | -- | -- | 6.76 s | 0.42 % |
| ▶ synchronizer | 6.46 s | 0.40 % | 6.46 s | 0.40 % |
| ▶ synchronizer | 6.40 s | 0.40 % | 6.40 s | 0.40 % |
| ▶ Dvppsys | 5.80 s | 0.36 % | 5.80 s | 0.36 % |
| ▶ UsgmiiPcsModel | -- | -- | 5.51 s | 0.34 % |
| ▶ CEP_RAMWRAP_1R1W | -- | -- | 5.42 s | 0.34 % |
| ▶ NifGeGroup | -- | -- | 5.17 s | 0.32 % |
| ▶ PcsSM | 5.08 s | 0.32 % | 5.08 s | 0.32 % |
| total | 1131.43 s | 70.50 % | 1131.43 s | 70.50 % |

Page: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52

Figure 18. Simprofile mem report (Time Module View)

We went back and had a look into the EDRAM generic model and found a real issue in the way one of the function was evaluated which was causing performance hit.

6. VCS 3-step flow

While setting up the infrastructure for SystemDV one of the biggest challenge was to compile all the files (of individual ASICs) together, with different vendor technology files, and duplicate module definitions. Duplicate module definitions (same module name getting used in different ASICs and their module definitions being different) was going to give module name conflicts. To avoid this we decided to use the VCS 3-step flow also known as UUM (Unified Use Model flow). Using this, the analysis of individual ASIC database were decoupled which avoided the module name conflicts.

VCS 3 step flow (Unified Use Model) involves following steps

1) Analyzing the design (vlogan command)

VCS MX provides you with the vhdlan and vlogan executables to analyze your VHDL and Verilog design code. vhdlan/vlogan analyzes your design and stores the intermediate files in the design or a work library.

Parsing of files in different ASICs can be decoupled using this. This avoids Module name conflicts e.g. same module name being present in different chips and having different module definitions and different IO port list also.

2) Elaborating the Design (vcs command)

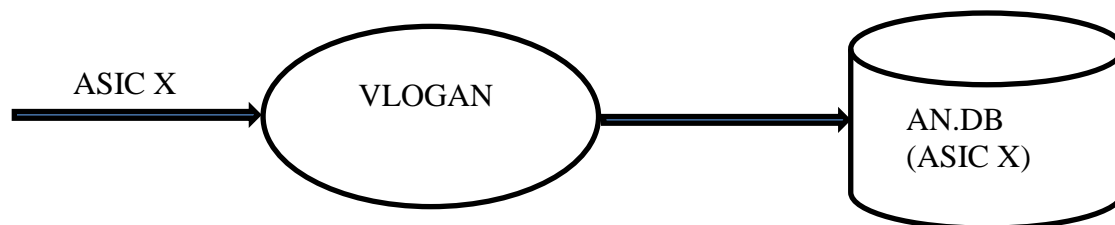
VCS MX provides you with the vcs executable to elaborate the design. This executable elaborates your design using the intermediate files in the design or work library, generates the object code, and statically links them to generate a binary simulation executable, simv.

3) Simulating the Design (./simv)

Simulate your design by executing the binary simulation executable, simv

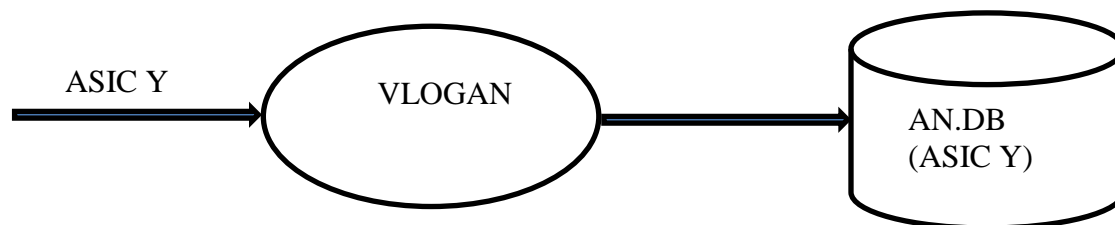
These steps are depicted as follows (taking ASIC X and ASIC Y as examples)

STEP 1 Analysis of ASIC X

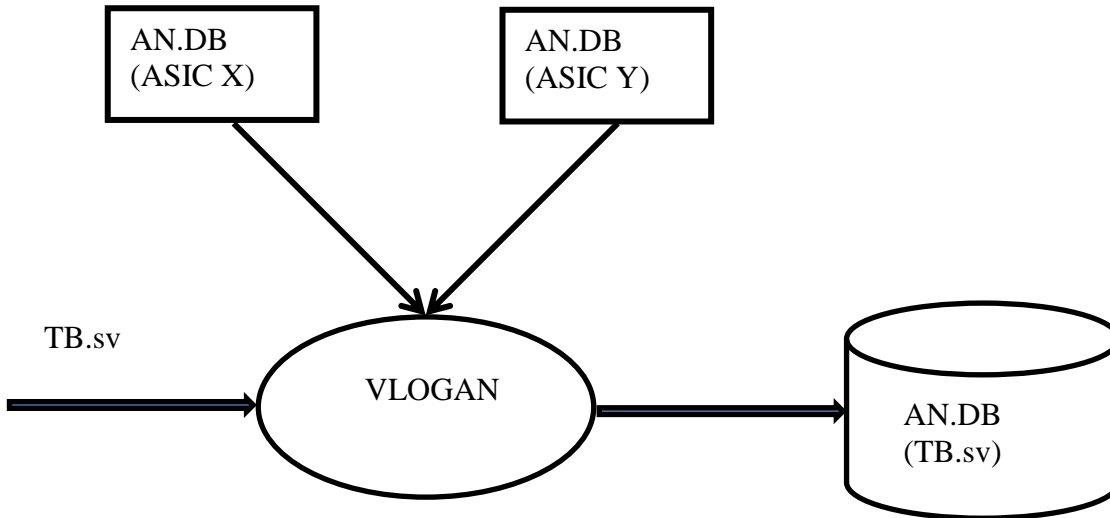


Note - AN.DB are the intermediate files created in WORK directory during the analysis step Which is used by subsequently for elaboration, creating object files and linking them statically.

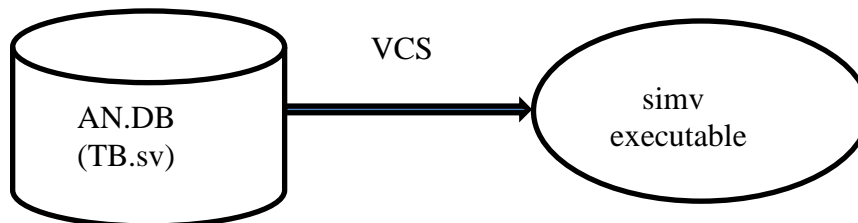
STEP 2 Analysis of ASIC Y



STEP 3 Analysis of Top level testbench (TB.sv) instantiating ASIC X and ASIC Y (point to the individual work AN.DB of both in the config file)



STEP 4 Elaboration of TB.sv (Top level testbench instantiating the 2 chips)



7. Results

For benchmarking the benefits of the optimized flow, simulations were run for all the three System Configs that we were testing. In each simulation, we sent 1000 Ethernet Packets (with Jumbo frame size). The below graph describes the Total Simulation Turn-Around Time (TAT) i.e. “Compile Time + Run Time” in the absolute bare-bone approach (where no optimization techniques) are used against the one achieved using above mentioned VCS techniques. Note that most options discussed here are about managing tradeoffs between Compile Time vs. Run Time but overall we got very good savings in our TAT especially after the 1st iteration.

- a) **1st ITERATION** is a fresh run with compilation done from scratch
- b) **2nd ITERATION** is a run in the same directory. So we already had our database compiled into a binary version by the previous compilation. Pre-compiled binary version is a prerequisite for some optimizations e.g. incremental compile & partition compile.

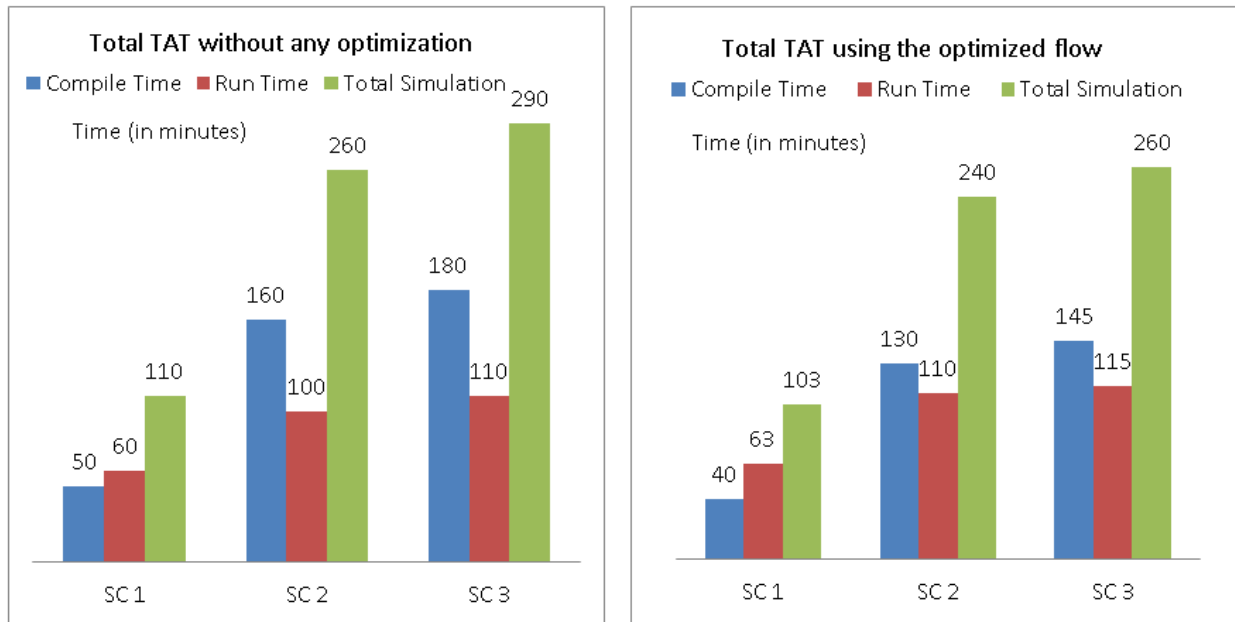


Figure 19. Comparison of Total TAT in the optimized flow vs. bare-bone approach (without any optimizations) → 1st ITERATION

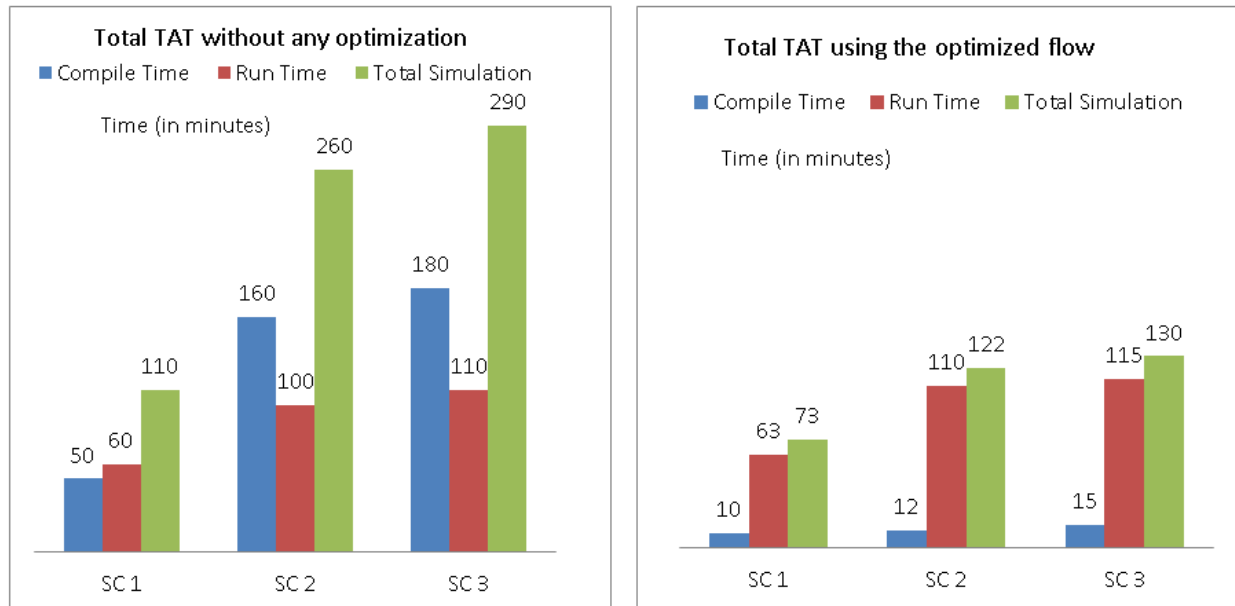


Figure 20. Comparison of Total TAT in the optimized flow vs. bare-bone approach (without any optimizations) → 2nd ITERATION

| System Config | Total TAT without any optimizations (bare-bone approach) | Total TAT using the optimized flow | Reduction in TAT |
|---------------|--|------------------------------------|------------------|
| SC1 | 110 mins | 103 mins | 6.36% |
| SC2 | 260 mins | 240 mins | 7.69% |
| SC3 | 290 mins | 260 mins | 10.34% |

Table 13. Reduction in TAT using the optimized flow (1st ITERATION)

| System Config | Total TAT without any optimizations (bare-bone approach) | Total TAT using the optimized flow | Reduction in TAT |
|---------------|--|------------------------------------|------------------|
| SC1 | 110 mins | 73 mins | 33.64% |
| SC2 | 260 mins | 122 mins | 53.08% |
| SC3 | 290 mins | 130 mins | 55.17% |

Table 14. Reduction in TAT using the optimized flow (2nd ITERATION)

8. Conclusions and Looking Forward

We were foreseeing tremendous performance challenges during the compilation and simulation of multiple ASICs while setting up our SystemDV infrastructure.

In our setup, testbench was having few lines of code changes (going from one run to other) during initial environment bringup. This was affecting the productivity of verification engineers during the testbench development phase and initial bringup of our SystemDV environment. Multiple iterations of simulations have to go through compilation again even for small changes and each compilation was taking around 1hour. Integration of partition compile flow in our setup gave a big boost to our compilation performance.

Also we got a good improvement in our runtime performance using the different VCS features like save & restore, interactive rewind and usage of multicore technology for VPD dumping.

And all these improvements we got using some of the features natively present in VCS. VCS simprofile gave us more insight into our design and testbench by identifying compilation and simulation bottlenecks. We were in collaboration with Synopsys team right from day one which helped us in seamlessly integrating all these optimization techniques in our flow and getting the whole flow working within 2 months of time. And all these without spending a single penny on our existing hardware and software resources. Disk space and LSF time slot saving was another benefit we got out of this optimized flow.

The paper also opens a large scope for the future work in this front, to help on

- **Multi-threaded simulation** Design-level parallelism (DLP) is achieved by partitioning a design and simulating it on multiple processor cores in parallel. Support for this is already there in VCS to some extent but this feature has couple of limitations which doesn't make it scalable and applicable for all designs. We need to explore this feature in some more detail and work with Synopsys so that this can be universally applied in our designs.
- **Dynamic black-boxing** Partial Elaboration flow can be used for dynamically black-boxing IP's/modules at runtime, thereby helping reduce memory footprint on large designs with single executable. This support is already present in VCS, need to see how we can leverage this in our SystemDV infrastructure.
- **Precompiled IP** Typical SoCs and big clusters of a chip consist of well-defined functional units which may have been designed by a different group, or could have been obtained from a 3rd party vendor. At any given time, not all of these functional units would be modified. Many remain static for weeks or even months. Having these functional units precompiled reduces the time to generate the final simv. The main advantage of using Precompiled IP (PIP) flow is fast elaboration/compile times when a Precompiled IP is shared across many targets/tests. Also sharing of Precompiled IP helps in Disk-space reduction. This is yet another promising feature that we are planning to explore in this front.

And finally, all these optimization techniques discussed in this paper can well be applied to any simulations (need not be Multi-chip simulations) where ever compile/run time needs to be optimized to save time.

9. Acknowledgements

We would like to thank all the team members from Synopsys for their timely support and tremendous contributions to the project. We start with Sharat Shreedhar and Shekhar Basavanna for being supportive to all our concerns right from day one. Special thanks to Sharat Shreedhar who was always eager to help us on all our queries and was very prompt in replying to all our mails. We also take this opportunity to convey our sincere appreciations towards Synopsys AE as well as the R&D team for their collaborative spirit in responding to any issues we faced during our initial bringup.

We are very thankful to Arun Abburu, Ashish Gogia and Milind Dixit who have guided me through the submission process of this paper. They gave their valuable suggestions and feedback during the implementation phase which made our SystemDV infrastructure robust and user-friendly.

10. References

- [1] vcsmx_ug.pdf
- [2] VCSLCAFeatues.pdf
- [3] IEEE paper “Incremental Compilation in the VCS Environment” by V. Kripa Sundar, Ashish V. Naik, Debashis Roy Chowdhury (Synopsys, Inc.)