MinSim: Optimized, Compiled VHDL Simulation Using Networked & Parallel Computers

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

Application Business Systems (IBM), Enterprise Systems (IBM) & Army High Performance Computing Research Center (Univ. of Minn.)

1.0 Abstract

This paper describes technology for optimized, native compiled code simulation of VHDL models using networked and parallel computers (MinSim). We describe the structure of the compiler and runtime system as well as several new optimization techniques.

2.0 Introduction

VHDL simulators have been around for nearly ten years. Early VHDL simulators used marginally suitable intermediate forms (programming languages and existing gate-level databases), incorporated minimal optimization and only utilized the power of a uniprocessor. Today, VHDL simulators are available using native compilers with optimization techniques pioneered in the programming language community (e.g. MTI's V-System & Cadence's LeapFrog). Dedicated simulation accelerators use parallel processing to accelerate substantial subsets of VHDL (e.g. Ikos's NSIM & Zycad's VIP). VHDL simulation has come a long way.

This paper provides a glimpse of the future in VHDL simulation technology: *MinSim*. MinSim is a parallel compiler and simulator resulting from more than six years of research at Application Business Systems (IBM), Carnegie Mellon University and the University of Minnesota.

Through this research, we are demonstrating both new optimization techniques uniquely suited to compiled HDL simulation and the power of parallel processing using general-purpose processors. Use of parallel processing to compile and simulate provides capacity and performance needed to effectively execute code on large system models. Such simulations are usually too bulky and too slow for all but the largest and most expensive uniprocessors.

MinSim uses general-purpose computer nodes connected by message-passing network to support parallel compilation and parallel simulation of a single VHDL model. Each node may be a uniprocessor or a shared memory multi-

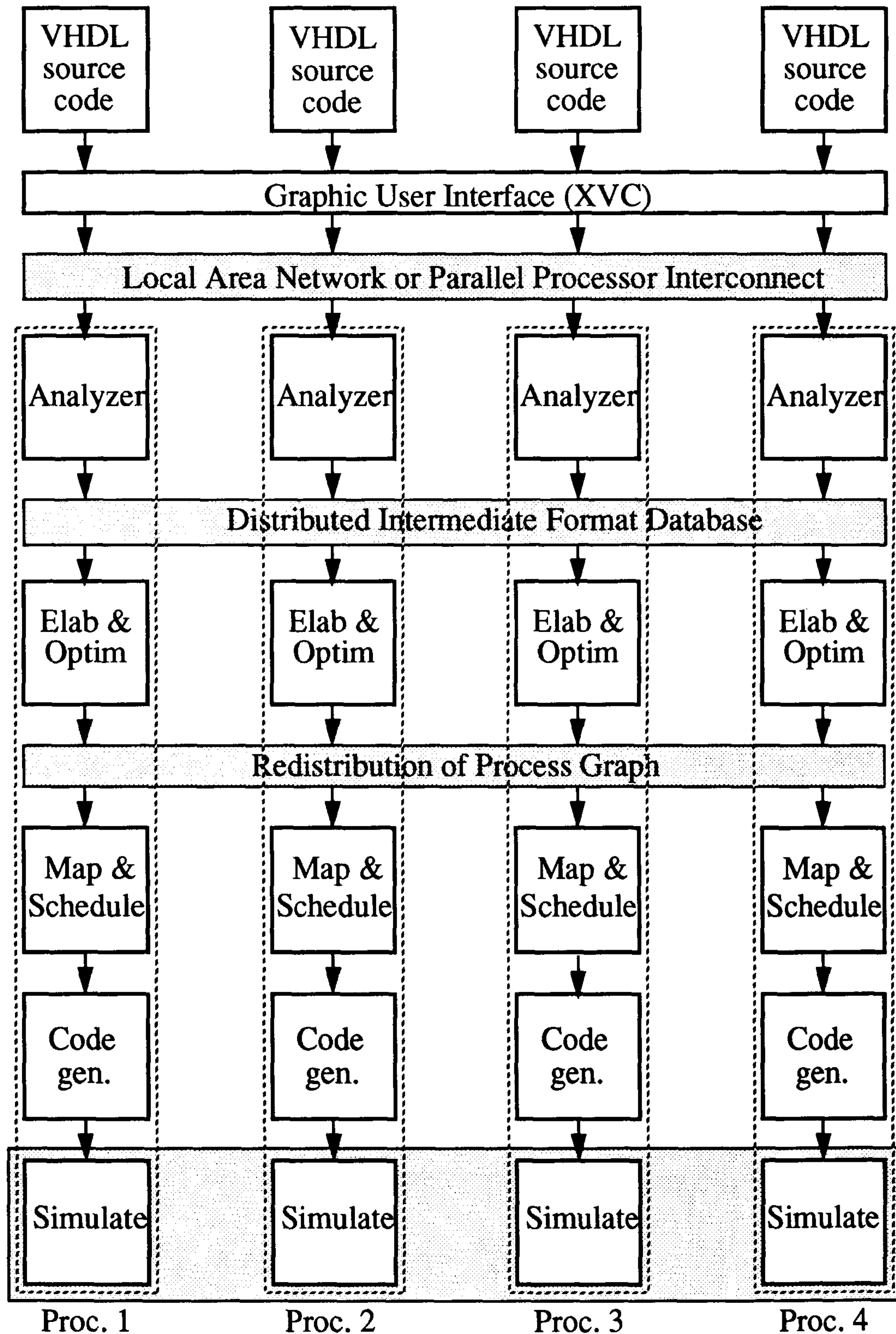

Figure 1. Parallel native code generator & parallel simulator

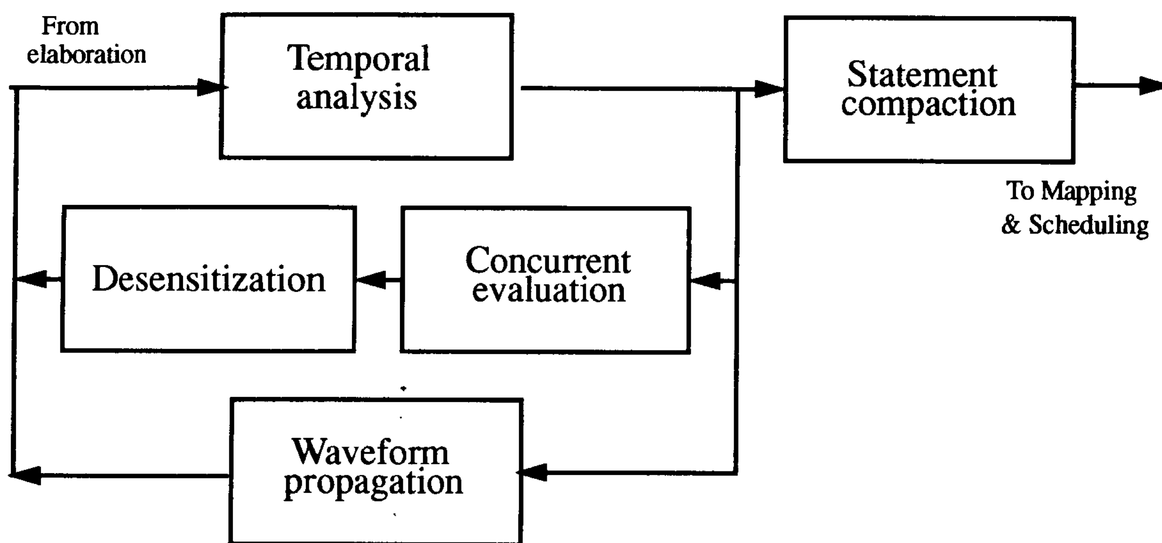

Figure 2: Structure of optimization phase

processor. Current evaluation uses uniprocessor IBM RISC System 6000 workstations, Intel 80486 personal computers and Sparc-based processors within Thinking Machine's CM-5 massively parallel computer. Networks used in the evaluation phase include token ring, Ethernet, IBM's Allnode switch and Thinking Machine's CM-5 Fat Tree.

Figure 1 illustrates MinSim's compiler and simulation structure. Many of the components shown in Figure 1 differ substantially from conventional compilers in order to facilitate optimization and parallel compilation.

3.0 Source Code Analyzer

MinSim's source code analyzer translates textual VHDL into a validated, memory-based intermediate representation. Many conventional VHDL analyzers follow the precedent of Intermetric's early VHDL analyzer by translating each analyzer design unit into a unique file within a database. In contrast, retaining the entire design database within MinSim's address space(s) accelerates analysis by reducing the number of independent I/O requests and allows ready access to all previously analyzed design units. Modification and recompilation cycles can be kept tight by dumping the compiler and address space(s) initialized with design units into an executable file, much as TeX and Emacs do.

4.0 Elaboration

Elaboration expands the hierarchy consisting of generate statements, component instance statements, most non-recursive subprogram calls and many loops with globally static bounds. At the conclusion of elaboration a directed graph represents the network of elaborated processes connected by signals. Each elaborated process consists of input ports, output ports, persistent state, transient variables and a sequence of triples. Each triple denotes a relatively simple operation expressed in canonical form. The simplicity of this intermediate is essential to subsequent, aggressive optimization steps.

5.0 Optimization

MinSim's optimizer introduces numerous innovations in HDL compiler technology. Figure 2 illustrates the interaction of a few of these optimizations. The next five subsections describe each optimization. For further details on the optimization phases, see [1].

5.1 Temporal Analysis

During simulation, the sequence of assignments to a signal often assume patterns in time. Clocks are the canonical example of such regular patterns, alternating between '0' and '1' with a predetermined period. If we also consider assignments independent of the value being assigned, many other signals in a contemporary, synchronous digital (sub)-system exhibit patterns during compilation.

Program flow analysis techniques [2] abstract the sequence of statements comprising each process into simultaneous equations defined using a process's variables, constants and symbols representing input data. These equations become the basis for attributed, context-sensitive grammars representing feasible instruction execution sequences and the values bound to signals or variables. These grammars consist of productions (resembling a BNF language definition) through which a start symbol may be recursively rewritten to describe a language of possible variable or signal values.

Each assignment in the production has a limited domain of values associated with a signal or variable at a given point. The domain of values may be a single value (literal), a set of values or a range (bounded by the assignment's subtype). The domain established by a particular production helps to statically identify possible subtype violations and assertions which are known to trigger or will never trigger.

5.2 Waveform Propagation

Based on the characterization of each process developed by temporal analysis, waveform propagation exports constraints across process boundaries. Iterative cycles of temporal analysis and waveform propagation provide a gradual extension from local optimization to global optimization.

There are three steps in each waveform propagation phase:

- Forward propagation
- Grammar alignment
- Backward propagation

Forward propagation defines a grammar at each input to a process based on the process grammar of the associated signal's driver.

Grammar alignment establishes a timing relationship between the productions of each input port grammar, resulting in the set of piecewise constant inputs applied to the process inputs. This alignment process is the most complex of the three, often limiting the degree of optimization available.

Backward propagation identifies those productions in each output port grammar which do not contribute to further simulation at any input. The process driving a given input may elide any computation which only serves to generate ignored waveform elements. In the extreme case, all elements of an output's grammar disappear, the signal disappears and thus there is no computation or storage associated with the signal during simulation.

Backward propagation annotates the grammar associated with the signal driving the data input of the flip-flop. This annotation indicates that any production which does not include the time interval immediately preceding the rising edge of the clock need not be characterized during simulation. Once the clock period is known during compilation, the event bearing messages incoming to the flip-

flop's data input need not actually include an interval.

In extreme cases, cycles of temporal analysis and waveform propagation eliminate all computation during simulation. More common models and modeling styles result in substantially simpler and thus more efficient executable simulations.

5.3 Input Desensitization

Conceptually, most conservative approaches [2] to parallel simulation require that all inputs have a known value before a corresponding evaluation occurs. Recognizing inputs which can be ignored is a well-known strategy for relaxing this requirement and thus boosting parallelism. Most earlier efforts at applying input desensitization to practical hardware description forced the designer to explicitly describe desensitization within the model. Since this extra burden adds to the already complex task of modeling, production simulators make little or no use of input desensitization.

Once temporal analysis and waveform propagation are complete, input desensitization occurs independently for each activation point in a process body. A relatively simple additional analysis identifies input values referenced along any path from a given wait statement. Once a processor begins evaluating a decision tree on behalf of a process (or processes), execution may either lead to evaluation of the process or continued suspension pending arrival of additional input values.

Often a group of trivial processes form a cluster with constant internal propagation delays. Through embedded scheduling, if a processor arrives at a given process, the inputs to that process are already defined. Combinational logic blocks are the most common example of these clusters. By desensitizing inputs to the cluster, rather than each individual gate, the probability of introducing substantial overhead due to desensitization drops.

5.4 Concurrent Evaluation

Conceptually, simulation of each elaborated process progresses through serialized phases of evaluation and suspension. For example, evaluation of a process 1 MS into a simulation conceptually completes before the simulator can begin evaluation of the process 2 MS into the simulation. On a parallel processor simulating a model with limited inter-processor parallelism, this serialization often reduces simulation speed. Concurrent evaluation is a new MinSim optimization which can eliminate this serial bottleneck.

First, temporal analysis identifies the persistent state of each (substantial) process. Aggregation of this state permits formation of one or more feedback signals representing the persistent state (shown in Figure 3). Separation of persistent state into multiple feedback signals helps to reduce or eliminate false dependencies.

For example, the persistent state representing the register file in a processor model may be split into 32 feedback pathways, one for each register in the file. This enables simulation of concurrent write operations on the register file, routed to different register addresses, at different points in the simulation time.

Input desensitization (described above) identifies situations in which the feedback pathways either do not exist (combinational logic) or are not referenced along any code pathway reachable from the current wait statement. In such situations, simulation may initiate a new evaluation (at a later simulation time) before prior evaluations complete.

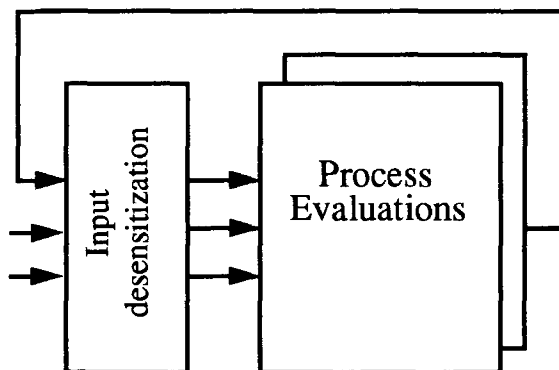

Figure 3: Concurrent Evaluation

Application of concurrent evaluation to a particular elaborated process requires that the compiler verify several properties of the process. All of the code paths executable after the resumption of a given wait statement must converge on a specific wait statement. For simplicity, the process must not include signal assignments which can result in residual projected waveform elements.

Effective use of concurrent evaluation requires the kind of tight coupling between two or more processors which is commonly restricted to shared memory. Fortunately, workstations and other MP nodes containing two or more processors are increasing in popularity.

5.5 Statement Compaction

VHDL provides a rich set of constructive mechanisms (enumerated types, arrays, records and overloading) through which the designer can customize data types to a particular abstraction level and realization technology. Compilers map these customized data types onto the set of primitive data types of the target processor. The effectiveness of this map can be key to simulation performance.

Simple rules for constructing a map from customized data types to types provided by the target processor are often less optimal than hand-tuned optimizations. For this reason, many simulation implementors hard-code multi-valued logic types, such as IEEE Standard 1164, into their simulators. Unfortunately, such hard-coding defeats the purpose of type construction mechanisms and is of little use when a model cannot use the particular hard-coded types. The spirit of VHDL dictates a more general solution in which the compiler optimizes this mapping.

Tabulation and enumeration subtyping are two key techniques used in compacting individual statements involving constructed types. Tabulation converts the subprogram body defining an overloaded function into a lookup table. By setting up the overloaded function so that its operands are scalars or aggregates with known constraints, table indices may be either scalar values or short vectors. Longer vectors may be handled through several references to the same or distinct lookup tables. Enumeration subtyping determines the set of enumerated values which actually occur in the source code or are reachable as a result of operations on values of the enumerated type. By reducing the actual set of an enumeration, the size of operator tables decreases.

The performance impact of tabulation varies substantially depending on the modeling style, time and memory available to the compiler. For some models, this technique alone reduces simulation time by more than an order of magnitude, for others it has little impact. Fortunately, tabulation and other statement compaction optimizations are essentially orthogonal to optimization techniques such as those described in prior sections.

6.0 Mapping

Mapping is a cooperative process of determining which node in the network or parallel processor a given process will simulate on. The design hierarchy provides an initial hint followed by an exchange of processes until the load is sufficiently balanced. Temporal analysis

and waveform propagation provides useful input data to the determination of static load balancing.

7.0 Scheduling

Discrete event simulators commonly use some form of a time wheel recording the time at which a future event occurs. In contrast, MinSim embeds each node's scheduler as an integral part of the executable simulation, closely resembling the levelized compiled code used by cycle simulators or some simulation accelerators.

Embedded scheduling envelops the body of each process or process collection with a dispatcher. When a processing element begins executing a dispatcher, the dispatcher examines events queued at the its inputs to determine if the associated process(es) may be evaluated over one or more time intervals. Each processing element traverses a statically determined cycle of dispatchers in an effort to advance simulation. The resulting simulation is an efficient hybrid of traditional event-driven and compiled strategies, yet preserves the semantics of an arbitrary VHDL model.

8.0 Code Generation

Following scheduling of those processes bound to a particular processor during the mapping phase, the code generation phase emits executable machine instructions directly into the compiler's address space. This code generation phase embodies instruction recognition, register allocation / optimization, peephole optimization, assembly and linking.

Earlier VHDL simulators often translated VHDL into a minimally suitable programming language such as Ada or C followed by translation (compilation) into assembly. The pathway from VHDL to executable simulation bridged several file formats (VHDL, C, assembly, relocatable object and executable object), leading to multiple parsing steps and no opportunity for propagating information backward. For example, reliance on the C compiler for function inlining, loop unrolling and constant propagation does not make the body of operator functions available for VHDL-specific optimization at the call site. HDLs such as VHDL often have backend constructs that are very different from typical programming languages, resulting in a substantial compromise in the fit between HDL and programming languages. These compromises ultimately reduce simulation performance.

MinSim's tight coupling of analysis through code generation permits very rapid, incremental modification of the simulation binary. These incremental modifications facilitate demand-based insertion of debugging control and optimizations based on actual dataflow during a running simulation.

9.0 Conclusions

Aggressive use of optimizing compiler technology, parallel compilation and parallel simulation defines the next generation of VHDL simulation technology. This paper uses a description of MinSim to provide the reader with a glimpse of the future.

10.0 Acknowledgments

This research has benefited from assistance of varying kinds from colleagues throughout International Business Machines (IBM), Carnegie Mellon's Robotics, Computer Science and Electrical Engineering Departments, The University of Minnesota's Army High Performance Computing Research Center and Computer Science Departments. Validation suites and test cases were provided by Philips Research and Semiconductor Divisions, Wright Research and Development Center (USAF), MCC, Intermetrics, Cadence, Vantage, Synopsys, Model Technologies, Convex, The University of Virginia, Virginia Tech, The University of

Cincinnati, The University of Pittsburgh and Menchini Consultancy.

10.1 References

[1] J. Willis and D. Siewiorek, "Optimizing VHDL Compilation for Parallel Simulation", IEEE Design and Test, Vol. 9, No. 3, Sept. 1992, pp 42-53.

[2] S.S. Muchnick and N.D. Jones, *Program Flow Analysis: Theory and Applications*, Prentice-Hall, Englewood Cliffs, N.J., 1981.

[3] R.E. Bryant, "Simulation of Packet Communication Architecture Computer Systems," Tech Report MIT/LCS/TR-188, Laboratory for Computer Science, MIT, Cambridge, Mass., 1977.

Readers can address correspondence to Willis at IBM, Dept. 53C, Highway 52 and 37th Street, Rochester, Minnesota 55901 or via willis@vnet.ibm.com (internet).