SPEEDING UP VERILOG GATE-LEVEL SIMULATION WITH BI-PARTITIONING

Lilian Janin, Doug Edwards

Advanced Processor Technologies Group
School of Computer Science, The University of Manchester, UK

lilian.janin@manchester.ac.uk (Lilian Janin)

Abstract

Iterative design methodologies based on a simulation-debugging-update cycle form the basis of Verilog design development. An automated flow to speed up iterative design cycles is presented here. Compilation speedup is obtained by partitioning the circuit in order to exploit the locality of code updates during a typical iteration, in order to recompile only the modified parts of the design. Simulation speedup is obtained by interfacing multiple instances of the same simulator together through a cosimulation interface, either on single-core or dual-core computers. Particular care is taken in the design of the cosimulation interface to ensure the same accuracy as during a single-kernel simulation.
A smartcard circuit embedding an asynchronous ARM processor is used as a demonstrator. The speedup is analysed on both single and dual core machines with gate-level simulation. An unexpected result is that even on a single-core computer, in some circumstances, partitioning a simulation and simulating both parts simultaneously leads to some speedup in spite of the losses due to the cosimulation interface. During the iterative design cycle experiments, the main result is a 30% speedup achieved with all the simulators on a single core and 50% speedup on dual-cores.

Keywords: Verilog, Cosimulation, Speedup.

Presenting author's biography

Lilian Janin is a Research Associate in the School of Computer Science at the University of Manchester, where he received his Ph.D. degree in 2004. His research interests are in simulation and visualisation of large asynchronous systems. He has been working since 2000 ond the Balsa asynchronous simulator and visualisation system. His work is now mainly focused on a co-simulation debugging environment for heterogeneous synchronous-asynchronous circuits.

1. Introduction

Most Verilog design flows are based on an iterative cycle including gate-level simulation, usually in the form simulation-debugging-update. This iterative cycle is an important part of the development process. Our approach aims at shortening the design cycle of large systems described in Verilog by speeding up this iterative simulation stage. This stage is characterised by short simulation lengths (bugs usually appear early on) and almost identical behaviour from one cycle to the next. This second property is due to iterative code changes that are usually localised to a small region of the source code.

Parallel simulation has been extensively researched to accelerate simulations [1, 2, 3]. However, the two enounced properties of iterative simulation lead to a new optimisation opportunity: short simulations of large designs imply that the time spent by the simulator to compile the Verilog code (sometimes to bytecode, sometimes to C and then machine code) is a large proportion of the total simulation time. Small localised changes imply that most of the design does not need recompiling from one iteration to the next. Some Verilog simulators already exploit this by offering incremental compilation [4] with file or module granularity, but others do not, and this paper shows that all simulators benefit from the following idea: partitioning the Verilog design to take advantage of localised code changes and recompiling only parts of the design at each iteration.

The difficulty raised by this approach is that the multiple parts need to be reconnected appropriately during the actual simulation. One way to do that is by using a cosimulation interface. Here we present a discrete event cosimulation interface specifically designed for handling communications between two processes running on a single processor and minimising the cosimulation interface overhead. We show that, in this particular situation, it is possible to design a cycle-accurate cosimulation interface with very low overhead, a non-obvious achievement [5] giving us the same accuracy as the original single-threaded simulators.

The cosimulation interface originally described for single-core experiments is then used as-is to measure the speedup obtained on dual-core processors, as those are becoming increasingly common in desktop computers.

Related work includes various researches intending to make simulation faster. Distributed discrete event simulation has gained extensive research focuses with diverse approaches depending on their different emphasis on conflicting goals such as simulation speed, accuracy and flexibility [1, 2, 5, 6].

Our automated design flow is presented in Section 2. Section 3 describes the time-accurate cosimulation interface. Results and speedups are analysed in Section 4, concluding on the 50% design cycle speedup obtained on today's dual-core desktop computers.

2. Automated design flow

Although we emphasise in this paper the compilation and simulation speedup benefits of our research, another aspect of our framework is to provide a transparent generation of cosimulation interfaces with links to a visualisation system for enhanced debugging. A graphical integrated environment facilitates the designer's task of learning this new set of tools. Even in the scope of this paper, it is worth mentioning these features, as they facilitate the integration of these new tools in designers' traditional flows and makes their adoption relatively easy.

Our proposed design flow is shown in Fig. 1. The traditional compile-simulate-debug-update design cycle is augmented by a setup stage intended to prepare the cosimulation environment (partitions and interfaces). This cosimulation setup stage takes a Verilog description of the design as input and generates two Verilog partitions via a GUI, with an interface linking each partition to the other. The two partitions are then compiled and simulated concurrently, the optional simulation traces are merged, and the subsequent debugging and code update stages are executed identically to those from a traditional design cycle. Typical design cycles do not change the structure of the circuit, and this flow therefore requires the same amount of manipulations from the designer as traditional flows, while exploiting concurrency. The setup stage needs to be repeated only when the design undergoes structural changes that modify the cosimulation interfaces. In this case, it is fully automated and should appear transparent to the designer.

The cosimulation setup stage can be decomposed into five phases (Fig. 2): analysis of the circuit's structure, generation of the hierarchy of components in an internal format (called GALSA - Globally Asynchronous Locally Synchronous or Asynchronous), partitioning of this set of components, generation of the cosimulation interfaces, and an optional pruning of the Verilog source code for the two partitions.

Most of these stages are fully automated, but some of them usually benefit from a minimal set of user decisions (such as choosing the partition), while other stages could expose useful information to the designer through a few manipulations.

2.1. Circuit structure analysis and generation of the GALSA structure

The circuit structure analysis consists of a Verilog parser, which is used to discover the design's modules and their connections. From this, a graph is generated where each node contains some information about one Verilog module, and where the edges represent the

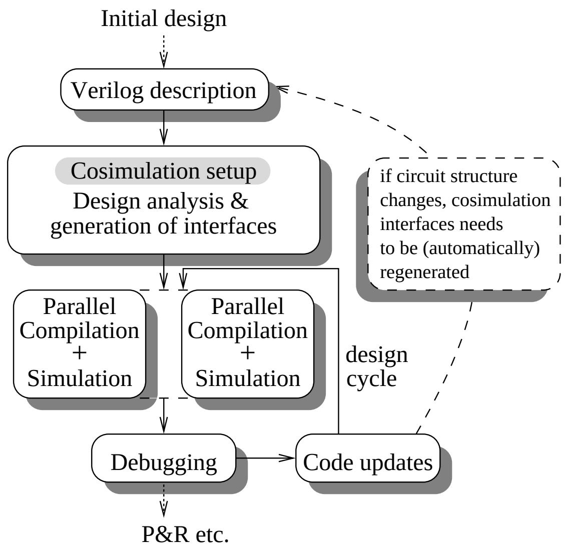
Initial design
Verilog description
Cosimulation setup
Design analysis & generation of interfaces
if circuit structure changes, cosimulation interfaces needs to be (automatically) regenerated
Parallel Compilation + Simulation
Parallel Compilation + Simulation
design cycle
Debugging
Code updates
P&R etc.

Fig. 1 Design flow iteration

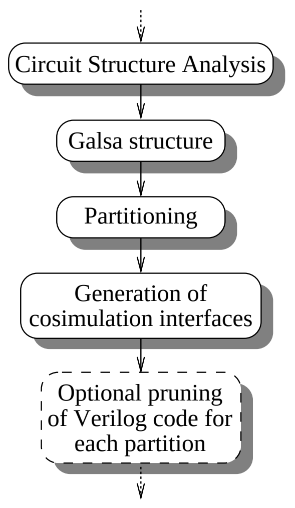
Circuit Structure Analysis
Galsa structure
Partitioning
Generation of cosimulation interfaces
Optional pruning of Verilog code for each partition

Fig. 2 Cosimulation setup stage

interconnect. The graph is stored in our internal GALSA format, able to handle modules in various languages. The Galsa format was designed to describe asynchronous interconnect, and has therefore specific support for asynchronous channels and protocols. During the analysis of the Verilog source code, separate wires can be gathered as channels if they appear to describe an asynchronous channel with a request, acknowledge and data. The first supported language was the asynchronous HDL Balsa [7].

2.2. Partitioning

Partitioning correctly a circuit is difficult. The problem of minimising the partition cut while balancing the simulation load is known to be NP-complete [8], and partitioning algorithms have been extensively studied without really emerging in the HDL world. Here however, the problem is considerably simplified due to the coarse granularity at the Verilog module level.

One important decision in this project was to bypass the traditional research on partitioning algorithms, for two reasons: firstly, the circuit we had in mind as a demonstrator (see Section 4) is made of one processor roughly using half of the resources (in terms of compilation time, simulation time and number of transistors) and the rest of the circuit using the other half. This rough partitioning, which didn't require any advanced analysis, is very much what engineers happen to do in practice, and still produced exploitable results. Secondly, and most importantly, we discovered that no partitioning was ever ideal for all stages and all compilers/simulators: one partition can be perfectly balanced for one Verilog simulator, and very unbalanced for another Verilog simulator, or one partition can produce two balanced compilation stages, but very unbalanced simulations, etc.

2.3. Optional pruning of partition code

Most Verilog simulators do not allow the designer to specify a top-level module. In this situation, they will consider as top-level all the modules that are not instantiated in any other modules, thus wasting a lot of resources. With our bi-partitioned system, each partition gets almost half of its modules unused. In order to save resources, the tool can prune every unused Verilog module and bundle the result in one file or directory. As detailed in the Results section, this brings up to 144% compilation speedup.

When activated, this stage is considered as part of the compilation process and is included in the short design cycle shown in Fig. 1 because it needs to be re-processed after each code update.

3. Time-accurate cosimulation interface

It is very simple to design an efficient cosimulation interface where simulation delays at the interface are not strictly respected, therefore leading to distributed simulation results being different from those obtained from a single-kernel simulation. An accurate scheduling of events across a cosimulation interface is more complex to achieve and requires advanced access to the simulator's event queue, which is not always available or obvious, as internal data can be hidden from the user.

A simple cosimulation technique and its accuracy shortcomings are illustrated in Fig. 4 and 5. In this example, a Verilog description is partitioned into two communicating parts synchronised at every simulation time step by using a pseudo clock. Each partition has one input and one output, but the interface scales obviously to m inputs and n outputs.

The output events are properly sent as soon as they happen. However, the reception of these sent messages is problematic as they should interrupt the target's verilog

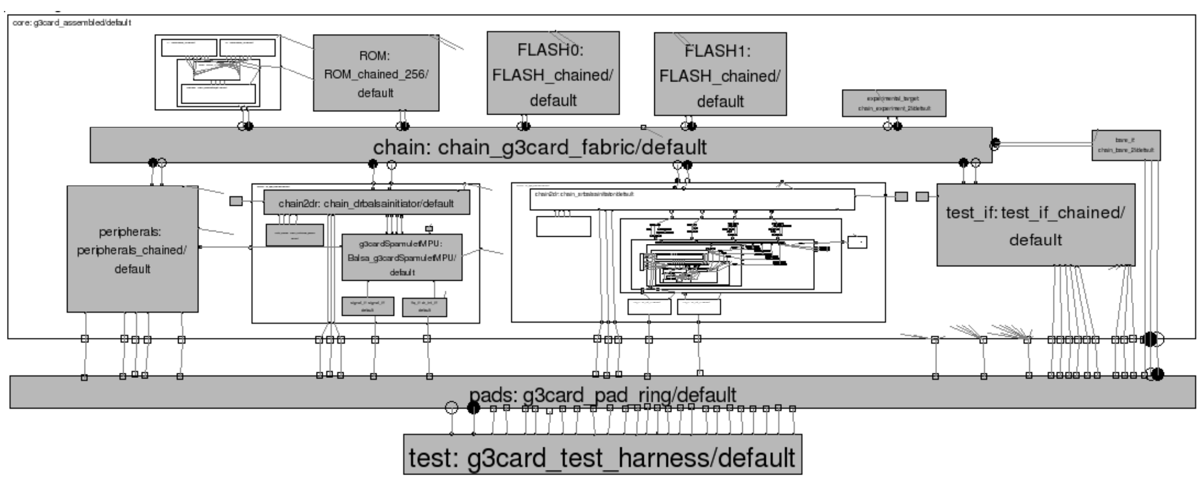
core: g3card_assembled/default
ROM: ROM_chained_256/ default
FLASH0: FLASH_chained/ default
FLASH1: FLASH_chained/ default
chain: chain_g3card_fabric/default
peripherals: peripherals_chained/ default
chain2dr: chain_drbalsainitiator/default
g3cardSpamuleMPU: Balsa_g3cardSpamuleMPU/ default
test_if: test_if_chained/ default
pads: g3card_pad_ring/default
test: g3card_test_harness/default

Fig. 3 GALSA graphical interface showing the architecture of the SPA-smartcard project

simulation at the proper simulation time or before (in which case they can then be rescheduled to the proper timestep), but not after. In this simple technique, the interruption of the simulation for receiving the messages is triggered by a pseudo clock calling receive_event at each time step. Receive_event sequentially receives each message sent by the other partition until a “clock tick” message indicates that subsequent events belong to future timesteps. This solution has two problems: first, it is not possible to know or specify when during the time step the function receive_event will be called. It could happen at the beginning of the time step, before other events are processed, or at the end of the timestep. In the first case, the messages generated by the other partition during this timestep will only be received at the next cycle; In the latter case, the input messages received might generate outputs to the other partition during the same timestep, which will only be processed in the next timestep by the target.

Basically, this technique cannot be used to transfer events with zero delay without introducing skew.

Obviously, what is needed is a way to receive events, process them, and repeat as many times as necessary within the same timestep; and to advance to the next simulation timestep only when we are certain that all partitions have processed all their events for the current timestep (we are in the conservative cosimulation mindset). For that, we need to be able to call our receive_event routine at the very end of each simulation timestep, and have it being triggered again at the end of the same timestep if we rescheduled some new intermediate events. Fortunately, similar requirements were expressed for hardware-software codesign years ago and have been integrated and refined in the Programming Language Interface (PLI) of Verilog simulators. PLI, now in its version 2, also called Verilog Procedural Interface (VPI), offers the possibility to schedule callbacks associated to various events: value change, simulation time change, last event in a time step queue, etc.

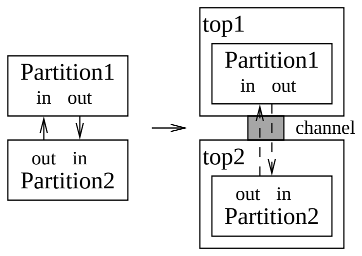

Fig. 4 Simple cosimulation i/f problem

```
module top1
  wire in, out;

  always @out
      /* non-blocking event submission */
      send_event (“out=”, out);

  always begin
      #1    /* pseudo-clock */
      /* non-blocking event submission */
      send_event (“clock tick”);

      /* read input events and block
         until “clock tick” message    */
      read_events (in);
  end

  Partition1 P1 (in, out);
end module
```

Fig. 5 Simple interface code

The cosimulation interface designed here is based on the VPI callbacks cbReadWriteSynch and cbAtEndOfSimTime, which are VPI events called at the beginning and end of simulation time steps. In our implementation, these events replace Fig. 5’s pseudo clock; and the inter-partition communication mechanism is based on a standard FIFO unix pipe for transporting the messages, as an implementation of Fig. 5’s send_event and read_events routines. The end-of-timestep callback is rescheduled at the end of the current timestep when read_events happens to schedule new intermediate events.

4. Results / Performance increase

The speedup due to partitioning is evaluated with five Verilog simulators, some commercial ones and some freely available. The results are reported anonymously due to the license agreements of commercial simulators. The Verilog test case is a smartcard chip embedding an asynchronous ARM v5 core [9] (architecture shown in Fig. 3). This is a reasonably large design reaching a million transistors. The results presented in this paper are based on a partitioning of this design into two parts: part1 is the entire circuit except the processor (i.e. on-chip network, memory, uart, I/O pads and behavioural test harness) and part2 is the processor. This coarse partitioning does not achieve a balanced load in most cases but has been chosen for its similarities with a typical real life partitioning. Furthermore, it is impossible to determine a partition that is balanced for every simulator. Even when considering a single simulator, as the results will show, part1 is dominant in most compilations while part2 takes over during simulations. All the tests are run on an Athlon 64 dual-core processor 4200+ 2.2GHz. During single-core experiments, the processes were forced into a specific processor core using the 'taskset' command.

Each simulator comes with different standard capabilities, reported in Tab. 1. Incremental compilation is probably the most important feature for a simulator when considering iterative design cycles: instead of requiring a full design recompilation, typical localised code changes can be simulated and debugged after the recompilation of only a single file or a single Verilog module.

A second important feature of simulators is the ability to specify a top-level module. When available, this feature allows the Verilog compiler to extract the relevant Verilog modules, leading to faster compilation and simulation. When this feature is not available, all the unused modules get instantiated as top-level modules, thus wasting a lot of resources.

Broadly speaking, our partitioning and pruning flow brings the benefits of incremental compilation and top-level specification to the simulators devoid of these features.

4.1. Compilation speedup

The analysis of the compilation speedup is made of two parts: compilation of the whole design and incremental compilation of localised code changes. Although we are interested in measuring the effects of our system during an iterative process, the full compilation gives an upper bound to the iterative compilation time.

Tab. 1 Simulators characteristics

Simulator:	1	2	3	4	5
Incremental compilation	file-based	file-based	module-based	No	No
Can specify top level	No	Yes	No	Yes	No

4.1.1. First iteration: Full design compilation

The time taken for the compilation of the entire design by each simulator is reported in Tab. 2. The first row corresponds to the non-partitioned design. The second row details the time taken to compile each part of the partitioned circuit and the cumulated time. The single-core speedup is the ratio between the time taken to compile the whole non-partitioned design (row1's non-partitioned compilation time) and the cumulated time for parts 1 and 2. The dual-core speedup is the ratio between the non-partitioned compilation time and the largest compilation time between part 1 and part 2.

We could imagine a "dual-core possible speedup", which would assume a load-balanced compilation of parts 1 and 2 based on the reported cumulated time (i.e. the compilation of part 1 and part 2 would take the same amount of time, equal to half the cumulated time). Ths would give a "dual-core possible speedup" equal to twice the reported single-core speedup. Although this speedup might be reached by a balanced partition, it would be quite an artificial figure.

The reported single-core speedup is a surprising outcome of this research, with the unexpected 53% speedup reached with one of the simulators on a single core. The setup is as expected: compiling the full design takes 90s. The design is then partitioned into two parts. Some modules such as the low-level cells are duplicated, as they are required in both parts, and a cosimulation interface is added to each part. In spite of these additions, the compilation of part1 and part2 takes 43s and 16s respectively, totalling less than the full compilation. Other simulators' compilers are behaving more logically.

The dual-core speedup, averaging 50%, is too irregular to draw definite conclusions. As we will see later, the results combining compilation and simulation are more consistent.

The timings reported in Tab. 2 for the compilation of parts 1 and 2 are including the optional pruning of Verilog code for each partition described previously. Tab. 3 shows the effects of this option on compilation time.

On the simulators where the top-level module cannot be specified, the speedup due to pruning ranges from 30% to 144%.

4.1.2. Iterative compilation: Localised updates

During a typical design cycle, the debugging process

Tab. 2 Full compilation speedup

	1	2	3	4	5
non-partitioned compilation	62s	31s	86s	86s	90s
compilation part1 + part2	42s +29s =71s	20s +16s =36s	81s +41s =122s	25s +58s =83s	43s +16s =59s
single-core speedup	0.87	0.86	0.70	1.04[1]	1.53[1]
dual-core speedup	1.48	1.55	1.06	1.48	2.09[1]

1. super-linear speedup

Tab. 3 Pruning speedup

	1	2	3	4	5
part1 + part 2 compilation without pruning	65s +49s =114s	22s +22s =44s	83s +76s =159s	25s +58s =83s	90s +54s =144s
part1 + part 2 compilation with pruning	42s +29s =71s	20s +16s =36s	81s +41s =122s	25s +58s =83s	43s +16s =59s
speedup due to pruning	1.61	1.22[1]	1.30	1.00[1]	2.44

1. Pruning integrated in simulator (top-level can be specified)

leads to small localised changes to the source code, which is then recompiled, simulated and debugged again. Here, we look at the variations in compilation time after such small localised changes are applied to our smartcard design. The experience is based on two code changes: change 1 is in a one line change in a small leaf module used at six others places in a small file; change 2 is a one line change in a large module located in a large Verilog file (300 kB). The corresponding recompilation timings are reported in Tab. 4.

The reported results show an average 70% speedup with the partitioned design with the simulators already implementing incremental compilation, and 2x to 3x speedup with the two other simulators. This experiment does not exploit any parallelism, as only one part needs recompiling at any time, and this speedup is therefore reached on a single core. Dual-cores might get extra benefits when code changes are distributed over both parts, but should be rare in practice.

4.2. Simulation speedup

It was expected that the compilation of parts of the circuit would be faster than the compilation of the whole circuit, leading to some speedup at compilation. However, cosimulation communications often add so much overhead at runtime that they render the whole process useless. Here a surprise awaits us again with the single-core experiment(Tab. 5). Again, simulator 5 unexpectedly performs 16% faster with two parts simulated simultaneously on the same processor core than one single simulation process. But another important achievement is the single-core speedup around 1.00 for all the other simulators, meaning that the cosimulation interface does not add any noticeable communication overhead.

Tab. 4 Iterative compilation speedup

	1	2	3	4[1]	5[1]
non-partitioned compilation: change1/ change2	50s/ 50s	20.5s/ 20.5s	25s/ 25s	86s/ 86s	90s/ 90s
change 1 in compilation part1	34s	10s	22s	25s	43s
change 2 in compilation part2	24s	11s	14s	58s	16s
average iterative compilation speedup	1.72	1.95	1.39	2.07	3.05

1. No incremental option (same results as full compilation)

An analysis reveals that, although they are synchronising at every time step, the two simulation processes behave very well, switching quickly without wait states, and never waiting for each other in sleep mode. 100% CPU is constantly used. This is shown in Tab. 6, were “sim part1 alone” and “sim part2 alone” are showing the results when no synchronisation is needed, in order to show the load distribution. These figures are obtained by first recording the communications during a normal cosimulation and then replaying one side of the cosimulation at a time, taking its inputs directly from the recorded file.

The dual-core results are less impressive, with 20-30% speedup. This is mainly due to the unbalanced load. The communication overhead in dual-core is around 20%, and a dual-core speedup of 1.80 could therefore be achieved.

4.3. Total speedup

This section sums up the compilation and simulation speedups reported previously and groups them together to produce a useful speedup estimate during a typical design cycle (Tab. 7).

It has been reported previously that a partitioned design was getting compiled and simulated faster than the non-partitioned design with one specific simulator. However, this is an isolated case. The core result of this research is the 30% speedup obtained with every simulator when considering an iterative design cycle. This is explained by the fact that only one part of the partitioned design needs to be recompiled after localised changes, and is made possible by the extremely low

Tab. 5 Simulation speedup

	1	2	3	4	5
full simulation	36s	110s	82s	112s	151s
cosim single-core	37s	112s	81s	111s	130s
cosim dual-core	30s	86s	63s	86s	122s
single-core speedup	0.97	0.98	1.01	1.01	1.16
dual-core speedup	1.20	1.28	1.30	1.30	1.24

Tab. 6 Communications overhead

	1	2	3	4	5
sim part1 alone	11.5s	37s	27s	39s	24s
sim part2 alone	24.5s	74s	53s	72s	105s
single-core overhead	1s	1s	1s	0s	1s
dual-core overhead	5.5s (22%)	12s (16%)	10s (19%)	14s (19%)	17s (16%)

overhead of the cosimulation interface on a single core.

With a dual-core configuration, 50% speedup is commonly obtained during iterative design cycles, and 40% speedup is reached for a full recompilation of the entire (but partitioned) design. Although not astounding, it might come for free on today's multi-core computers.

Sometimes extra speedup on dual-core is actually reachable due to the overlapping of the beginning of sim2 with the end of comp1 (or vice versa) but is not included here due to the small gain (max. 2s).

5. Conclusion

It has been shown that, independently of the Verilog simulator, 30% simulation speedup can be achieved simply by partitioning a design into two parts. The speedup comes from the fact that a typical iterative design cycle is based on localised code changes, which then require the recompilation of only one part of the partitioned design. During the simulation, the communications between the two parts need to be handled appropriately. This is done here using a classical cosimulation interface carefully designed to ensure the same cycle-accuracy as during a single-kernel simulation.

The partitioning and the generation of the cosimulation interface are made as transparent as possible by our automated GALSA system. The cosimulation can also be used on dual-core configurations to reach 50% speedup. In today's dual-core desktop computers, this speedup is obtained almost for free, an important fact given the constant advances in hardware developments.

Tab. 7 Total iterative speedup

	1	2	3	4	5
non-partitioned comp+sim (full / iterative)	98s/ 86s	141s/ 130.5s	168s/ 107s	198s/ 198s	241s/ 241s
single-core comp+sim (full / iterative)	108s/ 66s	148s/ 122.5s	203s/ 99s	194s/ 152.5s	189s/ 159.5s
dual-core comp+sim (full / iterative)	72s/ 59s	106s/ 96.5s	144s/ 81s	144s/ 127.5s	165s/ 151.5
single-core speedup (full / iterative)	0.91/ 1.30	0.95/ 1.27	0.83/ 1.28	1.02/ 1.30	1.28
dual-core speedup (full / iterative)	1.36/ 1.46	1.33/ 1.53	1.17/ 1.51	1.38/ 1.55	1.46

6. References

[1] P. Gerin, S. Yoo, G. Nicolescu, and A. A. Jerraya. Scalable and flexible cosimulation of SoC designs with heterogeneous multi-processor target architectures. *ASP-DAC*, pp. 63-68, 2001.

[2] T. Li, Y. Guo, S. Li, F. Ao, and G. Lio. Parallel verilog simulation: architecture and circuit partition. *ASP-DAC*, pp. 644-646, 2004.

[3] T. Watanabe, Y. Tanji, H. Kubota, and H. Asai. Parallel-distributed time-domain circuit simulation of power distribution networks with frequency-dependent parameters. *ASP-DAC*, pp. 832-837, 2006.

[4] V. K. Sundar, A. V. Ashish, and D. R. Chowdhury. Incremental compilation in the VCS environment. *International Verilog HDL Conference*, pp. 14-19, 1998.

[5] K.-H. Chang, W.-T. Tu, Y.-J. Yeh, and S.-Y. Kuo. Techniques to Reduce Synchronization in Distributed Parallel Logic Simulation. *Parallel and Distributed Computing and Systems*, 2004.

[6] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelly. Design of Embedded Systems: Formal Methods, Validation, and Synthesis. *Proceeding of the IEEE*, vol. 85 (3), March 1997.

[7] A. Bardsley, and D. A. Edwards. Balsa - An Asynchronous Hardware Synthesis Language. *The Computer Journal*, 45(1):12–18, January 2002.

[8] M. R. Garey, and D. S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman and Company, 1979.

[9] L. A. Plana, P. A. Riocreux, W.J. Bainbridge, A. Bardsley, J.D. Garside, and S. Temple. SPA - A Synthesisable Amulet Core for Smartcard Applications. *Async'2002*, pp. 201-210, 2002.