

Assessing the potential of multi-threaded VHDL simulation

John Sissler



IKOS SYSTEMS, INC.
1000 Wyckoff Ave
Mahwah, NJ 07430
Tel: (201) 848-8000
Fax: (201) 848-8189
Email: john@ikos.com

Abstract

As a standard modeling language, VHDL promises to reduce the cost of ASIC and standard component modeling while increasing the availability of high quality libraries. However, to fully realize its potential, VHDL must offer competitive performance. Although VHDL simulators continue to get faster, a significant performance and capacity gap still exists between VHDL and proprietary tools, particularly at the gate level. Multi-processor workstations provide a platform for accelerating VHDL simulation through the application of multi-threading. Given a pessimistic, time-synchronized implementation, the performance potential of a multi-threaded VHDL simulator is directly proportional to the degree of parallelism exhibited by a particular design executing a specific stimulus. One way to assess this potential is to instrument a VHDL simulator with the ability to generate statistics from which the parallelism of a design simulation can be determined. Several typical design simulations were executed on an instrumented version of the IKOS Voyager VS simulator. The results indicate the upper and lower bounds of the simulation performance that can be expected from a multi-threaded implementation employing a pessimistic concurrent algorithm.

1.0 Introduction

Gate-level modeling problems and poor simulation performance are the two primary impediments keeping VHDL from emerging as a standard language for both behavioral and gate-level simulation. The first problem is being tactically addressed by the current VITAL ASIC modeling effort and ultimately will be resolved via changes to the language. The performance gap that currently exists between VHDL and proprietary gate-level simulators will also close over time, as modeling standards such as VITAL are optimized on a per-implementation basis and incremental general-purpose optimizations are added to existing VHDL simulators. The increasing availability of multi-processor workstations featuring shared memory and efficient thread synchronization mechanisms offer yet another strategy for accelerating VHDL simulation. However, it's important to establish a reasonable set of expectations regarding the potential of multi-threaded simulation. Parallel logic simulation is notoriously difficult to implement efficiently due to time synchronization requirements. Optimistic algorithms which relax synchronization requirements consume significant amounts of CPU and memory; the overhead may well exceed the compute requirements of performing the actual simulation.

This paper assesses the acceleration potential of multi-threaded VHDL simulation. First, a brief overview of parallel discrete event simulation is provided. Next, a description of

the data produced by the instrumented kernel is given, along with an explanation of how the data is used to determine the degree of parallelism. Finally, the results of executing several design simulations are presented along with a set of conclusions drawn from the data.

2.0 Multi-threaded VHDL simulation

Concurrent algorithms for discrete event-driven simulation have been a popular research topic for many years [1-5]. Yet no successful commercial implementation of a concurrent digital logic simulator exists, other than solutions employing special-purpose hardware. While there has been great success in optimizing sequential logic simulation, most concurrent strategies have exhibited significant compilation (partitioning) or simulation overhead. An optimized sequential gate-level simulator may execute less than 100 CPU instructions to simulate a full single-gate cycle. In contrast, a concurrent simulator using an optimistic or virtual time algorithm may impose one to two orders of magnitude of processing overhead. There have been other factors which have blocked the commercial availability of concurrent simulators. Until recently, the market has not provided suitable multi-processor workstations nor has there been a standard software interface for implementing multi-threaded applications. Both of these constraints no longer hold; Sun Microsystems has shipped thousands of multi-processor workstations and the IEEE POSIX P1003.4a Threads Extension for Portable Operating Systems [6] presents a standard interface for constructing multi-threaded applications.

Given a highly-optimized sequential VHDL simulator kernel, the key to achieving maximum performance on a multi-processor platform is to maximize concurrency while minimizing overhead. Optimistic algorithms which relax the synchronization requirements of each thread offer concurrency but may incur unacceptable overhead. Pessimistic algorithms which require that each thread be time-synchronized reduce overhead but may not obtain satisfactory concurrency. A pragmatic solution to this dilemma may be to find an inexpensive pessimistic algorithm that can take advantage of the natural concurrency of the design being simulated. The performance potential of such a solution would depend on how much parallelism is inherent in each design and how little overhead is required to support concurrent execution.

Section 12.6.3 of the VHDL LRM defines the sequential algorithm of the simulation cycle as follows:

1. If no driver is active, then simulation time advances to the next time at which a driver becomes active or a process resumes. Simulation is complete when time advances to TIME'HIGH.
2. Each active explicit signal in the model is updated. (Events may occur on signals as a result.)
3. Each implicit signal in the model is updated. (Events may occur on signals as a result.)
4. Each process that has just resumed is executed until it suspends.

The IKOS Voyager VS VHDL kernel's implementation of the above algorithm is as follows. (Note that for simplicity, the algorithm described does not account for resolved or implicit signal handling.)

1. Retrieve the list of active transactions from the global queue. If the queue is empty, simulation is inert.

2. For each active transaction, invoke the driver which scheduled the transaction to update its net. The driver propagates the transaction to all signals on the driver's net and advances its waveform, scheduling the next transaction on the global queue. Signal events cause each signal's sensitivity list of processes to be inserted onto the process list.
3. For each process on the process list, execute the process until it suspends.

A pessimistic concurrent algorithm of the above sequential algorithm attempts to execute the body of each iterative loop in parallel. The goal is to execute each iteration of both signal update and process execution concurrently with all other iterations. Assuming that N copies (threads) of the kernel process are executing, a straight-forward algorithm for each thread is as follows.

1. Synchronize with parent thread, which retrieves the list of active transactions from the global queue. If queue is empty, simulation is inert.
2. While an active transaction can be retrieved from the list, invoke the driver which scheduled the transaction to process it. Transaction list access is protected via an atomic lock. The driver propagates the transaction to all signals on the driver's net and advances its waveform, scheduling the next transaction on the global queue. Queue access is protected via an atomic lock. Signal events cause each signal's sensitivity list of processes to be inserted onto the process list. Process list access is protected via an atomic lock.
3. Synchronize with parent thread, which retrieves the list of resumed processes.
4. While a process can be retrieved from the process list, execute the process until it suspends. Process list access is protected via an atomic lock.

The performance of the above algorithm is bounded by the number of processors, the mean number of active drivers and processes per simulation cycle, and the compute overhead required to implement the concurrent algorithm. Note that the algorithm is burdened with synchronization points and memory barriers. An actual optimized implementation of the above would seek to minimize both of these potential bottlenecks. There are several other primary factors which would affect performance, such as process complexity and the frequency of driver scheduling. Much of this data is readily obtained by examining the call-graph profile of typical simulator executions. However, the remainder of this paper will focus on determining the range of natural concurrency that is exhibited by single-ASIC RTL simulations.

3.0 Instrumenting the simulator kernel

To determine the natural concurrency inherent in a variety of VHDL simulations, the IKOS Voyager VS simulator kernel was modified to track two random variables. One random variable represented the number of active drivers per simulation cycle, the second modeled the number of VHDL processes executed per simulation cycle. Each random variable maintained its mean, variance, and standard deviation. Upon the completion of each simulation run, these statistics were recorded to standard output.

4.0 Simulation results

Eight IKOS customer designs were simulated using the instrumented kernel and the mean, variance, and standard deviation of the two random variables were recorded. All

of the designs modeled single ASIC components and most were written using a synthesizable RTL style. The size and complexity of both the ASIC model and the surrounding stimulus environment varied greatly. To communicate design size, raw lines-of-code statistics are provided.

The table below presents the data generated from the instrumented simulation runs.

TABLE 1. Concurrency statistics extracted from ASIC simulations.

Design	Raw Lines-of-code	Mean active drivers per cycle	Mean active processes per cycle
ASIC #1	51,143	29.58	24.13
ASIC #2	9,143	3.26	2.22
ASIC #3	17,587	5.82	2.87
ASIC #4	52,279	7.38	23.40
ASIC #5	13,861	54.71	52.74
ASIC #6	16,131	7.66	15.03
ASIC #7	14,831	8.86	12.51
ASIC #8	7,321	18.25	35.04

5.0 Conclusions

The data in table 1 indicates that the design simulations executed exhibited a fairly wide range of natural concurrency. Additional research is necessary to determine the cause of this variation, which could be some combination of design character and the relative maturity of the design and stimulus. However, since we are interested in determining the potential for multi-threaded acceleration during all stages of the design cycle, all of the above data is germane.

A fundamental and encouraging observation that can be drawn from the data is that the range of natural concurrency falls neatly within the 2 to 64 processor range of today's multi-processor workstations. Multi-processor workstations featuring shared memory and efficient memory barriers are the ideal platform for offering multi-threaded CAE tools. Multi-processor workstations offer the customer the ability to run existing applications transparently and offer the CAE vendor the ability to multi-thread software in a standard, familiar, and cost-effective environment. Standards such as the IEEE POSIX P1003.4a also promise to enable multi-threaded applications to be portable across a wide variety of computer platforms.

It is important to note that all of the above simulations modeled single-ASIC designs. Single processor workstations are usually a satisfactory platform for simulating ASIC models. However, single processor workstations may not be adequate to simulate large systems consisting of multiple ASIC models, large memories, and large standard component models. For at least some systems and perhaps the majority, the natural concurrency of a system simulation should be approximately the sum of the concurrency of each system component. Thus, a system simulation consisting of 2 ASIC models with a mean concurrency factor of 10 may exhibit a concurrency factor of approximately 20. Obviously more data is required to form a more accurate conclusion regarding the concurrency of system simulations.

Potential buyers of multi-threaded simulation products should assess real tool performance, not the relative performance of a multi-threaded simulator vs. its single processor implementation. VHDL simulators exhibit a wide range of performance, as shown by the table below, which presents the results of an independent benchmark published in EE Times [7]. A multi-threaded simulator may achieve a significant performance increase on a multi-processor workstation and still not compete with another simulator executing on a single-processor.

TABLE 2. EE Times VHDL Benchmark Results

VHDL Simulator	Simulation time (s)	Memory Usage (mb)
IKOS Systems Voyager VS 1.3	137	3.2
Cadence Leapfrog 1.0	212	3.2
Model Technology V-System 3.2	277	3.3
Vantage Analysis Spreadsheet 4.1	278	4.2
Viewlogic Viewsim VHDL 5.14	576	3.3
Mentor Graphics System-1076 8.2	989	20.5
Synopsys VSS 3.0a	1136	4.9

In conclusion, the data suggests that if overhead can be minimized, the degree of natural concurrency inherent in typical single-ASIC and multiple-ASIC system simulations is significant enough to exploit multi-processor workstations to accelerate VHDL simulation during several stages of system development. More data is needed to assess the potential for accelerating gate-level simulation using similar algorithms. It is likely that gate-level simulation will offer more potential concurrency but since gate-level VHDL processes may be less complex than behavioral processes the overhead required to support concurrency may become a more significant bottleneck.

References

- [1] R.E. Bryant. "Simulation on distributed systems", Proceedings of the First International Conference on Distributed Systems, 1979.
- [2] K.M. Chandy and J. Misra. "Distributed simulation: a case study in design and verification of distributed programs", IEEE Transactions on Software Engineering, SE5(5):440-452, 1979.
- [3] D. Jefferson and H. Sowizral. "Fast concurrent simulation using the time warp mechanism", Distributed Simulation 1985, the 1985 Society for Computer Simulation Multi-conference, 1985.
- [4] D. Jefferson. "Virtual time", ACM Transactions on Programming Languages and Systems, July 1985.

- [5] J. Misra, "Distributed discrete-event simulation", *Computing Surveys*, March 1986.
- [6] IEEE P1003.4a/D6 Draft. "Threads Extension for Portable Operating Systems", February 26, 1992.
- [7] David Wharton, "VHDL/Verilog take simulation run", *Electronic Engineering TIMES*, June 14, 1993.