Automatic Partitioning for Multi-core HDL Simulation

Gaurav Kumar, Sandeep Pagey, Mohit Sinha, Manu Chopra
+91-120-3984000
{gauravk,pagey,mohits,mchopra}@cadence.com

Abstract- **Recent advances in processor and memory architectures have attracted many software applications towards parallelization using multi-threaded implementations. Multi-core HDL simulation is one such application which combines parallelism inherent in an HDL specification and multi-core CPU architecture to achieve parallelization. The main requirement of deploying a multi-core HDL simulation is to partition the design to be simulated into segments which can be executed in parallel. This is a manual process. There is no known algorithm to determine a design's partition which would lead to maximum performance gain using multi-core HDL simulation. An ideal partitioning algorithm would require analysis of dynamic behavior of the design's simulation. Dynamic analysis is applicable to that specific run and would require one or more simulation runs which is expensive and would consume compute resources. This paper presents a method and algorithm to partition a design using static parameters. This algorithm does not require any simulation runs. The static partitioning method presented in this paper provides a useful starting point for deployment of multi-core HDL simulation. Further tuning of the partitions can be performed using dynamic analysis of multi-core HDL simulation performed using the initial partition generated using static partitioning algorithm.**
The partitioning performed using algorithm presented in this paper has provided good performance gains when used with multi-core HDL simulation.

I. INTRODUCTION

Front-end RTL verification and post-synthesis gate level verification (with and without timing) are important phases of integrated circuit design flows. Many integrated circuit design houses mention that time consumed for RTL and gate-level verification is of the order of 70% to 80% of total design cycle [1]. HDL logic simulation is one of the most popular methods of performing RTL and gate-level verification. With increasing size of the designs, the total time consumed by verification is growing rapidly. In order to meet time to market requirements, advanced need to be made in HDL logic simulation so that it can simulate same number of design cycles in less time. Many advancements in HDL logic simulation tools and associated methodologies have been made in recent years to improve the performance of verification. Hardware accelerators and in-circuit emulators are being used to speed up the verification process, but suffer from the requirement of high up-front expenditure required to use them.

Parallel or multi-core HDL simulation is emerging as a promising technology to address the performance requirements for verification. Multi-core HDL simulation belongs to the broader topic of parallel discrete event simulation [2]. Survey reports [3, 4, 5] discuss many aspects and techniques for multi-core HDL simulation. A specific parallel simulation algorithm is illustrated in [6].

All these multi-core HDL simulation techniques rely on achieving a partitioning of the work to be done in such a manner that the work-load in all the partitions is as balanced as possible. The work-load of a partition refers to CPU consumption to execute the simulation activities for that particular partition. A partition would typically be executed on one thread; multiple partitions being executed on multiple thread in multi-core HDL simulation.

The partitioning can broadly be performed in two classes; application level partitioning (ALP) or design level partitioning (DLP). ALP refers to partitioning based on well-defined independent tasks to be performed as part of HDL simulation. The independent tasks can be run in parallel. For example, in an ALP partition, core simulation and waveform dumping can be run in parallel with core simulation partition passing relevant information to waveform dumping partition for it to do the task of writing the waveform database. Another example of ALP partition is to partition core simulation and functional coverage tasks. Core simulation partition passes relevant information to functional coverage partition for it to do the task of processing and dumping functional coverage. DLP, on the other hand refers to partitioning the design for core HDL simulation itself. In DLP, the design is partitioned in such a manner that core simulation tasks of various partitions can be performed in parallel. The HDL processes, assignments and primitives are divided into partitions to be executed in parallel. DLP requires various partitions to be synchronized regularly to ensure that HDL semantics are correctly followed during simulation.

In this paper we focus our attention on partitioning problem with reference to DLP based multi-core HDL simulation.

A number of partitioning techniques have been presented over the past few years [7, 8]. Many of them flatten the design hierarchy and then do the partitioning. Some of them perform dynamic partitioning analyzing data at run-time.

In this paper, we present an automatic partitioning technique which does not lose the instance boundaries contained in the design to be simulated. This partitioning technique takes into consideration various parameters of the design and performs a partition which is likely to provide balanced load during simulation. Various design parameters and design style are considered to perform the partitioning. A dynamic bin-packing algorithm is at the center of our partitioning technique.

This paper is organized as follows. Section II describes the need of good partitioning in a DLP based multi-core HDL simulation. Section III describes the partitioning algorithm in detail. Section IV presents experimental data of application of the partitioning algorithm. Section V concludes this paper.

II. Partitioning in DLP based Multi-core HDL simulation

In DLP, the design is partitioned in a fine-grained manner such that HDL processes, assignments and primitives are divided across partitions. These partitions simulate their content in parallel. The partitions must be synchronized to ensure that the simulation follows the HDL semantics and produces correct simulation results by way of correct ordering of events. Fig. 1 provides a high-level picture of a DLP based multi-core HDL simulation.

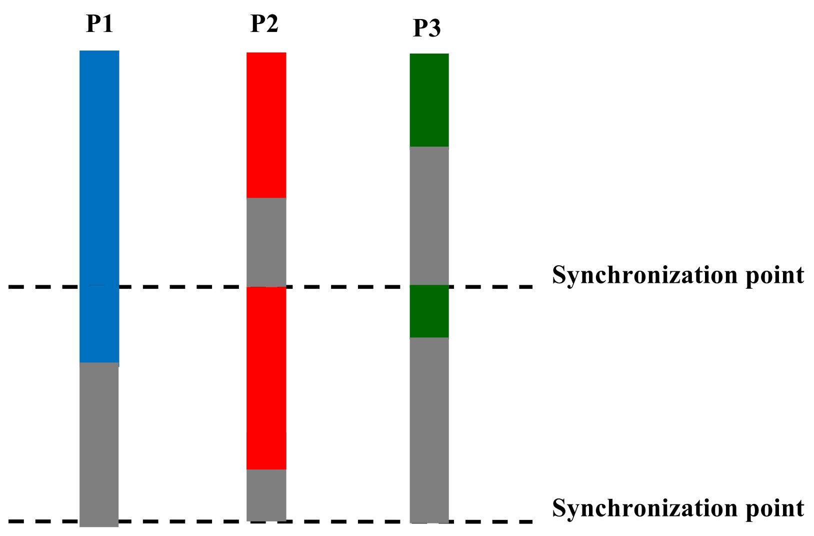

Fig. 1 : DLP based multi-core HDL simulation

Fig. 1 depicts a design divided into three partitions based on DLP. Each partition performs tasks related to HDL simulation for HDL content belonging to that respective partition before arriving at a synchronization point. The colored boxes indicate the useful work done by a particular partition and grey boxes indicate the time that a partition has no work to do before hitting a synchronization point. The grey boxes indicate the imbalance inherent in the partitioning. Usually, the grey boxes indicate wastage of CPU for that partition. A multi-core HDL simulation achieves best performance gains when the grey boxes are gotten rid of or minimized as much as possible.

A partitioning algorithm, in order to be effective, partitions the design in such a manner that the grey boxes are minimized during simulation, thereby achieving maximum performance gains from multi-core HDL simulation.

The next section provides details of our partitioning algorithm which performs partitioning at instance boundaries and attempts to distribute workload equally across partitions.

III. Partitioning Algorithm

In this section we discuss our partitioning algorithm in detail.

HDL designs specified at RTL abstraction or post-synthesis gate level designs have inherently parallel specifications in terms of processes, assignments, sequential elements and primitive gates. The partitioning algorithm equally distributes load across partitions based on these design parameters. Even though these design aspects can be equally distributed across partitions, the run-time characteristics may not lead to evenly balanced run-time of partitions. In order to take the run-time characteristics into consideration, the partitioning algorithm also considers use case in which the design is likely to be used. The knowledge of use case helps the partitioning algorithm in increasing the accuracy of estimating the load balancing at run-time.

The partitioning algorithm begins with annotating each node in the instance hierarchy with a weighted value obtained using a number of static parameters. The static parameters used are - number of processes, number of signal assignments, number of sequential elements and number of primitive gates. Without considering the use-case information, the weighted value at each node would be just the sum of all these numbers. Consideration of the use-case information causes more weighting to be given to a specific parameter to arrive at the weighted value of the node.

Let us first describe the use-cases and their impact on calculation of the weighted value associated with a node in the instance hierarchy. The following use-cases are considered in our partitioning algorithm.

RTL Design: For RTL designs, the weighted value of a node in the instance hierarchy tree is calculated by giving larger weighting to the parallel processes, for example initial/always blocks in Verilog or process statements in VHDL. This is due to the fact that these constructs are likely to consume more CPU resources during simulation and hence must be distributed equally across partitions for multi-core HDL simulation.

Gate level Design: For gate level designs, the weighted value of a node in the instance hierarchy tree is calculated by giving larger weighting to language primitives and user-defined primitives. This is so because for a gate level design, the simulation of primitive gate constructs is likely to consume more CPU resources.

ATPG Design: ATPG design refers to simulation of ATPG vectors after test pattern generation. This step verifies that the response to the ATPG test vectors is as expected. ATPG tools generate test patterns after inserting scan-chains in the design. Insertion of scan-chains causes sequential elements in the design to be replaced by equivalent scan elements and connecting the scan elements in the form of a chain. This allows specific test patterns to be scanned into the scan chains for detection of manufacturing faults. An ATPG tool generates patterns in such a manner that a large number of manufacturing faults are detected using each pattern. During simulation of ATPG designs, large amount of circuit activity is generated around the sequential elements. Hence, for partitioning, the weighted value of a node in the instance hierarchy is calculated by giving larger weighting to sequential elements.

The weightings assigned to various static parameters is a relative weighting. If there are n static parameters s_1 to s_n, with weightings of w_1 to w_n, respectively, the weighted value of a node is obtained as

$$w_{node} = (\text{count of } s_1 + \ldots + \text{count of } s_n) / (w_1 + \ldots + w_n)$$

The first step in the partitioning algorithm is to annotate each node in the instance hierarchy with a weighted value. Each node has two weighted values, self-weighted value and cumulative weighted value. The self-weighted value is the weighted value of parameters belonging to that node. The cumulative weighted value of a node is the sum of the cumulative weighted value of all child nodes of that node and its own self weighted value.

Annotation is performed in a bottom up fashion on the instance hierarchy tree. Initially, all the leaf level instances are assigned a weighted value. As explained above, the weighted value is calculated using the number of static parameters and the knowledge of the use-case. The non-leaf instance nodes of the hierarchy tree are then assigned a weighted value. The cumulative weighted value of a non-leaf instance node is determined as,

$$w = \text{sum of weights of all child instances} + \text{self-weight of the instance}$$

An example annotated instance hierarchy is shown in Fig. 2.

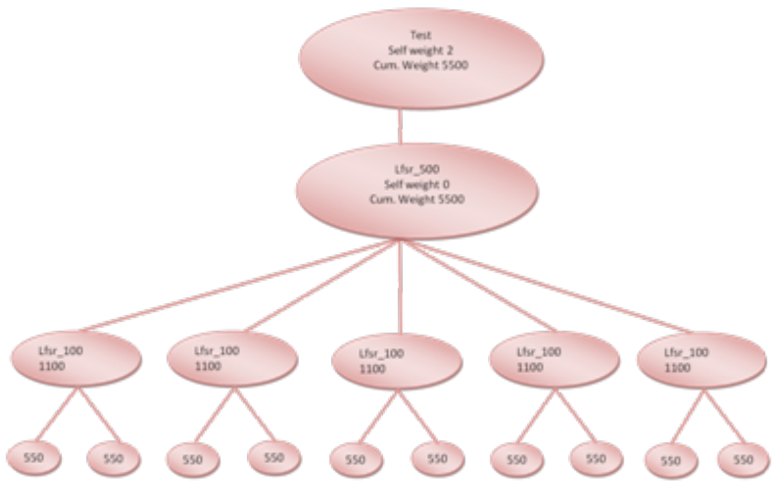

Fig. 2 : An example annotated instance hierarchy

After the annotation step is complete, the cumulative weighted value of the root or top node of the instance hierarchy tree indicates the total weighted value that must be equally distributed among partitions. This value is divided by N, where N is the number of partitions to be obtained, to get a goal weight G.

Next step of the partitioning algorithm is to perform bin-packing of the instance hierarchy into N bins of capacity G. We have developed a dynamic bin-packing algorithm, which is a variation of the standard bin-packing algorithm. In standard bin-packing algorithm, the items to be packed are of fixed size. In the static partitioning problem, the size of an item keeps varying as the packing progresses. For any node of the instance hierarchy to be packed in to a bin, if the node itself cannot be fit into any of the bins, its child nodes are attempted to be fit. If a child node is fit into a bin, the weight of the parent node is decreased and an attempt is made to fit the parent node again.

A brief explanation of the algorithm is as follows. Starting with the top node in the hierarchy, the algorithm does a breadth first traversal. At any stage if the node can be fit into a bin, the entire hierarchy below that node is considered to have been fit into the bin. The weight of the parent is reduced and the algorithm proceeds with the parent node. At any point if a node neither has any child node which can be fit, nor can itself be fit into a bin, it is forced fit into a bin even if there is overflow in that bin. This is done since there is no other way to fit that particular node into a bin. This algorithm is called dynamic bin-packing algorithm since the size of the bins to be fit keeps varying.

The algorithm takes as its inputs the instance hierarchy, the number of partitions to be generated and the use case to be considered. It generates a mapping of nodes in the instance hierarchy to N partitions. This partition mapping in turn is used for multi-core HDL simulation.

IV. Experimental Results

The partitioning algorithm was applied to a number of HDL designs and the generated partition was used with an HDL simulator which operates on instance level partitions and performs multi-core HDL simulation using it. The results are presented in Table 1. All the tests are real design blocks ranging from 5 million gates to 100 million gates.

Test	**Type of design and simulation**	**Number of Partitions**	**Performance ratio using multi-core simulation**
Test 1	Gate level	4	1.6x
Test 2	Gate level	2	1.36x
Test 3	Gate level	2	1.13x
Test 4	Gate level	4	1.39x
Test 5	Gate level	2	1.6x
Test 6	Gate level	4	2.1x
Test 7	RTL	2	0.9x
Test 8	ATPG	2	0.5x

Table 1: Experimental Results

The column "Performance ratio using multi-core simulation" is arrived at time taken by single core simulation by the time taken by multi-core simulation. It indicates the order by which multi-core simulation provided performance gains.

The experimental results in Table 1 indicate that the partitioning algorithm could achieve a good balanced partition for most tests, there by leading to performance gain when the generated partition is used of multi-core HDL simulation. However, there are a few tests for which the generated partition could not achieve performance gain, and in fact caused loss of performance with multi-core HDL simulation. This is due to the fact that the partitioning algorithm considers design parameters and use-case information to perform the partitioning. The activity distribution at run-time could cause the partition to be ineffective. The partitioning algorithm in any case provides a starting point for deployment of multi-core HDL simulation.

V. Conclusions

In this paper, we have presented a partitioning algorithm for instance level partitioning to be used with multi-core HDL simulation. The algorithm uses design parameters and knowledge of use-case to equally distribute work load such that maximum performance gains can be achieved using multi-core HDL simulation. The algorithm has shown good results on a number of designs. There are a few designs on which the generated partition caused performance degradation with multi-core HDL simulation. This indicates that more design parameters need to be used and the weightings used for use-cases might have to be tuned. In any case, the presented algorithm provides an initial partition that will allow easy deployment of a multi-core HDL simulation solution.

References

[1] A. Raynaud, "The new gate count: What is verification's real cost?", Electronic Design, October 2003.

[2] Richard M Fujimoto, "Parallel Discrete Event Simulation", Communications of The ACM, October 1990, Vol 33, No 10, pp 30 - 53

[3] Alois Ferscha and Satish K. Tripathi, "Parallel and distributed simulation of discrete event systems", Technical Report, University of Maryland at College Park, 1994

[4] Jason Liu, "Parallel Discrete-Event Simulation", Technical Report, Florida International University, 2009

[5] Gerd Meister, “A Survey on Parallel Logic Simulation”, Technical Report, University of Saarland, Germany, 1993

[6] Hansen Dai and Bill Paulsen, “Multithreading VHDL Simulation”, VHDL International Users Forum, 1994, pp 4.33 – 4.38

[7] Lijun Li and Carl Trooper, “A Multiway Design-driven Partitioning Algorithm for Distributed Verilog Simulation”, Simulation, Vol 85, Number 4, pp 257-270

[8] K-H Chang and Chris Browy, “Parallel Logic Simulation: Myth or Reality”, Computer, April 2012, pp 67-73