VCS optimization techniques for Multi-Chip simulations

Debashis Biswas
CISCO

March 23, 2015
SNUG Silicon Valley

Agenda

Introduction

Challenges with Multi-Chip simulations

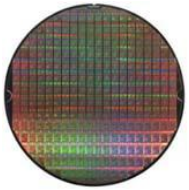
VCS techniques to improve COMPILE TIME performance

VCS techniques to improve RUN TIME performance

Results

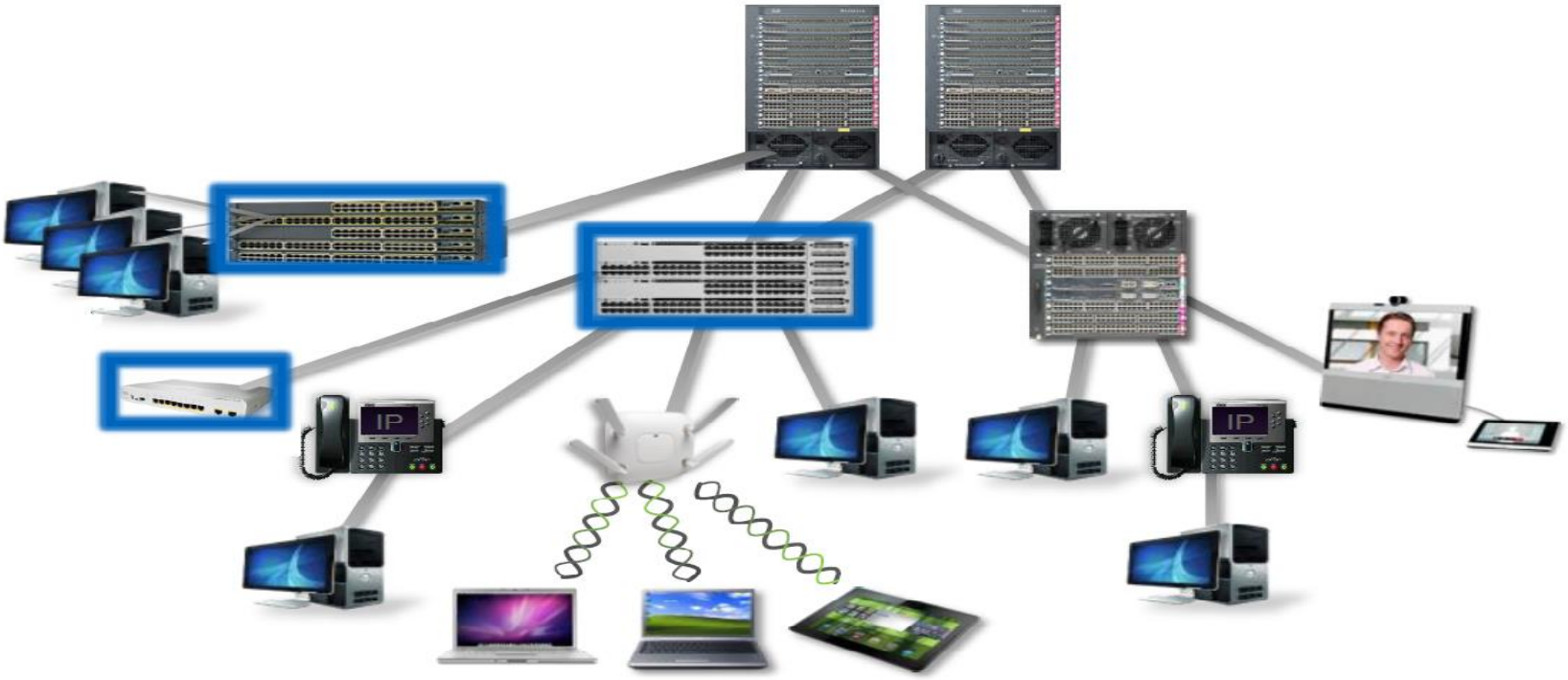
Conclusions and Looking Forward

Introduction



Cisco ASICs (Cisco's Secret Spice for Switches & Routers)

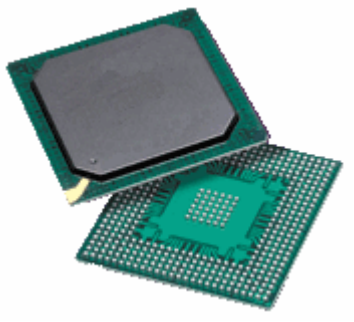
- Cisco ASICs powering the next generation of enterprise, campus, service provider, storage networks and Data Center switches.



- These ASICs support Unified Access, bring-your-own-device (BYOD) trend, mobility and Internet of Things (IoT).
- ASICs are complex ~150 million, truly System-On-chips.

Introduction

Why Multi-Chip Simulations??



- Silicon robustness needed to put better products to the market ahead of competition.
- Cost of the product needs to be reduced.
- Integration of more functionality into a single chip.
- ASICs used in Systems should be delivered in time (to reduce TTM).
- Integration & Interoperability of ASICs in the System as a whole should be seamless.

Introduction

How Multi-Chip Simulations helps??

- Validates that the current chip will work with any legacy chips.
- Validates any assumptions made between the chips.
- Multi-Chip (System) simulations focus on the system as a whole rather than the individual ASICs

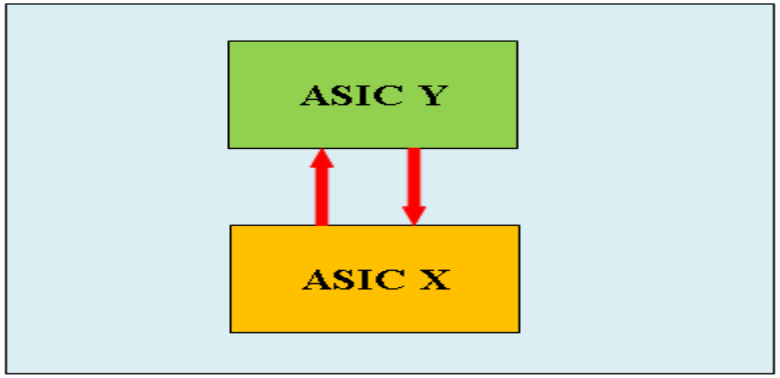
Our SystemDV setup for our next-gen products

ASIC X (design in progress) 28M+ multiport Ethernet switch containing MACs, forwarding logic, buffering and queuing logic/memory.

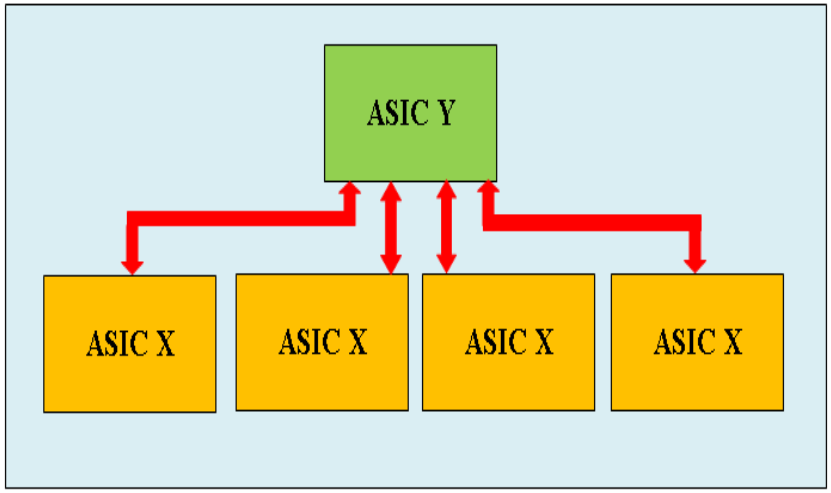
ASIC Y (design in progress) 59M+ multiport Ethernet switch containing MACs, forwarding logic, buffering and queuing logic/memory.

ASIC Z (legacy chip, already working in field)

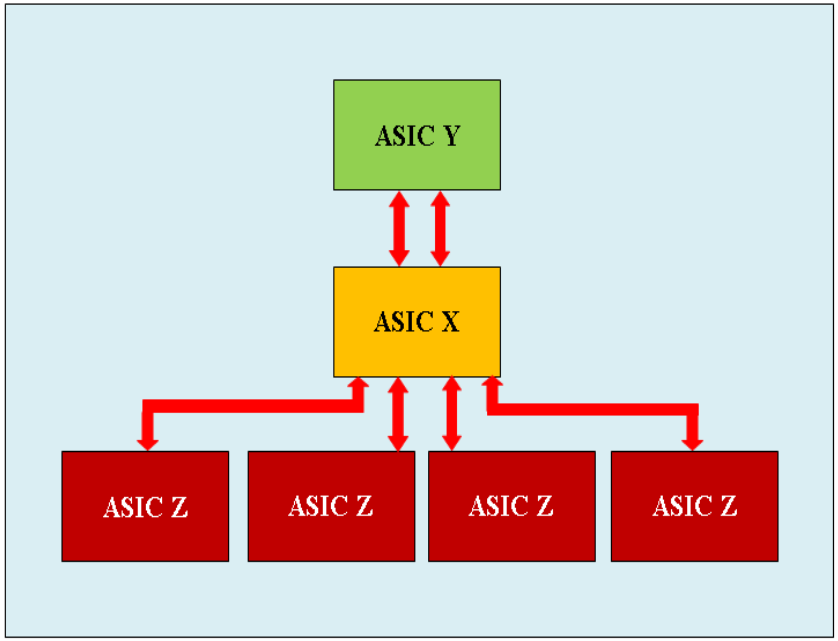
5M+ IEEE 802.1ae MacSec enabled Ethernet network ports aggregation chip.This has to interoperate with ASIC X & Y in the new systems.



System Config 1 (SC1)



System Config 2 (SC2)



System Config 3 (SC3)

Challenges with Multi-Chip Simulations

- A complete system-level test includes
  - many milliseconds of simulation time.
  - millions of lines of code
  - several gigabytes of memory
- Simulations is one of the key concerns (since overall gate count can be ~250-300 million).
- Compile time and run times are big bottlenecks to simulation.

VCS techniques to improve COMPILE TIME performance

Parallel compilation

Fast compilation

Incremental compilation

Partition compile

PARALLEL COMPILATION

- Object code generation during compilation divided into a number of threads.
- Individual threads are executed in parallel by a multi-core CPU.
- Infrastructure-based optimization.

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

- Specify using VCS compile-time option `-j[no_of_processes]`, as shown below

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

- For example, to fork off four parallel threads:

```
% vcs –j4 top
```

PARALLEL COMPILATION

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

- In general, there is no limit on number of cores that can be used
  - If resources aren’t sufficient, parallel compilation will be disabled and a warning will be issued
- Optimum choice of “N” is dictated by
  - Semaphore limitation of the machine’s OS
  - Sufficient memory to support the processes

PARALLEL COMPILATION

Memory usage overhead

- Memory overhead is N times that of a single “vcs” compile process
- Can be an issue for large designs where total memory usage exceeds virtual memory space

-j4 was an optimum choice for our very large design. Larger settings caused intermittent compilation failures because of the large memory usage.

PARALLEL COMPILATION

Improvement in compilation time using Parallel compilation

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%

Overall we are getting ~10% reduction in compilation time

FAST COMPILATION

- Enabled using ***–fastcomp*** compile-time option.
- Two levels of Fast Compilation:
  - `fastcomp=0`
    - same as `-fastcomp`
    - generally preferred option
  - `fastcomp=1`
    - applies more aggressive optimizations.

Word of caution!!!
Reduction in compile time will incur increase in run-time (in our design it was 5-7%)

FAST COMPILATION

Compile time reduction using *-fastcomp=0*

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%

Compile time reduction using *-fastcomp=1*

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%

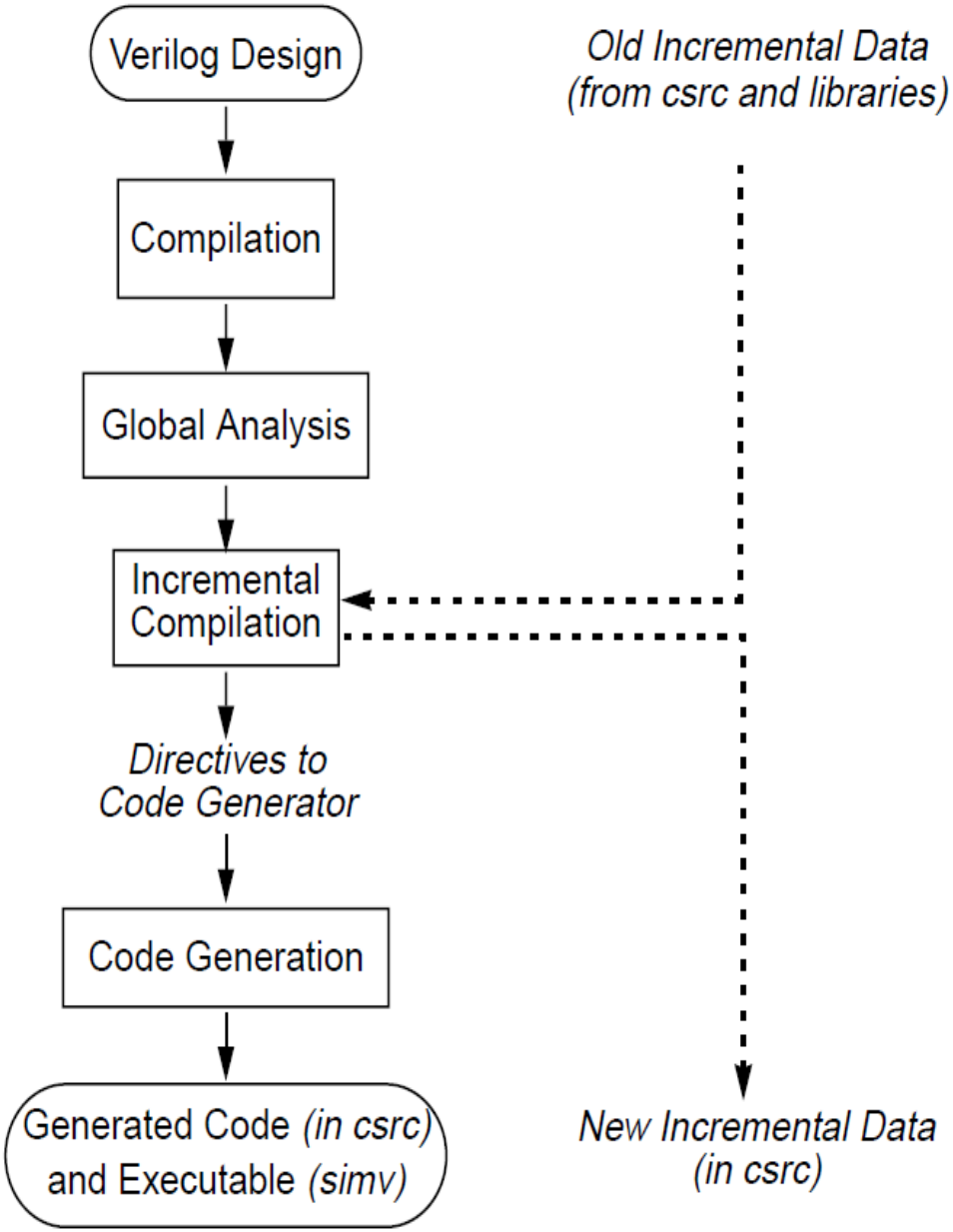
Overall we are getting ~6-10% reduction in compilation time

INCREMENTAL COMPILATION

- Invoked by the **-M** option
- Entire generated code is kept in **csrc** directory.
- “History” of successive compilations is kept track of in the **csrc** by incremental compilation.
- Compares the history against the current design to make re-compilation decisions.

INCREMENTAL COMPILATION

The Architecture of Incremental Compilation in VCS

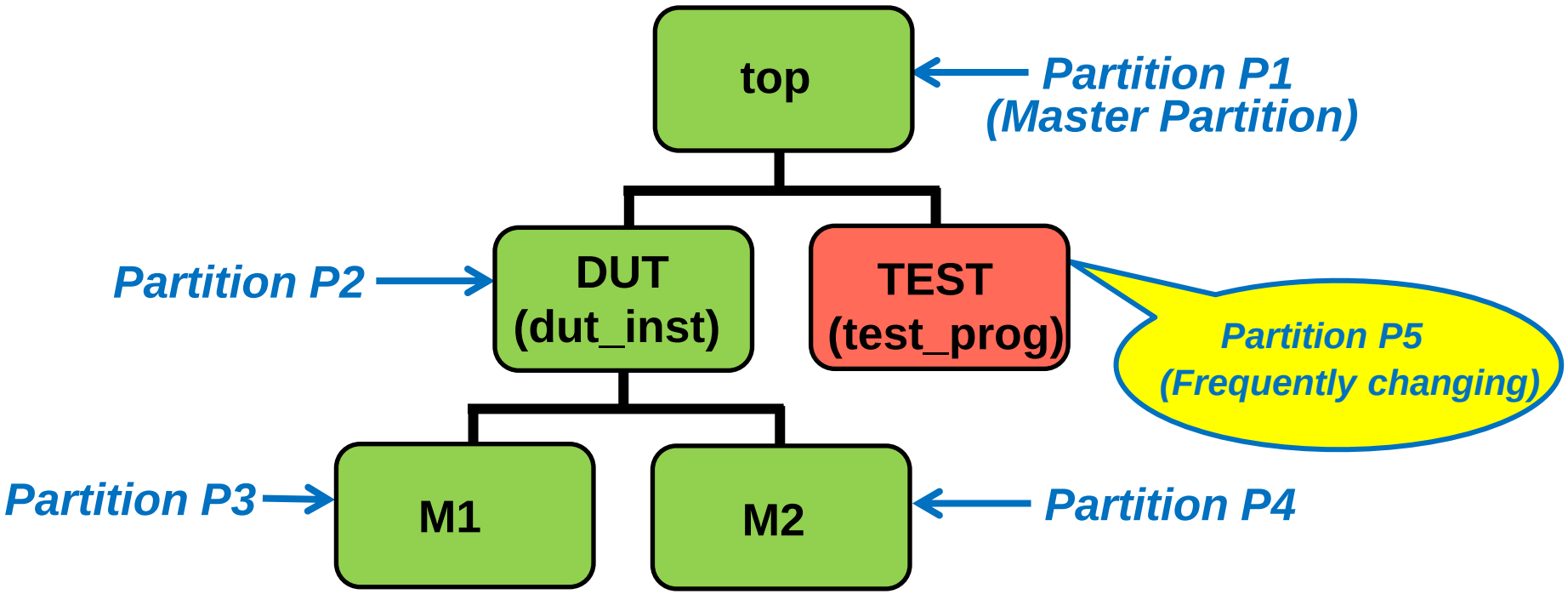


INCREMENTAL COMPILATION

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%

Overall we are getting 35% reduction in compilation time

PARTITION COMPILE



Partitions can be specified:

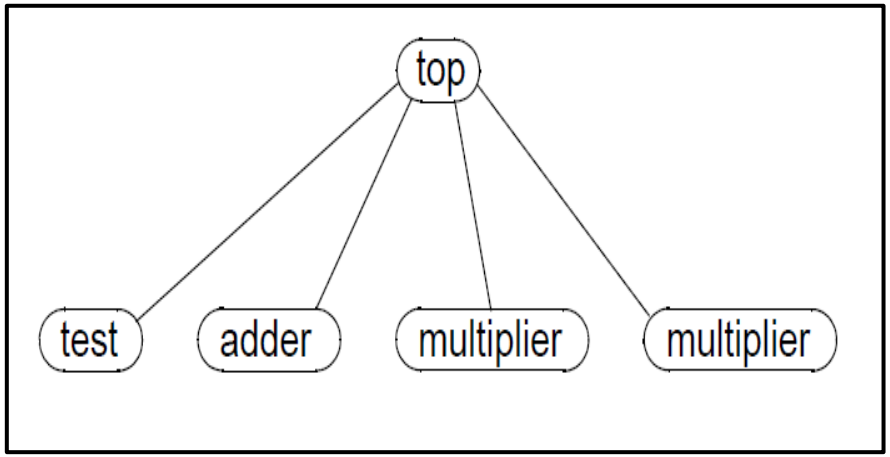
- Using Autopartitioning
- Manually in a v2k config file.

Slight runtime penalty (~5% observed in our design)

PARTITION COMPILE

Specifying partitions using v2k configuration file

Let's consider the following example hierarchy




```
//topcfg.v
config topcfg;
//top-level module
design top;
//partition for program test
partition instance top.t1;

instance top.m1 use multiplier;
instance top.m2 use multiplier;
//partition for multiplier instance m1
partition instance top.m1;
//partition for multiplier instance m2
partition instance top.m2;
endconfig
```

- The above configuration specifies
  - one partition for program block test
  - one partition for m1 instance
  - one partition for 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 ex-:

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

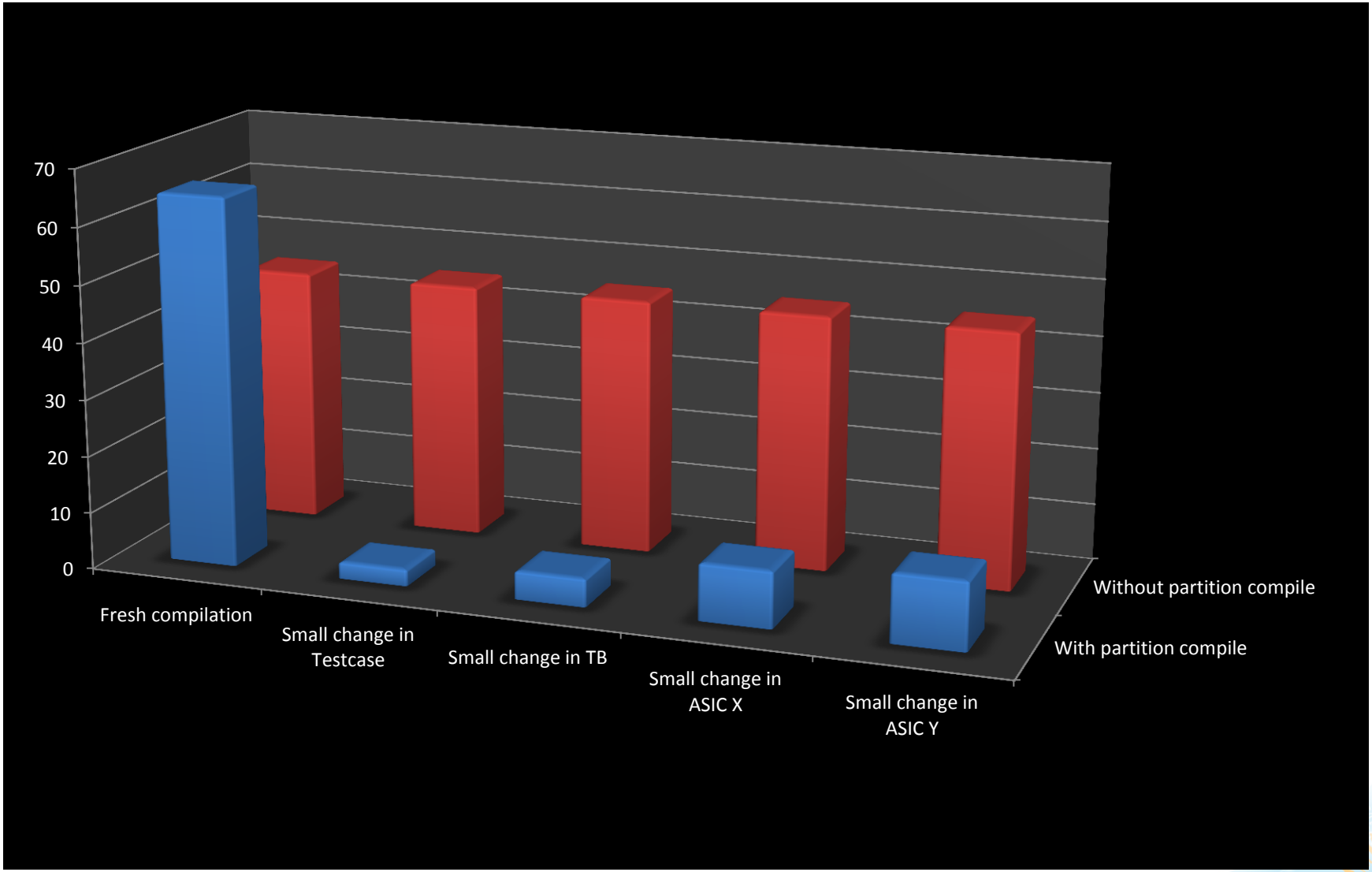
PARTITION COMPILE

Guidelines for specifying Partitions

- VCS created ***auto-partitions*** are good to go in most of the cases.
- 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.
- When in doubt, use ***"-pcmakeprof"*** to profile the time spent in compiling each partition.

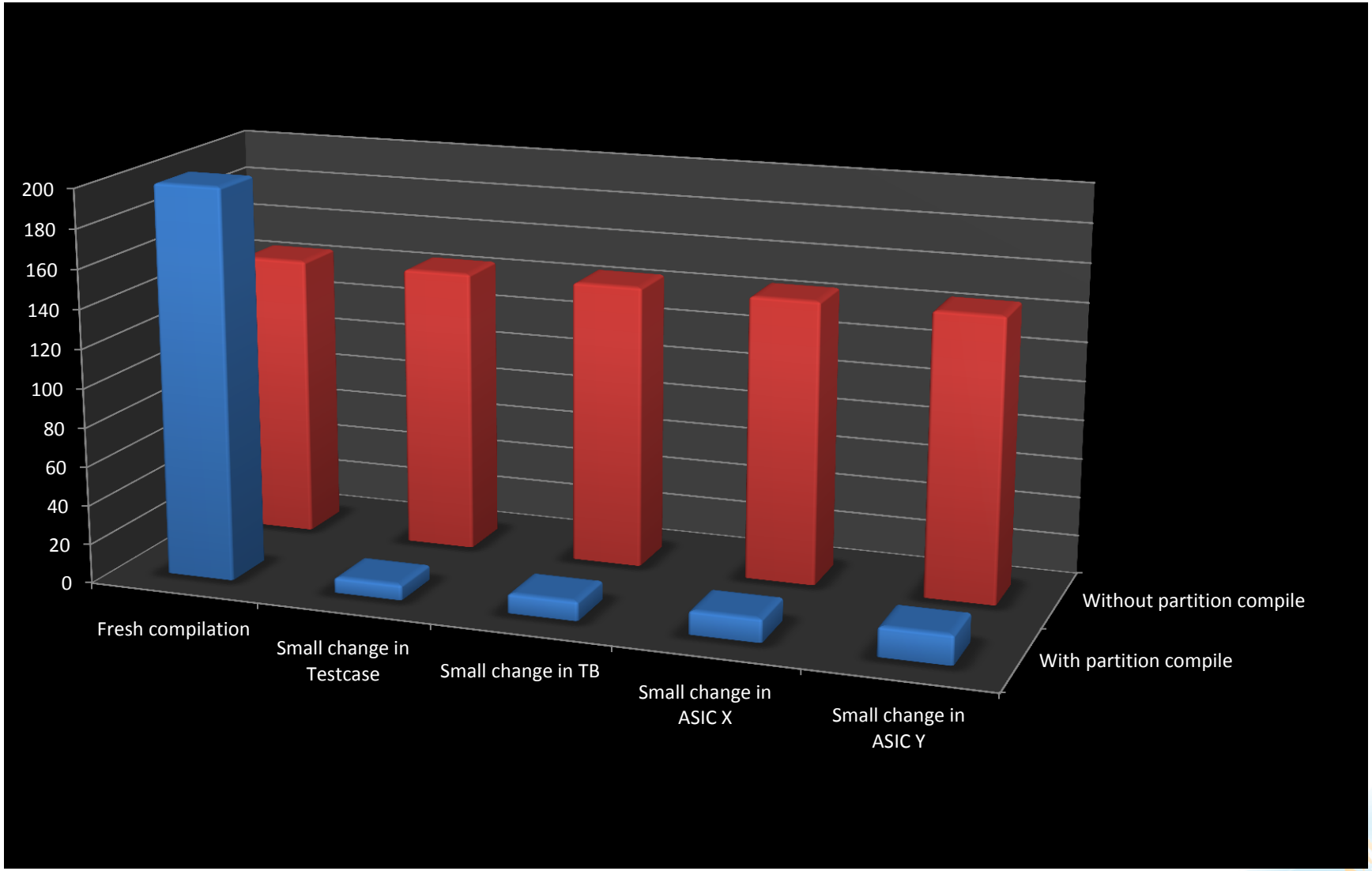
PARTITION COMPILE

Simulation Results-Compile time comparison (in mins) in System Config 1



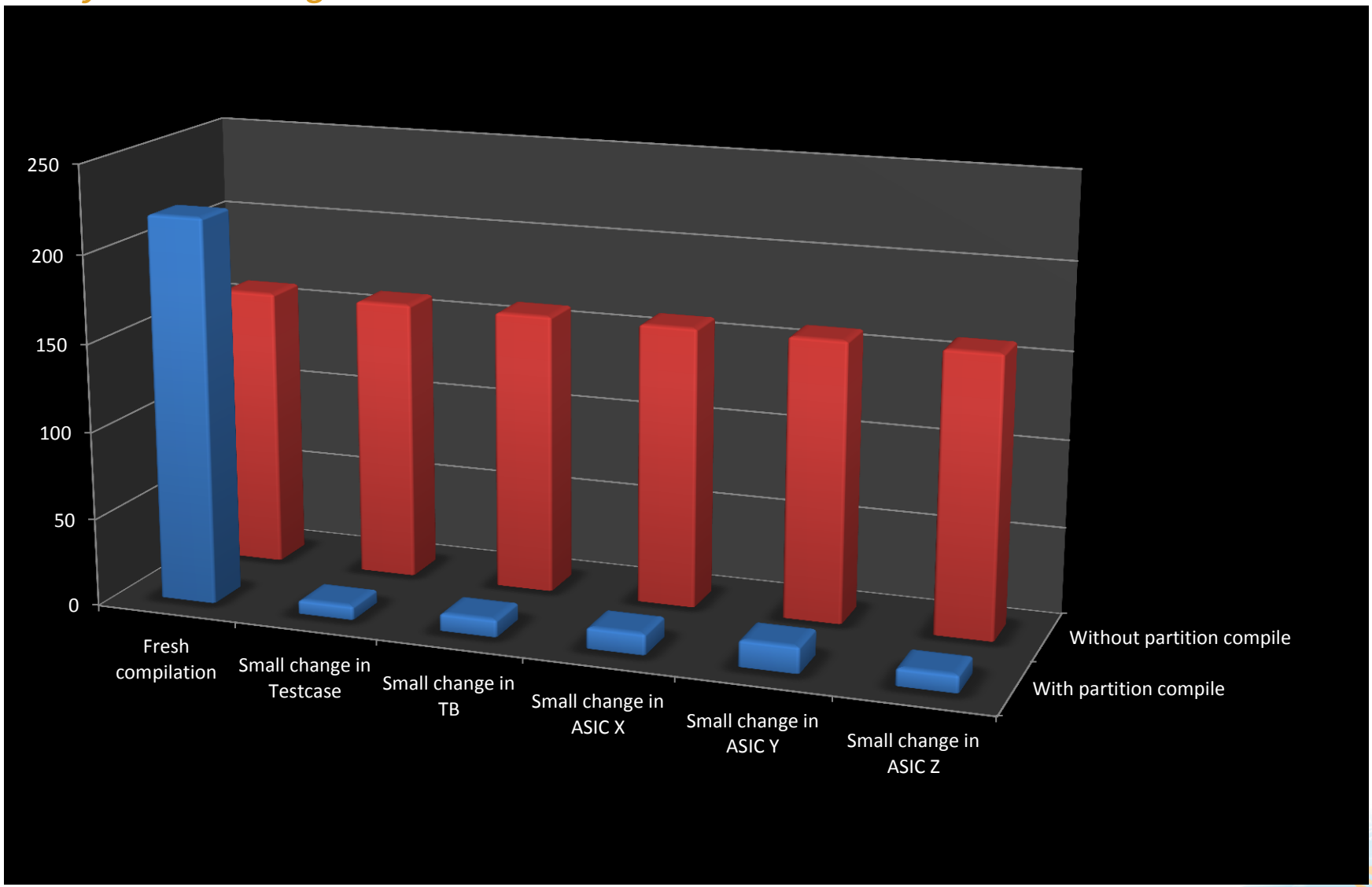
PARTITION COMPILE

Simulation Results-Compile time comparison (in mins) in System Config 2



PARTITION COMPILE

Simulation Results-Compile time comparison (in mins) in System Config 3



PARTITION COMPILE

Big disk space savings

- Big disk space savings through sharing of partition data.
- Generate the partitions by running the first test.
- Use ***-partcomp_dir=<dir_path>*** option to specify a directory where to store the common partition data.

Subsequent tests can reuse the partitions by already created by the first test using the ***-partcomp_sharedlib=<dir_path>*** *option.*

PARTITION COMPILE

Big disk space savings through sharing of partitions

We had 25 testcases in our testplan.

Disk space consumption for 25 testcases

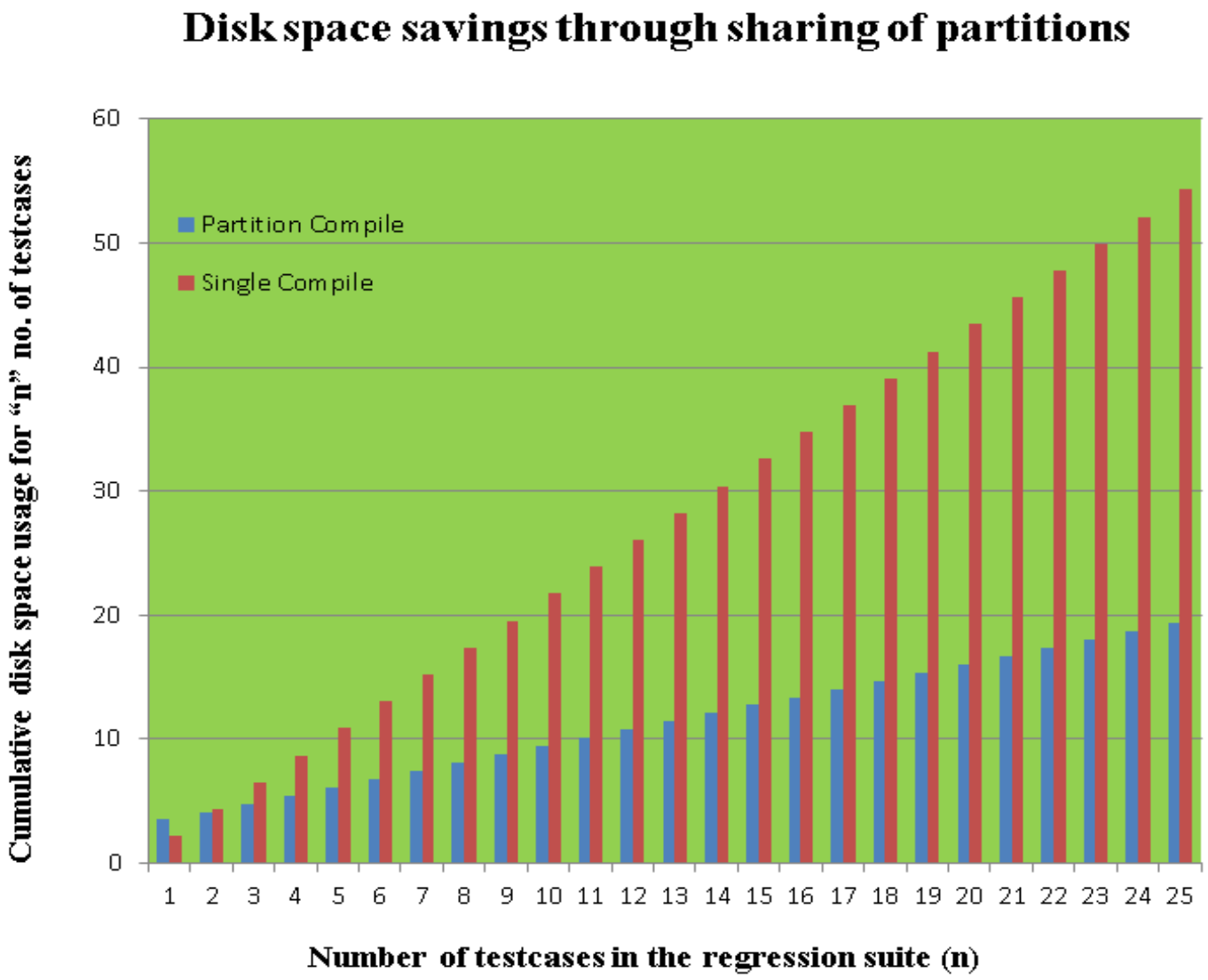
25*(simv) + simv.daidir + csrc + Global Partition + Local Partition → ***(Partition compile)***
25*{simv + simv.daidir + csrc} → ***(Single Compile)***

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	
csrc	9.6M	1.1G	
Local partition	20M	0	Partition database reused for all 25 testcases in partition compile
Global Partition	2.8G	0	
TOTAL DISK USAGE for 25 testcases	19.33G	54.3G	~65% savings

~65 % savings in disk space with partition compile compared to single compile

PARTITION COMPILE

Big disk space savings through sharing of partitions



VCS techniques to improve RUN TIME performance

- Radiant technology
- Save & Restore
- VCS Multicore Technology
- Dump Scope Control
- Interactive rewind

Radiant Technology

- Brings in certain code based optimizations where it internally generates Meta code for certain constructs to bring runtime improvement.
- Some of these optimisations include
  - Function inlining
  - Task inlining
  - Loop unrolling
  - Common subexpression elimination
  - Dead code (unreachable code) elimination
- Enabled by using ***+rad*** option

Radiant Technology

Simulation Results

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%

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

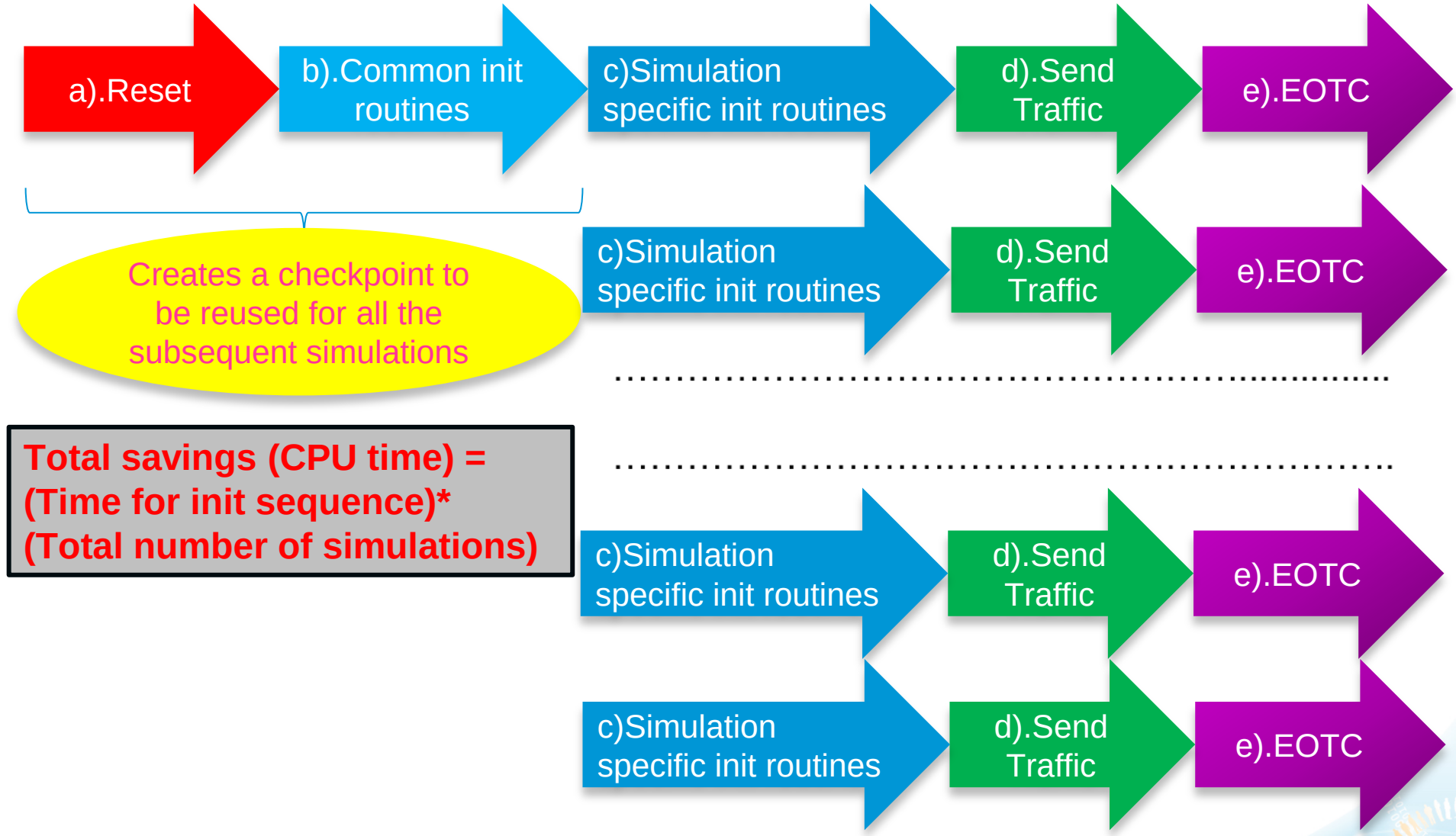
Radiant Technology

Known Limitations

- Dumping VCD files limits rad optimizations, so you won't get the full runtime performance benefits.
- Can expose race-conditions in the design
  - Random stability not guaranteed
  - Code coverage results may vary 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).
- Back-annotation of SDF files is not supported.

SAVE AND RESTORE

How does it help??



SAVE AND RESTORE

Simulation Results

SAVE & RESTORE – Run time comparison

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	

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.

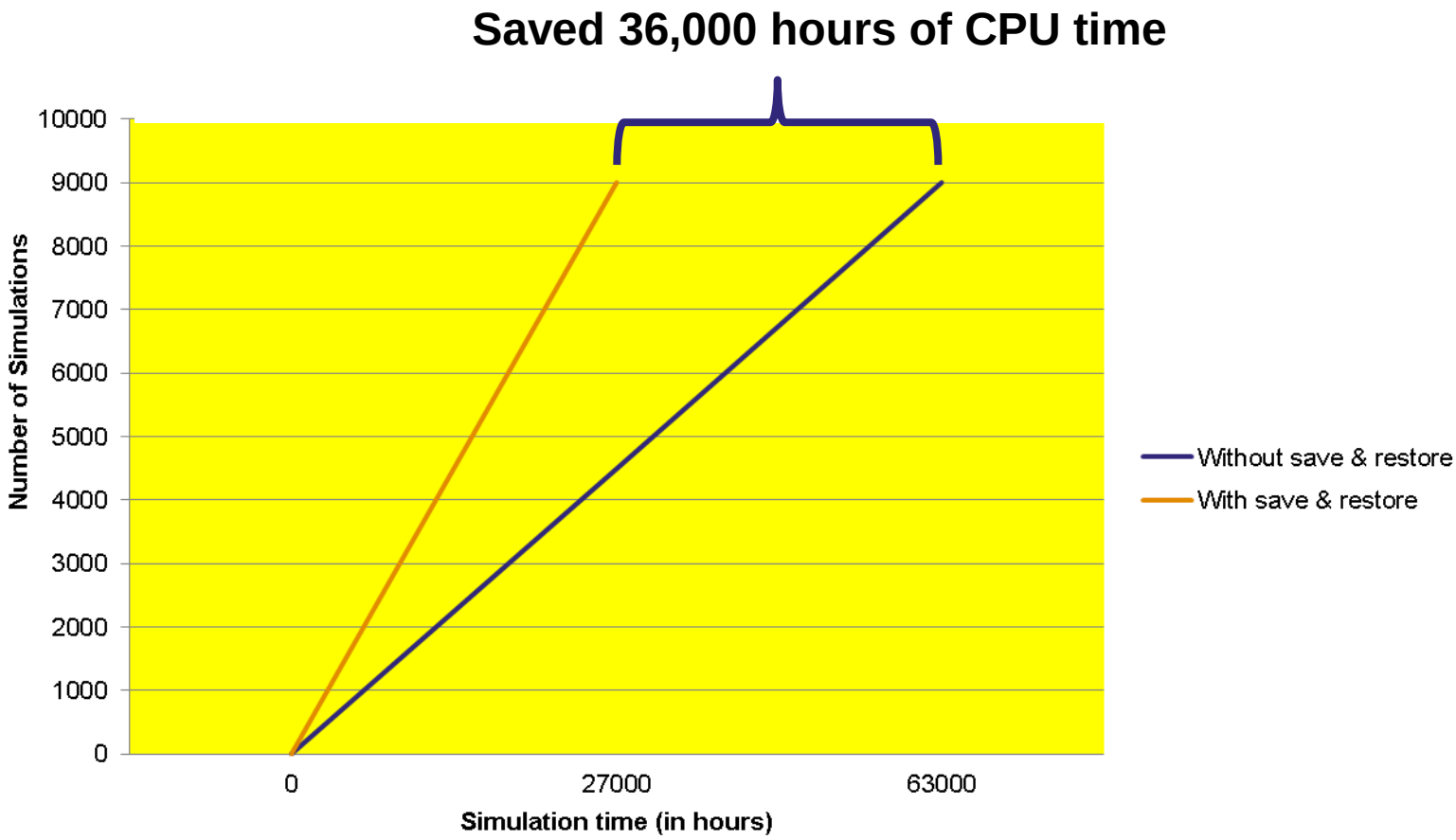
Assuming that we are getting 100 LSF slots for ProjectX (dictated by LSF scheduler)

Total savings = 36,000/100 hours

= 360 hours (15days)

SAVE AND RESTORE

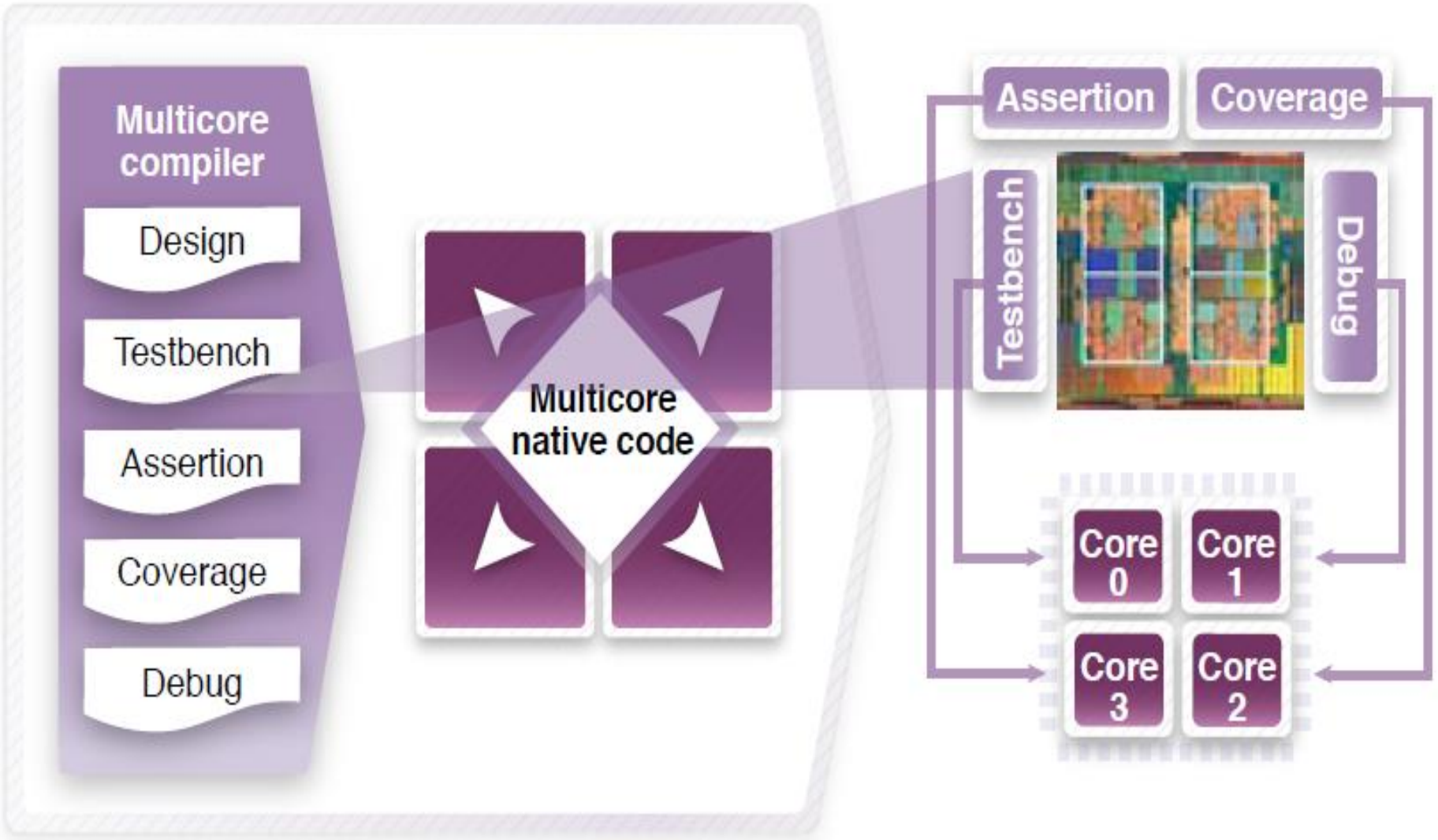
Simulation Time Comparison



VCS Multicore Technology

- Leverages the computing power of multiple processors in one machine to improve simulation TAT.
- Two robust Use Models
  - Design level parallelism (DLP)
  - Application Level Parallelism (ALP)

Application Level Parallelism



ALP

Use Model

ALP technology	Compile Switch
Assertions	-parallel+sva[=*NCORES*]
Functional Coverage	-parallel+fc[=*NCORES*]
Toggle Coverage	-parallel+tgl[=*NCORES*]
VPD Dumping	-parallel+vpd[=*NCORES*]
FSDB Dumping	-parallel+mtfsdb

Can also be enabled at runtime so user can choose parallel or serial simulation without having to recompile.

ALP

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

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%

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

Dump Scope Control

Controlling what design scope(s) to dump improves the simulation time and also reduces the size of the VPD created.

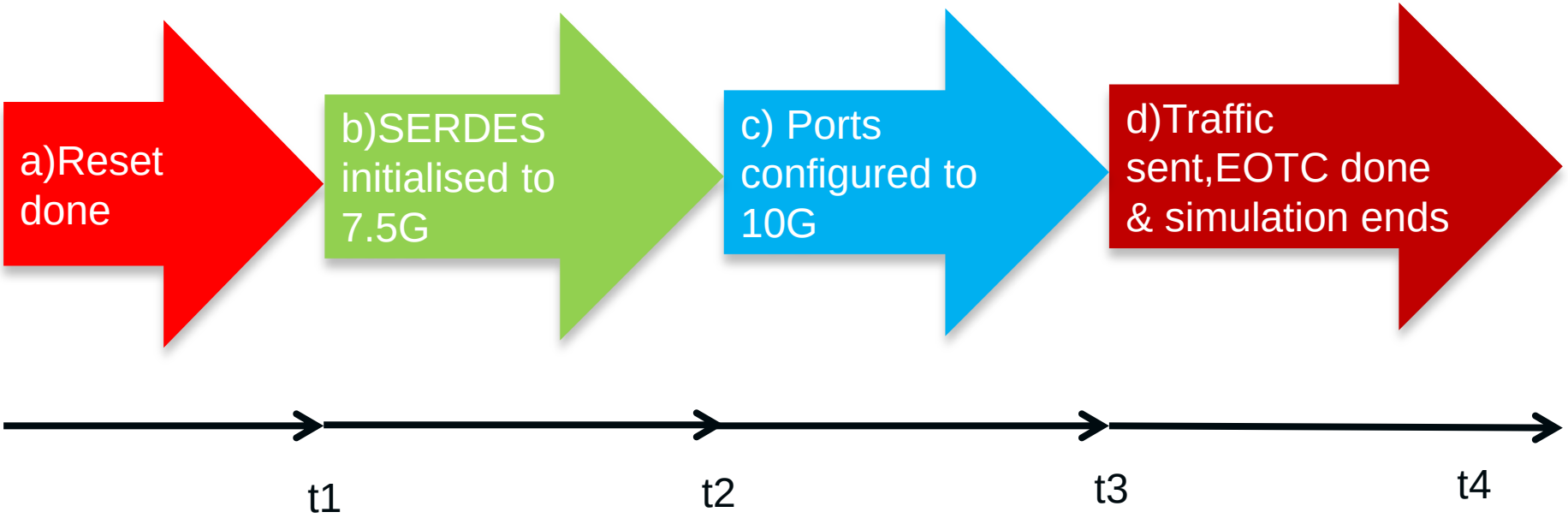
Savings in Disk Space and Simulation time using Dump scope control

System Configuration	Scenario	Simulation Time	VPD size	Reduction in Simulation time	Reduction in VPD Size
SC1	Dump all the signals	150 mins	9 GB	40%	88%
	Dump only signals in specified scope	90 mins	1 GB		
SC2	Dump all the signals	180 mins	11 GB	38%	89%
	Dump only signals in specified scope	110 mins	1.2 GB		
SC3	Dump all the signals	220 mins	12 GB	31%	87%
	Dump only signals in specified scope	150 mins	1.5GB		

Interactive Rewind

Improving Debug TAT (Turn Around Time)

Interactive Rewind



a)Reset	1hr
b)Serdes initialisation	8 hrs
c)Port configuration	2hrs
d)Traffic and EOTC	7hrs

Total simulation time=18hrs

Interactive Rewind

After running for 18hrs my simulation FAILS☹

a)Will my simulation PASS if

i)SERDES is programmed at 10G???

ii)If the Port Speed were 40G???

Can we save debug cycles by moving back to t1,t2 or t3 ???

SOLUTION-----

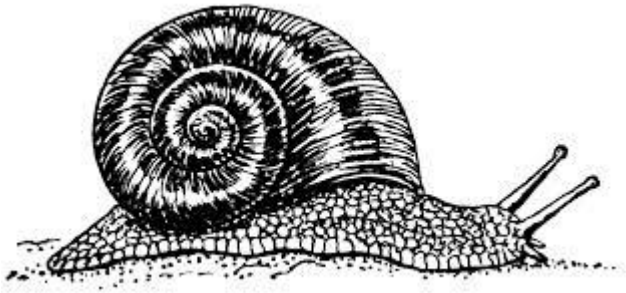
Use “INTERACTIVE REWIND” feature by creating multiple checkpoints at t1, t2, t3 etc……

Interactive Rewind

Advantages

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

Results



Results

- Compare
  - ***Absolute bare bone simulation*** TAT from compile to end of run
  - ***Optimized flow*** using the aforementioned optimization techniques.

    Parallel compilation

    Fast compilation

    Incremental compilation

    Partition compile

    Radiant technology

    ALP
- Comparison in two flavours
  - ***1st ITERATION*** *is a fresh run with compilation done from scratch*
  - ***2nd ITERATION*** *is a run in the same directory*

    (So we already had our database compiled into a binary version by the previous compilation → a prerequisite for some optimizations e.g. incremental compile & partition compile)

Results

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

System Config	Total TAT without any optimisations (bare-bone approach)	Total TAT using the optimised 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%

Results

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

System Config	Total TAT without any optimisations (bare-bone approach)	Total TAT using the optimised 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%

Conclusions and Looking Forward

Conclusions

- Integration of the VCS optimization techniques in our flow helped in optimizing
  - COMPILE Time
  - RUN Time.
  - Disk usage.
- No extra investment in terms of Hardware resources and Software licensing costs.

Future Work

- ***Multi-threaded simulation*** Design-level parallelism (DLP) is achieved by partitioning a design and simulating it on multiple processor cores in parallel.
- ***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.
- ***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.
  - Precompiled IP (PIP) flow gives fast elaboration/compile times when a Precompiled IP is shared across many targets/tests).
  - Also sharing of Precompiled IP helps in Disk-space reduction.

Thank You