Use of Embedded Scheduling to Compile VHDL for Effective Parallel Simulation

John Willis
System Technology & Architecture Div.
IBM Corporation
Rochester, MN USA
jwillis@acm.org

Zhiyuan Li
Dept. of CS, University of Minnesota
200 Union Street SE, #4-192
Minneapolis, MN 55455 USA
li@cs.umn.edu

Tsang-Puu Lin
Dept. of CS, University of Minnesota
200 Union Street, #4-192
Minneapolis, MN 55455 USA

This work was supported in part by Carnegie Mellon University, the IBM Corporation (including Grant Number 340-659), by the Army Research Office (Contract DAAL03-89-C-0038 with the University of Minnesota Army High Performance Computing Research Center) and by the Graduate College of the University of Minnesota. The work is continuing at FTL Systems and the University of Minnesota.

Abstract

This paper describes VHDL compilation techniques, embodied in the Auriga compiler [3,14], which facilitate parallel or distributed simulation by embedding evaluation scheduling in the emitted code. Unlike earlier but related cycle-driven techniques which map VHDL into simpler temporal semantics, the techniques described here preserve VHDL's full temporal semantics. Experimental results indicate effective simulation acceleration using as many as 16 processors. Ongoing work involves evaluation with much larger models and machine configurations.

1: Introduction

Compiler researchers view the automated translation (compilation) of mainstream computer languages as an essential long term objective for effective execution on parallel, message-based computers. VHDL's intrinsic parallelism and static process graph make VHDL an excellent intermediate challenge for parallel compiler researchers. This work describes a successful step toward these challenges.

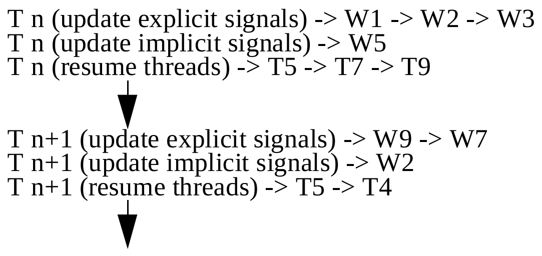
Many contemporary research efforts addressing parallel VHDL simulation focus on *run-time* algorithms for logically correct and ideally decoupled parallel execution [1, 2, 16]. In contrast, the emphasis of this work is on *compile-time* efforts which target a simplified, asynchronous, conservative run-time environment. Relative to earlier papers by some of the same authors, this paper provides greater detail on the preparations for embedded scheduling, embedded scheduling itself and two sets of performance measurements from a prototype of the final Auriga compiler [3, 14].

This paper begins by describing the intrinsic problem being addressed in the context of an earlier approach to optimized simulation: cycle-driven simulation. We discuss the translation of VHDL source code into an intermediate form consisting of communicating sequential processes followed by clustering of processes to reduce the number of independently scheduled threads. Rather than commutating the resulting set of threads with a generic thread scheduler, we use an adaptation of Bryant/Chandy/Misra's conservative discrete event simulation algorithm where the scheduler is an integral “glue” for the compiled threads. Mapping and scheduling algorithms take into account communication and computational resource utilization patterns to arrive at an assignment of threads to processors (map) and a static evaluation order (schedule). Following a presentation of experimental results we discuss efforts to adopt the technology for production use and future extensions.

2: Background

Analysis and elaboration of VHDL models conceptually yields multiple instruction "threads". Each thread corresponds to the behavior of an elaborated process, concurrent statement or perhaps an implicit process (e.g. derived from a signal of resolved type). Execution of each thread is enabled at the beginning of execution and resumes at the expiration of a time interval or when one or more signals to which the thread is sensitive become active. The simulation kernel commutates each processor among a set of active threads.

Figure 1: Data Structure used by Generic Scheduler


T n (update explicit signals) -> W1 -> W2 -> W3
T n (update implicit signals) -> W5
T n (resume threads) -> T5 -> T7 -> T9
T n+1 (update explicit signals) -> W9 -> W7
T n+1 (update implicit signals) -> W2
T n+1 (resume threads) -> T5 -> T4

Most VHDL simulator implementations use a generic simulation kernel driven by some form of a "time-wheel" [5, 8], such as the one shown in Figure 1. Corresponding to each time step with a pending waveform element or scheduled thread resumption (T), there is a list of pending waveform elements which need to update signals and a list of threads which need to be scheduled for execution. The generic simulation scheduler provides model-independent functionality needed to schedule waveform elements, schedule thread resumptions, update signals and execute threads. The data structure "customizes" the generic simulation kernel for the execution (simulation) of a specific model.

While generic simulation kernels are common among contemporary uniprocessor VHDL simulators, this generic approach implies several significant run-time computational costs, beyond actual thread execution, including:

- waveform element scheduling,
- thread scheduling,
- signal updating, and
- hardware processor scheduling.

Cycle-driven or levelized-compiled-code (LCC) [6, 7] abstracts timing information from the run-time model and statically orders thread evaluation. These simplifications enable gate-level simulation mapped directly into several machine instructions per gate-level evaluation, often resulting in an order of magnitude acceleration in simulation performance. Unfortunately this simulation performance comes at a cost:

- applicability is limited to synchronous designs,
- timing information is lost and related errors masked,
- can slow simulations with low probability of thread activity at a given instant in simulation time, and
- correspondence between VHDL source code and executable is reduced, complicating source code debug.

Use of a generic simulation kernel on each processor within a parallel simulation has another, somewhat hidden performance deficit. Waveform elements and thread evaluations are typically queued in an effectively random order, resulting in the commutation of processors among thread executions in effectively random order. Unfortunately, in the context of a parallel simulation, the results of executing some threads are critical to enable computations on other processors (perhaps executing out-of-phase or at a distinct time step) while other evaluations are non-critical. Generic kernels cannot exploit knowledge of the critical path without incurring yet more run-time overhead.

Several research groups have combined cycle-driven and event-driven simulation [3, 9] in an effort to locally harness the advantages of both. This paper addresses some of the techniques needed to extend hybrid LCC/event simulation for optimized execution on parallel processors.

3: VHDL to CSP Translation

Early in the compilation process, VHDL source code must be analyzed into an intermediate form, then elaborated into a form resembling a communicating sequential processes (CSP) paradigm [10]. The CSP model facilitates execution using either conservative [11] (Bryant/Chandy/Misra) or optimistic (Jefferson) [12] algorithms for parallel discrete event simulation. This work uses a conservative, asynchronous algorithm.

Elaboration flattens the VHDL design hierarchy by "rewriting" component instances (with bound entity and architecture) and concurrent statements in terms of their process equivalents. This rewriting is largely defined by the IEEE VHDL 1076-93 language definition [13].

Processes elaborate into instruction threads and optional thread-local data. The instruction threads are sequential (with embedded control flow constructs such as if, case, and loops). Most subprogram calls are inlined (to facilitate optimization and improve performance). Only select recursive calls and calls to subprograms with foreign bodies actually generate synchronous subprogram call and return. Instruction threads include wait statements, which suspend execution of the instruction thread pending activity on message channels or expiration of a time interval. Thread-local data can be divided into transient data and data which persists across one or more wait statements. Transient data can be allocated on a stack shared by all threads whereas persistent data can be transformed into a message communication channel driven and received (solely) by a single thread. Wait statements embedded in subprograms require special handling [14].

Signals elaborate into one or more communication channels along which messages may flow. Messages represent signal activity (signal assignments) and thus events (signal assignments resulting in a change of signal value). Signals of composite type may be either driven on an element-by-element basis by distinct processes or resolved. If each element is driven by a distinct driver, the signal can be decomposed into multiple message channels. Resolved signals can be translated into implicit processes interposed between drivers and receivers. The resulting message channels are unidirectional with a unique driver and one or more receivers (message channels with no receivers can be optimized out of the run-time system in the absence of source code debugging or logging considerations).

4: Thread Clustering

When executing a sequence of threads during simulation, the cost of commutating a processor among multiple threads capable of resuming is not zero. One means of eliminating this computational cost is to collapse two or more threads into a single thread. In literature describing parallel, discrete event simulation, thread collapsing is often described as collecting multiple “physical processes” into a single “logical process”.

Among those threads which are suitable for thread clustering, analysis identifies sets of activation regions which resume execution based on the same criteria and dependencies between activation regions such that execution of one activation region within the thread (conditionally) enables another thread.

Activation regions with common resumption conditions can be collapsed into a single thread with a common prologue followed by either a concatenation of the threads (uniprocessor node) or *do-across* parallel execution for shared-memory multiprocessors. In this way, the cost of thread switching is paid once for the set of activation regions rather than once for each activation region.

In a like manner, activation regions with dependencies can be collapsed into a single thread such that the machine instructions from one activation region follow those from a dominating activation region (region on which another activation region depends). Since there can be several dominating activation regions, instructions from one thread may be replicated. A simple example is the collapsing of an OR gate into multiple threads.

In the following discussion, the properties of a set of collapsed threads closely resemble those of un-collapsed threads (except for shared state); thus the discussion will not distinguish between collapsed and un-collapsed threads.

5: Mapping to Processors

Following optimizing transformations on the intermediate CSP representation [3], the mapping phase of compilation determines which processor or shared memory multiprocessor (node) will be used to execute each activation region.

The mapping algorithm results from experimental trials using a variety of alternatives. We refer to it as the Bottleneck Reduction Heuristic (BRH). BRH uses dataflow analysis already generated by earlier optimizing transformations [3] to estimate both the average computational workload required to execute all activation regions and inter-processor queuing implied by a tentative mapping of activation regions to nodes. A second metric, the flow rate, helps to determine a priori the optimal number of processors for execution of a particular model. Iteration serves to establish the final mapping (and number of processors actually used). For further details of the BRH algorithm, see [4].

6: Scheduling Each Processor

Once the set of activation regions mapped to each processor or shared memory multiprocessor is determined (see Section 5 above), the scheduling phase determines the static order in which code for activation regions will be laid out in memory.

Generally, activation regions are scheduled so as to minimize the average number of activation regions between generation and use of a message (manifestation of signal activity). The criticality of a message for enabling evaluation on other nodes also guides the static scheduling algorithm.

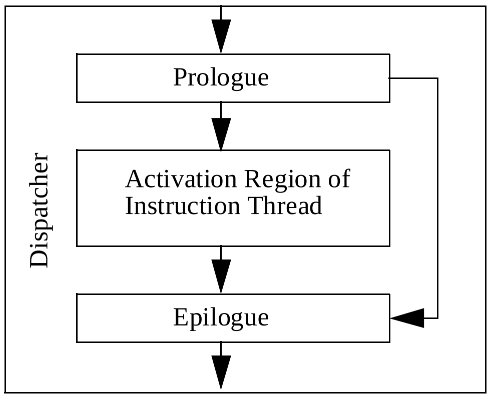
7: Embedded VHDL Scheduling

Embedded scheduling prefaces each activation region (described in Section 5.0 above) by a prologue and follows each activation region by an epilogue. A prologue, activation region and epilogue are referred to as an embedded *dispatcher,* as shown in Figure 2. These dispatchers implement an asynchronous, conservative evaluation paradigm.

When execution reaches the prologue, the associated activation region may be skipped entirely, execute once, or execute many times before proceeding to the epilogue. The ability to introduce this control flow complexity prior to the activation region is a key difference between traditional levelized compiled code and this work. In localized cases where levelized code compilation would be feasible, no machine instructions are emitted at all for the prologue.

As the localized VHDL being compiled and the need for observability require more complex timing, the prologue can grow in complexity (and increased run-time). Sensitivity lists with aggregates that don't match one to one with the signal / message pathways complicate prologues. The ability to probe the internal state of a process at a fixed instant in time can complicate the prologue (and/or epilogue).

Figure 2: Composition of an embedded dispatcher

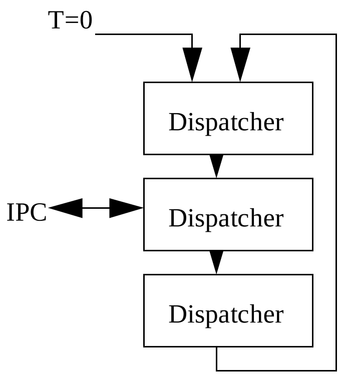


When execution reaches the epilogue, waveform elements assigned by the thread may be scheduled, "null messages" may be generated denoting the assignment of a value to a message channel for an extended period of time or messages may be generated indicating that activity took place even though the value remains unchanged.

The epilogue can choose for execution to fall-through to the next dispatcher or can make use of context to choose a more optimal transfer of control. For example, as a result of projected waveform elements assigned during evaluation of this dispatcher, the epilogue can often determine that other dispatchers are more or less likely to be able to step forward in time (via an evaluation), branching accordingly.

The set of dispatchers mapped to a given processor are generally mapped contiguously in instruction memory with an unconditional branch after the last dispatcher's prologue returning control flow to the first dispatcher (see Figure 3). Ancillary dispatchers may be inserted in the sequence of dispatchers to support inter-processor communication, logging or shared variables. Details will be presented by the Auriga team in an upcoming publication on distributed simulation of VHDL shared variables.

Figure 3: Arrangement of dispatchers on processor



8: Experimental Results

To evaluate the performance of these techniques, we have undertaken numerous experiments on a Thinking Machines CM-5, other message-based parallel machines, networks populated by Encore and Sequent shared memory multiprocessors and a simulated testbed running with multiple processes on a uniprocessor SPARC

Although work on actual parallel machines has provided very satisfying feedback as to the feasibility and performance available, none of the available machines and programming environments provided the instrumentation accuracy and parameter variability of our testbed. The testbed, written by Tsang-Puu Lin, uses the same uniprocessor as each CM-5 processing element (SPARC-based); communication is emulated in the test-bed by parameterized code. Parameters within the testbed include:

- Number of processors (2 to 512),
- Communication latency (50, 100, & 200 clock cycles), and,
- Message sizes were fixed at 20 bytes (CMMD brief message) with an inter-message latency of 5 cycles and 10 cycles to enqueue or dequeue.

The following figures report speedup results for the two relatively small models described in Table 1. The adder model was written at the University of Pittsburgh by Steven Levitan (with an added test-bench) and the queuelevel model was written at the University of Virginia by R. Rao. Models approaching 100K dispatchers have been effectively accelerated on the prototype using 64 processors. Further, audited performance data should be available during the conference presentation.

Table 1: Characterization of Models Used

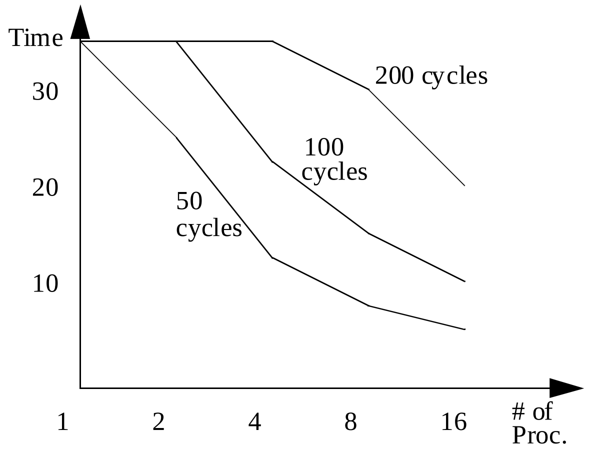
Model Identifier	**Number of Dispatchers**	**Cycles**
Adder	**44**	**No**
Multiprocessor	**39**	**Yes**

Figures 4 and 5 report the execution time as a function of the logarithmic number of processors and communication time. The CM-5 communication latency is between 50 and 100 cycles. Distributed systems, even with optimized device drivers and switches often have communication latencies of at least 400 cycles. TCP/IP over Ethernet has communication latencies *much* higher, often tens of thousands of cycles. Such high latency platforms do *not* appear to be viable for generalized, effective simulation acceleration.

Table 2: Speedups for CM-5 like testbed parameters

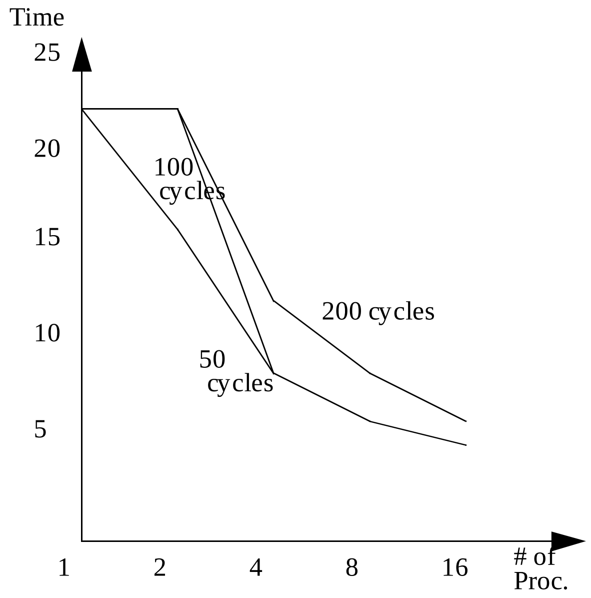
Model Descriptor	**Number of Processors**	**Speedup Observed**
Adder	4	3 times
Multiprocessor	8	5 times

Figure 4: Speedup for adder example



For CM-5 like-machines, we measured the approximate speedups reported in Table 2. The models were compiled through to the CSP intermediate form using Auriga, then translated for execution on the test-bed by prototype code. At the time this data was gathered, optimization strategies such as thread clustering (described above), input desensitization [3], temporal analysis [3], waveform analysis [3] and subtype enumeration [3] were not incorporated in the measurement. Use of these optimizations and much larger models effectively uses more processors and increases the speedup observed.

Figure 5: Speedup for multiprocessor example



9: Conclusions

From these and other runs on actual parallel and distributed hardware, have been lead to several conclusions:

- Speedups of 5 to 7 times are feasible for small models and relatively modest size parallel processors.
- Low communications latency resulting from tuned operating system kernels, direct device driver paths and accelerated interconnects are mandatory for accelerated simulation.
- Compile-time analysis, optimization and native code generation can dramatically improve uniprocessor and parallel processor performance relative to perform from generic simulation kernels running on uniprocessor.

10: Future Work

When compiling a language such as VHDL, there is an *enormous* gap between getting excellent uniprocessor performance and/or parallel processor acceleration using a prototype compiler/run-time system and having a production-ready tool. This paper describes the former and not the latter effort. Research prototypes need not address:

- Full language coverage (digital, analog)

- Analysis, elaboration and run-time error handling
- Reasonable optimization of diverse modeling styles
- Robust handling of hardware failures
- Portability among diverse hardware platforms
- Source level debugging and waveform display
- Support for related standards (Vital, OMF...)

Substantial effort, measured in tens of person-years, separates a successful prototype from a successful parallel product. This difference helps to explain the large number of parallel VHDL research projects / publications and the absence of any optimizing VHDL compilers targeting parallel or distributed, general-purpose processors.

Efforts to transform Auriga's technology into a production-quality compiler began at IBM (MinSim [15]) and are continuing at FTL Systems. This effort is progressing toward publishable measurements of multi-million process designs simulated with hundreds of processors.

Given that most of Auriga's complexity is related to optimization and code generation from a CSP-based intermediate (in a global, persistent memory database) rather than VHDL, there is potentially a great deal of leverage analyzing other hardware description languages (e.g. analog VHDL and Verilog) and even programming languages (e.g. C++) into Auriga's CSP intermediate. In this way, we hope to address the broader challenge of automatically translating mainstream programming languages for effective execution on parallel, message-based computers.

11: Acknowledgments

The authors appreciate the many useful comments which shaped this work. Daniel Siewiorek (CMU) guided development of the Auriga compiler at Carnegie Mellon University. Dalibor Vrsalovic advocated a "work-pile" scheduling algorithm, predecessor to the one described here. Don Austin (AHPCRC) supported experimental aspects of this work on the CM-5 massively parallel processor. Thanks to the Steve Levitan at the University of Pittsburgh and R. Rao of Virginia Tech for the VHDL models referenced in this paper. Thanks to Jeff Snyder (IBM) and two anonymous reviewers for their many suggestions resulting in improvements to this paper.

12: References:

[1] T. McBrayer, V. Krishnaswamy, S. Mohanty, L. Moore, X. Liu, J. Carter, D. Charley, P.A. Wilsey, Da.A. Hensgen, H.W. Carter, P. Chawla, J. Colier, S. Bilik, *VAST: Time Warp Simulation of VHDL on SMP Workstations*, VHDL International User's Forum, Conference Management Services, November, 1994, pages 4.17-4.32.

[2] Hansen Dai and Bill Paulsen, *Multithreading VHDL Simulation*, VHDL International User's Forum, Conference Management Services, November, 1994, pages 4.33-4.38.

[3] John C. Willis and Daniel P. Siewiorek, *Optimizing VHDL Compilation for Parallel Simulation*, IEEE Design & Test of Computers, September, 1992.

[4] T.P. Lin, Z. Li and J. Willis, *Mapping Discrete Simulation Tasks on Message Passing Parallel Processors*, University of Minnesota, AHPCRC Preprint 94-022.

[5] Manuel A. d'Abreu, *Gate-Level Simulation*, IEEE Design & Test, December, 1985.

[6] Gregory F. Pfister, *The Yorktown Simulation Engine*, In 19th Design Automation Conference, pages 51-54, 1982.

[7] Lang-Terng Wang, Nathan E. Hoover, and John J. Zasio. *SSIM: A Software Levelized Compiled-Code Simulator*. In 24th ACM/IEEE Design Automation Conference, 1987.

[8] M. Abramovici, Y.H. Levendel and P.R. Menon, *A Logic Simulation Machine*, In 19th Design Automation Conference, pages 65-73, 1982.

[9] Zhicheng Wang and Peter M. Mauer. *LECSIM: A Levelized Event Driven Compiled Logic Simulator*. In 27th ACM/IEEE Design Automation Conference, pages 491-496, June 1990.

[10] C.A.R. Hoare, *Communicating Sequential Processes*, Communications of the ACM, Volume 21, Number 8, pages 666-677, August 1978.

[11] K.M. Chandy and J. Misra, *Asynchronous Distributed Simulation Via a Sequence of Parallel Computations*, Communications of the ACM, April 1981.

[12] David R. Jefferson, *Virtual Time*, ACM Transactions on Programming Languages and Systems, 7(3): pages 404-425, July 1985.

[13] *IEEE Standard VHDL Language Reference Manual*, ANSI/IEEE Std. 1076-1993.

[14] John C. Willis, *Optimizing VHDL Compilation for Parallel Simulation*, PhD Dissertation, Computer and Electrical Engineering Department, Carnegie-Mellon University, October 1991.

[15] John Willis, Rob Newshutz, Lance Thompson, Jeff Graves, Tom Dillinger, Jeff Snyder, Nimish Radia, Joe Skovira, David Blaauw, Sidhartha Mohanty, Zhiyuan Li, Sandra Samelson and Matt Lin, *MinSim: Optimized VHDL Simulation Using Networked & Parallel Computers*, VHDL International User's Forum, Conference Management Services, October, 1993.

[16] Larry Soule and Anoop Gupta, *An Evaluation of the Chandy/Misra/Bryant Algorithm for Digital Logic Simulation*, 6th Annual Workshop on Parallel and Distributed Simulation, January, 1992.