Journal of Parallel and Distributed Computing **62**, 468–493 (2002)
doi:10.1006/jpdc.2001.1805, available online at http://www.idealibrary.com on IDEAL®

Analysis and Simulation of Mixed-Technology VLSI Systems

Dale E. Martin, Radharamanan Radhakrishnan, Dhananjai M. Rao, Malolan Chetlur, Krishnan Subramani, and Philip A. Wilsey

Experimental Computing Laboratory, University of Cincinnati, Cincinnati, Ohio 45221-0030
E-mail: dmartin@ececs.uc.edu, ramanan@ececs.uc.edu, dmadhava@ececs.uc.edu, mal@ececs.uc.edu, skrish@ececs.uc.edu, paw@ececs.uc.edu

Received July 17, 2000; revised February 3, 2001; accepted March 7, 2001

Circuit simulation has proven to be one of the most important computer aided design (CAD) methods for verification and analysis of, integrated circuit designs. A popular approach to modeling circuits for simulation purposes is to use a hardware description language such as VHDL. VHDL has had a tremendous impact in fostering and accelerating CAD systems development in the digital arena. Similar efforts have also been carried out in the analog domain which has resulted in tools such as SPICE. However, with the growing trend of hardware designs that contain both analog and digital components, comprehensive design environments that seamlessly integrate analog and digital circuitry are needed. Simulation of digital or analog circuits is, however, exacerbated by high-resource (CPU and memory) demands that increase when analog and digital models are integrated in a mixed-mode (analog and digital) simulation. A cost-effective solution to this problem is the application of parallel discrete-event simulation (PDES) algorithms on a distributed memory platform such as a cluster of workstations. In this paper, we detail our efforts in architecting an analysis and simulation environment for mixed-technology VLSI systems. In addition, we describe the design issues faced in the application of PDES algorithms to mixed-technology VLSI system simulation.

1. INTRODUCTION

Recent advances in integrated circuit (IC) fabrication technology have allowed designers to integrate a larger number of transistors onto a microchip than ever before. The pace of technological advancement and the market demand for such technological innovations has, however, put software/hardware vendors under constant pressure to build efficient and more reliable systems. In order to succeed in such volatile environments, hardware manufacturers must focus their efforts on

0743-7315/02 $35.00

improving three major characteristics of their systems: correctness, reliability, and time to market. While all three of these factors play a major role, correctness and reliability are especially critical for the success of a product. As a result, a time- and cost-efficient testing strategy is a vital part of the system life cycle. Since designing and fabricating the hardware is a time-consuming and expensive process, a designer usually first models the hardware in software. This is a quick and inexpensive method for analyzing and testing the design. The model of the system is usually developed using a hardware description language (HDL). The model is then tested by exercising it with a set of input vectors and studying its behavior during the test. HDLs are essentially programming languages that provide special language constructs that can be used to model hardware systems. In addition, since a hardware system is inherently parallel (i.e., all the components of the system execute simultaneously), the HDL used to model the hardware must also provide constructs for modeling these parallel characteristics. The VHDL hardware description language [24] is one such hardware modeling language. In addition to basic programming language features, VHDL also provides hierarchical design capabilities. This allows the modeling of systems at various levels of abstraction. Once the system model has been constructed, the model can be simulated according to the semantics specified in the VHDL Language Reference Manual (LRM) [24]. The simulation model can then be used to analyze and verify the design.

The capacity and throughput of software-based hardware simulation tools has not kept pace with the growth of modern designs. The capacity limits of existing sequential simulation technology have already been exceeded and circuit designers cannot, in general, simulate a complete circuit design. The circuit design must be subdivided into smaller subassemblies to allow simulation and verification. This leads to difficulties in ensuring that the complete system will actually perform as expected. Furthermore, even when a system is organized as a set of smaller subassemblies, simulation times are long and frequently result in abbreviated simulation test schedules. However, the issues of capacity and large execution times can be addressed through the innovative use of parallel processing—either through special-purpose parallel hardware [3] or by deploying parallel software algorithms executing on large general-purpose multiprocessor machines or clusters of workstations [7, 10, 42, 44]. While special-purpose accelerators operate well, providing excellent speedup, their extreme cost makes software solutions on general-purpose hardware a much more attractive alternative. Specifically, the increased computing resources available in a parallel computing system (such as a cluster/network of workstations) can be utilized to help overcome the time and space burdens of hardware simulations. In addition, parallel discrete-event simulation techniques can be employed to perform a synchronized distributed simulation on the distributed execution platform. However, the application of parallel discrete event simulation (PDES) techniques to enhance the performance of circuit simulation has met with limited success. This is due to the fact that the time taken to execute an event (event granularity) is typically very small and, in general, each event that is processed will generate one or more events that must be communicated to other parallel processes (resulting in a very high communication-to-computation ratio) [5, 44].

In this paper, the process of constructing an environment for analyzing and simulating VLSI circuit designs written in a hardware description language such as VHDL is described. Specifically, we detail the issues involved in architecting and implementing a VHDL compiler (SAVANT's *scram* compiler), a VHDL simulation kernel (TyVIS), and a general-purpose distributed simulation support system (WARPED) required for large-scale simulations of VHDL logic circuit designs. In addition, we analyze the reasons for the less-than-satisfactory results of previous attempts at applying PDES techniques to distributed circuit simulation. These issues and others served as the motivation for the design of several optimizations (which we will describe in this paper). We have implemented these optimizations as part of the analysis and simulation environment. The remainder of the paper is organized as follows. Section 2 describes in greater detail the domain of parallel discrete-event simulation (focusing on digital logic simulation in particular) and briefly summarizes the large amount of related work in this area. Section 3 introduces and describes in detail the architecture of the VHDL analysis and simulation environment. After the environment is introduced, Section 4 describes how PDES techniques are used in performing parallel VHDL simulation. Section 5 introduces and describes optimizations (along with some experimental results) for improving the performance of the parallel logic simulator. Section 6 describes how this environment has been extended to simulate mixed-signal (analog and digital) circuit designs and presents the unique design challenges the mixed-signal domain presents to the simulation tool developer. Finally, Section 7 presents some concluding remarks.

2. BACKGROUND AND RELATED WORK

In this section a brief overview of PDES is presented. In PDES the system to be simulated is decomposed into *physical processes* that are modeled as *logical processes* (LPs). The simulation is composed of a set of communicating LPs that execute concurrently. LPs communicate by exchanging time-stamped messages. In order to maintain causality, LPs must process messages in strictly nondecreasing time-stamp order [26, 29]. Two basic synchronization protocols are used to ensure that this condition is not violated: (i) *conservative* and (ii) *optimistic*. Conservative protocols [14, 31] strictly avoid causality errors, while optimistic protocols, such as Time Warp [26], allow causality errors to occur, but implement some recovery mechanism.

Conservative protocols were the first distributed simulation mechanisms. The basic problem conservative mechanisms must address is the determination of "safe" events. In order to execute an event, the conservative process must first determine that it is impossible for it to receive another event with a timestamp lower that the event it is currently trying to execute. If the process can independently guarantee this, then the event is deemed to be safe and can be executed without fear of a causality violation. Processes containing no safe events must block; this can lead to deadlock situations if no appropriate precautions are taken. In order to guarantee that an event is safe for execution and to prevent deadlock, processes circulate *null messages* to inform each other about their current simulation times. Several studies

on conservative mechanisms and optimizations to conservative protocols have been presented in the literature [14, 31].

In contrast, optimistic protocols such as Time Warp [26] allow causality errors to occur, but implement some recovery mechanism. In a Time Warp synchronized simulator, each LP operates as a distinct discrete event simulator, maintaining input and output event lists, a state queue, and a local simulation time (called *local virtual time* or LVT). The state and output queue are present to support rollback processing. As each LP executes asynchronously, it is possible for an LP to receive an event from the past (some LPs will be executing faster than others and hence will have local virtual times greater than others)—violating the causality constraints of the events in the simulation. Such messages are called *straggler* messages. On receipt of a straggler message, the LP must roll back to undo some work that has been done. Rollback involves two steps: (i) restoring the state to a time preceding the time-stamp of the straggler and (ii) canceling any output event messages that were erroneously sent (by sending *anti-messages*). After rollback, the events are re-executed in the correct causal order.

In general, parallel simulation of logic circuits is implemented by identifying each gate of the logic circuit as a parallel activity and modeling each gate as a logical process. Logical processes operate as individual discrete event driven simulators that are synchronized using either (i) a centralized time distribution mechanism, (ii) a distributed, conservatively synchronized mechanism [31], or (iii) a distributed, optimistically synchronized mechanism [26]. Typically, the LPs are statically partitioned and assigned to the parallel processes, and event information is exchanged by time-stamped messages. Each processed event corresponds to a bit value change and the computational requirements (for each logic gate) are minimal. This level of division results in an extremely fine grain of parallelism. As a result, the number of parallel tasks is extremely high. The time for a single execution of an LP simulating a logic gate can be dwarfed by the communication, synchronization, and operational overhead of the simulation engine. With such fine-grained parallelism, the costs of sending or receiving messages from/to other LPs can be equivalent to or even higher than the costs of processing a event. In conservatively synchronized parallel simulations, the sheer number of null synchronization messages that must be sent for each event cycle can be daunting [44]. If the simulation system is synchronized optimistically, the operational overhead burden can be of even more concern. Specifically, an optimistically synchronized (using for example the Time Warp [26] protocol) simulation requires each process to store history information about its previous states, inputs, and outputs, in case of a rollback.

Several empirical studies have shown that the total amount of parallelism available in low-level hardware simulations is limited. Soulé and Blank [43] investigated the available parallelism in a model of the 8080 microprocessor, consisting of 3439 gates, or LPs. Their study showed that, at any given time segment during the simulation, only 0.1–0.5% of the LPs were active and that only 10–20% of the LPs were active during each microprocessor clock cycle. Likewise, Bailey studied the available parallelism of nine different models ranging in size from 208 to 61,600 transistors [5]. In these studies, Bailey reported that the percentage of LPs active at a given moment in simulation time (i.e., the average parallelism) ranged from 0.11%

to 5.9%. Bailey also performed scalability studies on a selected set of models to determine the change in available parallelism as the size of a given model increased. The results did not illustrate any significant variation in the percentage of LPs active at a given simulation time. However, there are two factors in these analyses that need to be considered. First, the aforementioned numbers are parallelism percentages, not absolute parallelism. Even the smallest simulation described by Bailey was measured as having an average of 9.8 LPs active at a given time. Larger models ranged up to 540 active LPs. The second factor is that the aforementioned measurements were taken on conservatively synchronized simulators. Unlike a conservatively synchronized simulator, an optimistically synchronized system is not restricted to simulating serially through simulated time; different LPs can be executing events at different simulation times simultaneously. In defense of this idea, Briner [10] reports that by allowing the LPs to advance asynchronously, higher levels of parallelism are available. In certain models, he reports as much as 50 times more parallelism is available in an optimistic simulation than in the conservative simulation of the same model. Bauer *et al.* [8, 9] report similar results. The aforementioned issues and results motivated the development of the analysis and simulation environment described in this paper. Specifically, the parallel simulator employs the Time Warp protocol to synchronize the distributed LPs in the simulation. The following sections describe the analysis and simulation environment in greater detail.

3. THE ANALYSIS AND SIMULATION ENVIRONMENT

In this section, we detail the design and implementation of an environment for analyzing and simulating VLSI circuit designs written in VHDL. Key to the analysis part of such an environment is the design compiler. A design compiler can be broadly divided into three functional units: (a) the front-end parser (responsible for parsing and syntax checking); (b) the semantic analyzer (responsible for validating the semantics of the input); and (c) the back-end code generator (responsible for generating simulator specific code). As part of the SAVANT project, Wilsey *et al.* [51] designed and developed the front-end parser and semantic analyzer (called *scram*). The *scram* analyzer performs the parsing, the syntax checking, and the semantic checking and transforms the source into an intermediate representation (IR) for further processing. The back-end to this compiler is a code-generator, which generates C^{++} code from the intermediate representation. In order to correctly handle and simulate VHDL models, a VHDL simulation kernel (called TyVIS) was developed on top of a general purpose parallel discrete-event simulation kernel called WARPED [37].

Figure 1 illustrates the architecture of the analysis and simulation environment. The input (model descriptions in VHDL) is transformed into a basic intermediate representation (IR) by the *scram* analyzer. The IR is constructed during the syntactic and semantic analysis phase. Complex constructs in the IR are then reduced to equivalent simpler structures during the transmute phase. Using this reduced IR, the code-generator generates C^{++} code suitable for simulating the design specified

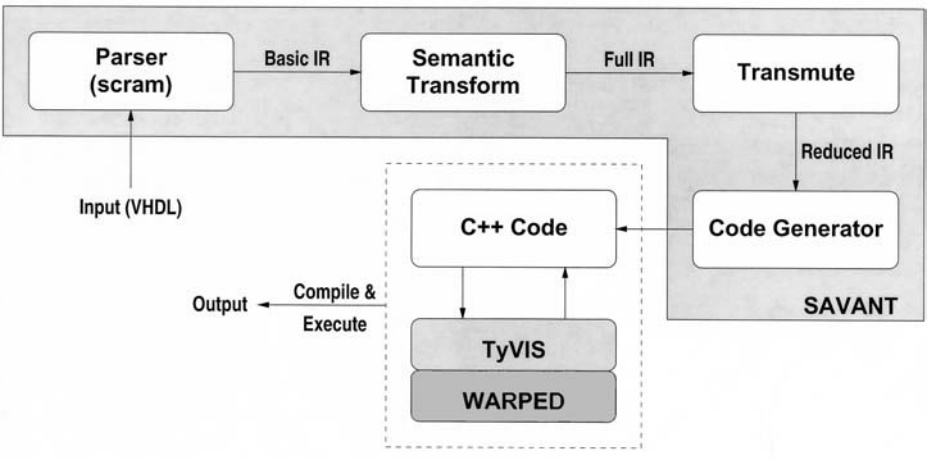

FIG. 1. Components of the VHDL analyzer and simulator.

by the input VHDL model. The generated code is then compiled with the TyVIS and WARPED simulation libraries to obtain an executable. When this executable is executed, it performs a parallel discrete-event simulation and produces the required simulation output of the input VHDL model. The following subsections describe each of the major components of the environment in greater detail.

3.1. The *scram* Analyzer

The front end of the analyzer is responsible for parsing the input VHDL model, checking for semantic validity, and building the intermediate representation. The *scram* analyzer (constructed by supplying the VHDL grammar to PCCTS [34]) creates a basic intermediate structure from the input VHDL. This phase identifies all the syntactic and semantic errors in the input and generates appropriate error messages. Once a syntactically valid VHDL input has been parsed, it is analyzed for semantic validity.[1] During this phase, the intermediate representation is semantically transformed to the appropriate structures. For instance, an *indexed name* created by the parser may be an array object, a subprogram call with arguments, or a subprogram call without arguments which returns an array object. Such differences can be determined only during the semantic analysis. The intermediate representation is transformed appropriately to reflect this semantic information.

The intermediate representation generated by the *scram* analyzer conforms to the AIRE (advanced intermediate representation with extensibility) specifications [50]. This specification defines a set of interface classes for representing the corresponding constructs in VHDL. The AIRE specification is hierarchically defined and our implementation is in C++. The properties of these classes are specified by member objects and public method interfaces. The specification may be extended to add functionality by inserting user-defined classes into the hierarchy. The VHDL

[1] While the syntactic and semantic phases can be implemented as separate phases in a design compiler, there is no clear demarcation between these phases in the *scram* analyzer. Both of these phases are tightly coupled.

analyzer and code generator have been implemented as one such extension to the AIRE specifications. Further details on the AIRE is available in the literature [50].

3.2. The Code Generator

The intermediate representation (IR) created by the analyzer is in the form of a class hierarchy (i.e., a parse tree). The information contained in the input VHDL source code is embedded into this representation in such a way that the code generator can access it easily. Beginning with the *design file* (the top-level construct representing the file that is being analyzed), the code generator traverses the IR and writes TyVIS compliant C^{++} code suitable for simulating the structures modeled by the input VHDL.

The code generator is built as a back end to the *scram* analyzer. Various code generation methods defined in the extension classes implement the code generation for the corresponding classes. The object-oriented design of the TyVIS simulation kernel makes it possible to perform code generation by recursive descent. For example, an architecture declaration in VHDL, built as a class in the intermediate representation, contains information about declarations in the declarative region and the concurrent statements contained in it as member objects. The code generation of an architecture declaration initiates the code generation of the concurrent statements in it (recursively). This way, code generation is invoked for every structure in the intermediate representation starting from the *design file*, which represents the VHDL file being analyzed, to every *concurrent statement*, *sequential statement*, *declaration*, and so on until the leaf elements are reached in the parse tree. Some of these structures may be rewritten as combinations of other structures without any difference in functionality. Such structures are *transmuted* to their equivalent alternative structures (according to the rules specified in the LRM). This is done so that code generation needs to be performed only for a minimal set of classes.

The basic philosophy behind code generation has been to: (a) keep the code generator simple; (b) minimize the code that is generated; and (c) make the generated code extensible. Most of the generated code is in the form of class instantiations or calls to functions in order to minimize the code that is published. These classes and functions are defined in a class library associated with the TyVIS kernel. The object oriented design of the simulation kernel makes it extensible and reconfigurable. The user can add/modify/remove any optimizations/implementations in the kernel by defining another class and providing the specified interface. Such a design has been governed by the basic aim of this research which has been to study and improve the performance of VHDL simulation through the novel use of PDES techniques.

3.3. The Simulation Subsystem

The simulation system has been built in two layers: the VHDL-specific layer called TyVIS, and a general-purpose parallel discrete-event simulation kernel called WARPED. The WARPED [37] simulation kernel provides the functionality to develop

applications modeled as discrete event simulations. Considerable effort has been made to define a standard programming interface to hide the details of the simulation algorithm from the application interface. All simulation specific activities such as state saving, are performed automatically by the kernel without intervention from the application. Consequently, an implementation of the WARPED interface can be constructed using either conservative or optimistic parallel synchronization techniques; furthermore, the simulation kernel can also exist as a sequential kernel. In fact, the current software distribution of WARPED includes both sequential and parallel implementations.

The parallel implementation of WARPED is an optimistic discrete event simulator based on the Time Warp [26] synchronization paradigm. The simulation is partitioned into various concurrently executing entities called *logical processes* (LPs). A logical process interacts with others by exchanging events. Different LPs may be assigned to different processors, thus distributing the simulation across a network of workstations. Events are sent between LPs using message send and receive calls compliant with the MPI message passing standard [23], a portable communication layer. Event messages between LPs hosted on the same processor are exchanged directly, without routing the event message through the network. On the arrival of an event, the process responsible for handling that event is invoked and executed. This execution may result in further events being generated. Thus, simulation proceeds by processing events and advancing the simulation time. The distributed nature of the simulation complicates certain basic functions of the simulation, such as computing the time up to which simulation has proceeded and the termination of simulation. Various distributed algorithms and optimizations [37, 38] are built into the WARPED kernel to implement these tasks. Another advantage of this design is extensibility, since the TyVIS kernel is developed on top of the WARPED kernel without any modifications to the latter (allowing separate development on either kernel).

The TyVIS simulation kernel provides the functionality necessary to implement a VHDL simulator. Since most of the basic simulation requirements, such as scheduling and synchronizing are handled by WARPED kernel, these are *inherited* by the TyVIS kernel (by inheriting the C^{++} classes in TyVIS from the corresponding classes in WARPED). However, this alone is not sufficient for a VHDL simulation kernel. It requires features such as *elaboration*, a VHDL type system, *signal propagation*, file I/O, wait statements, and several other VHDL-specific features. TyVIS extends the basic simulation kernel to provide these additional features. The next section discusses the implementation of these features in detail.

4. APPLYING PDES TECHNIQUES TO VHDL SIMULATION

Achieving a VHDL simulation is by no means a simple task. A wide variety of features such as support for concurrency, a powerful type system, data sharing between concurrent objects (signals), and process suspension and resumption need to be supported by the underlying simulation kernel. The TyVIS VHDL simulation kernel provides support for the aforementioned features by taking advantage of

object-oriented design features provided by C++ and making novel use of the WARPED parallel discrete-event simulation kernel. This section provides an overview of the components of TyVIS and the issues involved in simulating a VHDL design.

One of the primary reasons for developing a software model of a system is to simulate it. This typically involves three phases: *analysis*, *elaboration*, and *execution*. As described earlier, the analysis phase involves examination of the VHDL description and detection of syntactic and static semantic errors. The whole model of the system need not be analyzed at once. Instead, it is possible to analyze *design units* such as entity and architecture body declarations, separately. For example, consider the VHDL model illustrated in Fig. 2. The VHDL source for the model (design units 1 and 2) and the test bench (design units 3 and 4) used to test the model is stored in a single design file (inverter.vhdl). If the analyzer finds no errors in the design unit, it creates an intermediate representation of the unit and archives it in a library. The second phase (elaboration) in simulating a model is the process of walking through the design hierarchy and creating all of the objects defined in declarations. The final product of design elaboration is a collection of signals and processes. A model must be reducible to a collection of signals and processes in order to simulate it. A VHDL design can be built by hierarchically decomposing the design into components and interconnecting them. Reusable components can be developed and instantiated over and over again. For example, in Fig. 2 an inverter gate is implemented and instantiated as a component (in architecture *test* of the *testInverter* entity). In addition, local signals (*myInput* and *myOutput*) are connected to the inverter gate's *input* and *output* signals (through the *port map* declaration). By instantiating this component several times and connecting it in different configurations, various circuits can be built. These hierarchical design structures help in

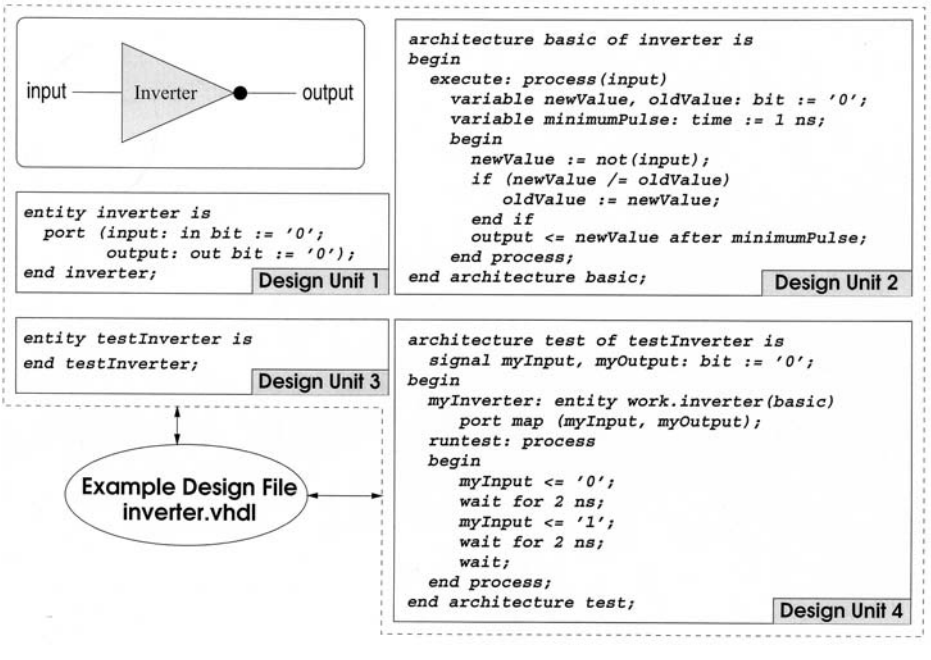

FIG. 2. VHDL Model of an inventer.

organization of the design but not in its simulation. The process by which hierarchical structures are flattened and made into a collection of connected processes is called the elaboration of the design hierarchy. The third phase of simulation is the execution of the model. In discrete-event simulation, the passage of time is simulated in discrete steps, depending on when events occur. At some simulation time, a process may be stimulated by changing the value on a signal to which it is sensitive. For example, in Fig. 2, the *execute* process in *design unit 2* is sensitive to changes in the signal named *input*. When the signal *myInput* is assigned a new value, the *input* signal of the inverter gate also get as signed the new value. This triggers the process and the process may now schedule new values to be assigned to signals at some later simulated time. This process is know as "scheduling a transaction" on a signal. If the new value is different from the previous value on the signal, an *event* occurs, and other processes sensitive to the signal may be resumed.

Simulation of a VHDL model starts with an *initialization phase*, followed by repetitive execution of a *simulation cycle*. During the initialization phase, each signal is assigned an initial value. The simulation time is set to zero, then each process instance is activated and its sequential statements are executed. Execution of the process continues until it reaches a wait statement, which causes the process to be suspended. During the simulation cycle, the simulation time is first advanced to the next time at which a transaction on a signal is scheduled. Second, all transactions scheduled at this time are executed. This may cause events to occur on some signals. This causes all processes that are sensitive to these signals to be resumed and executed until they reach a wait statement and suspend. When all the processes have suspended, the simulation cycle is repeated. When no further transactions are scheduled for execution, then there is nothing left to be done and the simulation can terminate.

4.1. Implementing Elaboration in TyVIS

The VHDL LRM defines two kinds of elaboration, *static* and *dynamic*. The elaboration of the design that can be performed just before the commencement of the simulation, such as propagation of generic constant values and computation of the net list of the signals, is called static elaboration. Certain declarations, such as type declarations in subprograms and interface constants in subprograms, cannot be statically determined at compile time. These constructs have to be dynamically elaborated during the execution of the model. For this reason, TyVIS defines a third kind of elaboration, *run-time elaboration*, which occurs just before the simulation of the model. Instructions necessary to elaborate the design are embedded into the generated code. Execution of the code causes the complete elaboration of the design before simulation commences. The information necessary for execution of the model is gathered during elaboration and passed on to the simulation kernel.

The run-time elaboration scheme in TyVIS combines a top-down and a bottom-up approach. The design hierarchy is elaborated in three phases, the *instantiate* phase, the *netinfo* phase, and the *connect* phase. The components and processes in the design are instantiated first in a top-down approach in the instantiate phase. The netinfo phase is performed hand in hand with the instantiate phase. At the end

of the instantiate phase of every design unit, its netinfo phase is initiated. When a component instantiation statement is elaborated, it fills in the information related to the signals it reads and drives. This information is then passed up the hierarchy to the component that instantiated it. The component in turn fills in the local information. This process is repeated recursively until the topmost design unit is reached. At this point all the information about all the signals (in the form of an elaboration tree data structure) is present in the topmost design unit. It then initiates the connect phase of the elaboration, where the complete elaboration tree is passed on to the instantiated components and processes which record the necessary data. At the end of the connect phase, the elaboration of the model is complete, and simulation can begin.

A two-phase approach (instantiate and netinfo) is sufficient to elaborate the entire design if only an uniprocessor (sequential) simulation is desired. However, for a distributed simulation, the connect phase is necessary since the elaboration information needs to be distributed to each individual VHDL process. Distribution of elaboration information is necessary so that the distributed processes resident on different processors can access and record information about processes local to this processor. During simulation, only the part of the elaboration tree that is pertinent to the set of processes resident on this processor is accessed. This translates to shorter access times as only a part of the entire data structure is searched and accessed. At the end of the connect phase, the signal flow in the VHDL circuit description is available in the form of a net-list. This net-list represents the communication topology of all the simulation objects.

4.2. Performing the VHDL Simulation

Since VHDL is a concurrent programming language, it is an ideal application for parallel simulation since it can exploit the inherent parallelism. Furthermore, a close inspection of the VHDL simulation semantics would reveal that signals and process scheduling are event based interactions. This would suggest discrete event simulation as the ideal simulation paradigm. Putting these together, the use of WARPED, an optimistically synchronized parallel discrete event simulation framework, as the underlying simulation kernel seems to be the correct choice. In addition, the optimistic algorithm based on Time Warp provides additional speculative processing capability that could potentially exploit additional parallelism and provide additional speedup. Of course, an incorrect prediction could result in an inferior performance, but any good partitioning and load balancing algorithm can alleviate this problem. Since WARPED is a distributed discrete-event simulation kernel, it can be used to execute large simulations on a network of small and inexpensive workstations with moderate memory and computational capacities. The kernel has been carefully implemented to make sure that it is independent of both the platform and the network. The portability of the kernel assures that the simulation may be executed on a network of *small* workstations, or on a large multiprocessor machine.

At the end of elaboration, the VHDL model is ready to begin simulation. Simulation begins by executing all the processes once in the initialization phase. This

results in certain events being generated, which are maintained in the *input queue* of the corresponding process by WARPED. These events are scheduled in a lowest time-stamp first (LTSF) order. A process is scheduled for execution when it has an event that requires processing. The process examines the events in its input queue and processes them as required. Then, depending on the conditions for the resumption of that process, either it resumes execution of the sequential statements in it, or returns control back to the underlying simulation kernel. Any VHDL specific activity is handled by the TyVIS kernel while any optimistic simulation activity (such as rollback) is handled by the WARPED kernel.

4.2.1. Issues in Parallel VHDL Simulation

After elaboration, the hierarchical model is now a set of VHDL processes and signals. Each VHDL process is mapped to a WARPED logical process. Similar to the way execution of a logical process changes the state of the logical process in a discrete-event simulation, the execution of sequential statements in a process changes the process's state by modifying the values of the variables, signals, or files associated with it. Changing the value of process variables is handled by invoking a TyVIS kernel method with the variable in question and the new value of the variable as the method's arguments (*variable assignment*). The result is a change in the variable's value. However, signal assignments and file I/O are complex because of the distributed nature of the simulation kernel. A signal contains a value that is shared between concurrently executing processes. Any change to the signal value has to be propagated to other processes that are sensitive to this signal. This is achieved by scheduling an event for the appropriate processes after a prespecified delay.

The simulation time, a measure of the progress of simulation, is updated by the kernel as events are processed. Simulation time of a process may be set back by the kernel to "roll back" from an incorrect computation. This is a result of the optimistic characteristic of Time Warp. The states of the processes have to be saved periodically to permit it to roll back to a state in the past. However, the global simulation time, measured by the progress of the Global Virtual Time (GVT), increases monotonically, guaranteeing simulation progress. File I/O also requires special attention because of the optimistic and distributed nature of a Time Warp synchronized simulation. File types in VHDL provide a method of writing data to the persistent storage devices in the computer, generally a file system. Many of the frequently used methods are predefined for all the file types, but a user can add more properties to the file type by defining subprograms. The following two questions have to be answered to implement file type objects in TyVIS/WARPED:

- Due to the distributed nature of WARPED, how are simultaneous operations to the same file type object handled?
- The execution of an event is not confirmed until it is committed. Therefore as a result of the execution, I/O can not be performed immediately. How are file operations handled correctly?

The answer to the first question is obtained partly from the VHDL LRM. If multiple file objects are associated with the same external file, the language does not specify any order in which operations on the file are performed. Since this order is not defined, the scheduling of the file operations can be safely neglected by TyVIS. It then becomes the responsibility of WARPED to define this ordering. The problem raised by the second question can be solved by maintaining separate file queues for each of the file type objects in the system. Each of these file queues becomes an additional simulation object. Since file types declared in subprograms are elaborated dynamically, file queues have to be dynamically instantiated and manipulated to accommodate dynamic elaboration of file type declarations. As a result, after dynamic elaboration, the number of simulation objects in the simulation may vary depending on how many file queue objects were instantiated. The commitment of the events representing file operations are managed by WARPED. Since the WARPED kernel uses a GVT estimation algorithm to determine when it is safe to garbage collect old history items, the estimated GVT value is used to commit irreversible file I/O operations.

In some senses, performing a VHDL simulation using the TyVIS and WARPED simulation kernels is similar to performing a process-oriented [32] simulation. Process-oriented views are useful for modeling but are difficult to implement efficiently in a simulation system [35]. The simulation of a collection of VHDL processes (post-elaboration) can be viewed as implementing a process-oriented simulation. A key issue in this type of simulation is providing support for the capability of invoking *wait* statements over nested procedure calls at any point in a procedure body. A wait statement causes the simulation to suspend processing of the sequential statements following it for a given time period and resume when any one of the signals in its sensitivity list (if any) become active and a condition clause (if any) evaluates to true. Once the time period expires, it resumes irrespective of signal sensitivity or the condition. If the time period expression is omitted, the wait never resumes (other than due to activation of a signal it is sensitive to or the condition clause evaluates to true). An implementation of the wait statement must be able to suspend execution at any point in a process or a procedure, and resume the execution from the same point depending on the conditions or when the time has expired, whichever occurs first. In addition, an implementation of the wait statement in WARPED must be able to rollback by restoring a previous state.

The execution of a wait statement proceeds in three distinct phases. First, when the wait statement is encountered, an event (WAIT_MESSAGE) is sent by the process to itself. This event will be scheduled after a prespecified delay as illustrated by Fig. 3a (step 1). To identify the wait statements within a process uniquely, an ID is assigned to each wait statement. The ID (or label) of the wait statement that the process is waiting on and the time at which it was executed is recorded so that execution can be resumed from the same point. The second phase starts with the arrival of a wait event. When this event is received by the process after the delay interval, a flag in the state of the process is set indicating that the time delay has occurred (step 2). Activity of signals is detected by examining a list of flags maintained in the kernel.

The third and final phase (Fig. 3a, step 3) takes place when the body of the process is executed. Irrespective of whether the process resumes or remains

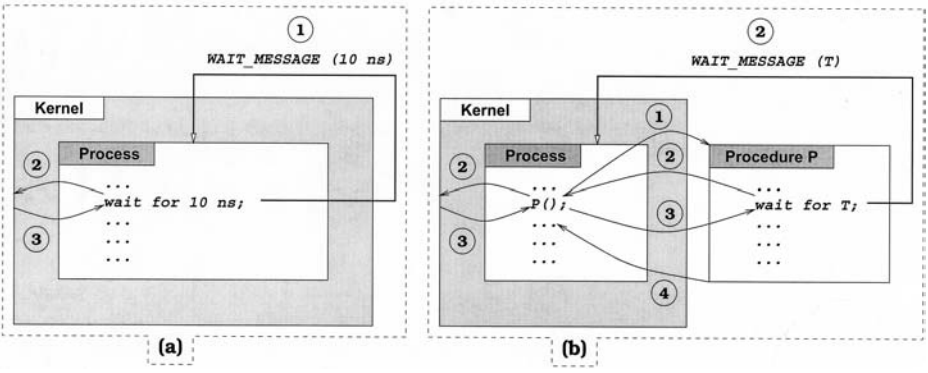

FIG. 3. Handling *wait* statements in processes and procedures.

suspended, the body of the process is executed. The first step is to examine the *waitLabel* variable in the state, which indicates the wait statement on which the process is waiting on. Depending on this, it makes an unconditional jump (goto) to the corresponding instruction. Here, a call to the method *resumeWait* is made with the ID of the wait statement and the value of the condition expression. This method determines, according to the rules specified in the VHDL LRM, if the wait resumes or continues suspension. On resumption, the statements following the wait statement are executed until the process suspends again. If the wait does not resume, control is transferred back to the kernel.

VHDL also allows a wait statement in the body of a procedure. A few problems arise when such a wait statement is implemented in the TyVIS kernel. The control has to be transferred back to the kernel, which cannot be achieved with just a return statement, as in the case of a wait statement in a process. The calling path has to be backtracked all the way back to the kernel. To resume from the wait statement, control has to be transferred to the wait statement from the kernel. The path taken by the control transfer has to be the same as the natural call path. The temporary variables in the procedures must be stored in the state of the process from which the call was initiated to take care of roll back and state saving. Finally, the same procedure may be invoked by different processes, and when the wait statement returns, control should be transferred to the correct process.

This problem is solved in TyVIS by maintaining a "call stack" in the process's state which records information regarding the path of the call and the value of the local variables in the procedure. Figure 3b illustrates the control transfer from the process to the procedure call and vice versa (steps 1 through 4). When a wait statement is executed in a procedure, the *waitLabel* variable in the process's state is set to WAITING_IN_PROC. The local variables in the procedure and the location of the wait statement are saved in the call stack. The control is then transferred back to the kernel. In case a procedure body contains a procedure call, the local variables and the location of the procedure call are stored in the call stack. These values are required if a wait statement is executed in the body of the called procedure. When a wait in a process statement is detected in the final phase of the execution of the wait statement, it retrieves the path the process took to the wait statement and the local variables of the procedures in the path from the call stack. Finally, while the call

stack has been used specifically to implement wait statements in VHDL procedure statements in the TyVIS kernel, Perumalla and Fujimoto [35] use the same principle in the simulation of telecommunication models (written in the TeD language [36]). They call it *stack reconstruction*.

5. OPTIMIZING PARALLEL VHDL SIMULATION

Unfortunately, the application of parallel discrete event simulation (PDES) techniques to speed circuit simulation has met with limited success. This is due to the fact that event granularities are very small and, in general, each event processed will generate one or more events that must be communicated to other parallel processes (resulting in a very high communication-to-computation ratio) [5, 44]. Thus, parallel logic simulators have yet to produce acceptable performance. In this section, we present several methods for overcoming these performance issues in parallel simulation. Using the SAVANT/TyVIS/WARPED tool suite, we have developed several optimizations to speed up the combined performance of the simulation system. Specifically, we investigate the following optimizations: (a) partitioning; (b) rollback relaxation; and (c) fine-grained communication optimizations. The following sections detail each of these optimizations.

5.1. Partitioning

To extract better performance from parallel logic simulators, partitioning techniques are necessary [16, 41, 45]. The partitioning techniques can exploit the parallelism inherent in either (i) the simulation algorithm or (ii) the circuit being simulated. The amount of parallelism that can be gained from the former method is limited by the algorithm used for simulation. The latter method attempts to improve performance by dividing the circuit to be simulated across processors. Hence, during simulation, the workload is distributed and the concurrency and parallelism in the circuit are exploited. Its success is bounded by the amount of parallelism inherent in the circuit and the number of processors available for simulation. Partitioning algorithms, in general, concentrate on achieving speedup by improving concurrency, minimizing inter-processor communication, and balancing the processor workload based on the circuit being simulated [2].

Several techniques have been developed to partition logic circuits for parallel simulation [2, 16, 28, 41, 45]. The algorithms address various issues related to concurrency, communication and load balancing. In addition to investigating and implementing existing partitioning algorithms, we have developed a new partitioning algorithm based on a multilevel heuristic. The new multilevel approach [46] to partitioning attempts to optimize the concurrency, communication, and load balance factors by decoupling them into separate phases. The multilevel algorithm for partitioning has been studied and analyzed in [27] and has been shown to produce high-quality partitions (measured with respect to edges cut, i.e., the number of edges that cross partition boundaries) over several partitioning algorithms such as the inertial and the spectral bisection algorithms. The complexity of the multilevel algorithm is $\mathcal{O}(\mathcal{N}_{\mathcal{E}})$, where $\mathcal{N}_{\mathcal{E}}$ represents the number of edges in the

TABLE 1
Characteristics of Benchmarks

Circuit	Inputs	Gates	Outputs
s5378	35	2,779	49
s9234	36	5,597	39
s15850	77	10,383	150

circuit graph making the multilevel partitioning technique a fast linear time heuristic. Further details about the multilevel partitioning algorithm are available in the literature [46]. The partitioning algorithms that have been implemented and profiled in the SAVANT/TyVIS/WARPED simulation system include (a) random, (b) topological, (c) depth first search (DFS), (d) cluster (or breadth first search), (e) fanout cone, and (f) our new multilevel algorithm.

An empirical evaluation of the performance of different partitioning strategies in the SAVANT/TyVIS/WARPED environment was carried out using three of the ISCAS '89 benchmarks [11]. The characteristics of the benchmarks used in the experiments are shown in Table 1. All the partitioning algorithms failed to provide speedup for benchmarks with less than 2500 gates, since such models were small enough for the sequential simulator to outperform the parallel version. The parallel simulation experiments were conducted on eight workstations inter-connected by fast Ethernet. Each workstation consisted of dual Pentium II processors with 128 MB of RAM running Linux 2.2.12. The experiments were repeated five times and the average was used as the representative value in all the characteristics. It was observed that the multilevel algorithm outperformed all other partitioning algorithms when more than four nodes were involved in the simulation. The performance of the Cluster and DFS algorithms deteriorates with increased number of

TABLE 2
Simulation Time (in Seconds) for the Different Partitioning Algorithm

Circuit	No. of nodes	Random	DFS	Cluster	Topological	Cone	Multilevel
s5378	2	166.44	118.72	97.45	128.63	166.54	*91.66*
(149.96)	4	116.11	84.80	83.28	331.45	113.11	*84.07*
	6	131.95	76.12	96.86	194.34	96.07	*63.61*
	8	101.89	81.09	78.62	152.91	76.56	*52.94*
s9234	2	675.07	473.90	417.63	577.14	701.10	*529.39*
(651.24)	4	496.30	424.41	322.02	434.85	502.60	*341.84*
	6	520.80	320.98	373.41	539.59	414.65	*316.96*
	8	383.32	489.97	415.02	360.90	351.35	*290.31*
s15850	4	2090.82	1279.19	1317.28	2272.62	1832.24	*1043.43*
(2154.21)	6	1434.79	906.08	1351.17	1439.99	1363.40	*943.91*
	8	1407.33	947.64	1215.64	2735.07	1176.36	*864.03*

nodes due to lack of concurrency. The lack of concurrency also increases the number of rollbacks in the simulations. The performance of the Topological algorithm is limited due to increased communication overheads; more signals are split across partitions for concurrency. The parallel simulation execution times for all the benchmarks have been tabulated in Table 2. Underneath each circuit's name, the time taken for a sequential simulation is also shown (in brackets). As illustrated in the table, the multilevel strategy performs better than other strategies when the number of processors employed lies between 8 (four workstations) and 16 (eight workstations). When two nodes were employed to simulate the s15850 model, the simulations ran out of memory and hence the results are not presented in Table 2.

5.2. Rollback Relaxation

Rollback relaxation [48] is an optimization for Time Warp simulators that reduces the cost of state savings and can potentially reduce rollback costs. Two important problems with Time Warp optimistic simulation are the impact of state savings and rollback costs. As discussed in [20], these problems are tightly interwoven. If the state is saved after processing each event, then empirical data indicates that rollback costs are less than 1/10 of the cost as the original forward computation. However, saving the state after every event is particularly costly, leading to suggestions to (i) alter the frequency of state savings [17], (ii) use hardware modules to assist in state savings [21], or (iii) save only the incremental state change caused by event processing [33]. While these approaches reduce the over head in terms of space, they can dramatically increase the cost of rollback. Thus, as parallel simulations increase in complexity and size, rollback costs may quickly become significant.

The impact of large rollback costs must not be overlooked. More precisely, some analytical models have shown that if rollback is expensive, then the progress in simulated time per unit of real time can decrease as the simulation proceeds and the time to perform rollback can increase exponentially, leading to unstable behavior [30]. Thus, it becomes important to carefully control increases in rollback costs. Rollback relaxation takes advantage of the fact that some processes do not contain internal state and thus they do not strictly require the maintenance of a state queue. Processes with internal states that are potentially live between processing distinct events must still be processed as usual: However, memoryless processes (those without states) do not require state queues and can be managed so that, upon receipt of a straggler message, minimal recomputation is necessary. These memoryless processes allow the advantages of rollback reevaluation [22, 49] without its cost.

The notion of rollback relaxation arises with the classification of logical processes into two categories, namely *state-full processes* and *memoryless processes*. More precisely, a set of processes in a simulation system can be partitioned into two nonintersecting subsets:

(a) *Memoried Process:* The set of processes whose output events are defined as a function of both input event values and internal state values. In such processes,

event processing may use internal state information from previous event processing activities to produce an output event.

(b) *Memoryless Process:* The set of processes whose output events are completely determined by the values of its input events. Event processing by a memoryless process will never use internal information from past event processing to produce an output event. Note, a process with internal state variables can qualify as a memoryless process provided that all of its internal state variables are not live[2] and the process has a single entry point for event processing.

While the concept of rollback relaxation can be used with any Time Warp Simulator, we have implemented rollback relaxation as it relates to a specific instance of a Time Warp simulator. In particular, we have implemented rollback relaxation in a parallel logic simulator. Each logical process will be assumed to correspond to a physical digital system component and events will denote changes to values exchanged between hardware components. This restriction allows a simple introduction to rollback relaxation. The generalization to other applications is trivial and ensured because of the use of standard compiler analysis techniques. Initial implementations of rollback relaxation in a parallel logic simulator resulted in a significant improvement in the performance of the parallel logic simulator. On average, a speedup in the range of 15–40% was obtained when rollback relaxation was employed in the parallel logic simulator. A similar technique for reducing the state saving costs called *reverse computation* is employed by Carothers *et al.* [12].

5.3. Fine-Grained Communication Optimizations

Several strategies for minimizing the communication times of distributed applications (such as parallel logic simulators) have been explored. The application must be partitioned so that processes that communicate most often are mapped to the same physical processor [6]. In most cases, it is possible to hide some of the communication cost by overlapping it with the computation [23]. While these (and other) strategies result in improving the performance, they are specific to the application and the communication subsystem. In addition, with the advent of cheap and powerful hardware for workstations and networks, a new cluster-based architecture for distributed processing applications has been envisioned. Clearly, fine-grained parallel logic simulators that communicate frequently are not the ideal candidates for such architectures because of their high latency communication costs. So what can be done to improve the communication latencies of such cluster-based architectures? Depending on what kind of cost–performance trade-offs are required, the communication latency can be improved along the following three dimensions: (a) using faster network hardware; (b) using more efficient communication libraries; and (c) using communication optimizations that exploit the simulation's inherent communication characteristics. A number of experiments [15, 39] have been carried out on the WARPED simulation kernel to reduce the effect of communication latency on the performance of the simulation. Specifically, Chetlur

[2] The definition of liveness in this context is derived from compiler optimization theory. Effectively, a variable is live if its value is used in the program before it is defined.

et al. [15] have designed and developed a *message aggregation* facility for WARPED that aggregates application messages into larger chunks before sending them across the network. In several experiments carried out with different models (queuing and logic circuit models), speed-ups of over 50% were reported [15, 39].

6. EXTENDING PARALLEL LOGIC SIMULATION TO MIXED-TECHNOLOGY SIMULATION

In this section, we first describe the domain of mixed-technology simulation and then present how the SAVANT/TyVIS/WARPED environment has been extended to handle mixed-technology simulation. Mixed-technology (or mixed-mode) simulation is the simulation of a combination of two distinct mathematical models. Zeigler [52] classifies these two mathematical models as the class of *discrete event models* and the class of *differential equation models*. Both models are characterized by their behavior. Differential equation models exhibit continuous changes in time and state; thus, the time derivatives (i.e., rates of change) are governed by differential equations. Cellier [13] reports that differential equation models, in general, are simulated as a discrete-time model on a digital computer to avoid infinitely many state changes. Values in between the discrete-time stamps are then interpolated. On the other hand, discrete event models exhibit discrete changes in time and state. State changes only occur at individual time points (events) and are always discontinuous. No changes in state occur in between these time points. Due to the difference in the two underlying models, mixed-mode simulation results in two different simulation paradigms. There are many simulation algorithms, each with its own properties, that can simulate either differential equation models [4] or discrete event models [14]. However, these algorithms cannot be used for mixed-mode simulation as the two paradigms have to be combined. Figure 4 shows an example of a mixed-mode simulation model and its results. The model contains two discrete-event processes with signals (A, B) and one process that solves differential equations, with an internal signal. The model description in Fig. 4a includes interface functions (IF) between the two domains. The IF maps discrete signals to continuous signals and vice versa. This is illustrated by the vertical lines in Fig. 4b. Although the IFs are critical for mixed-mode simulation (as two different signal types are being connected), they introduce new problems during design and simulation [40]. For example, how is a digital signal described at the behavioral level connected to the same signal in the analog domain (described at the electrical level)? Hence, the IFs are not only crucial for correct modeling, but also act as a bridge between the two different simulation paradigms.

It is obvious that for correct simulation of a combination of these models, inter action via the interface functions will have to be supported. This implies that the simulation is divided into distinct simulation time[3] intervals in which no interaction between the two simulation models occurs, leaving distinct points where the interaction (or communication) does takes place. To model this interaction, the

[3] Every discrete event simulation contains a state variable called the simulation clock to model the flow of time during a simulation.

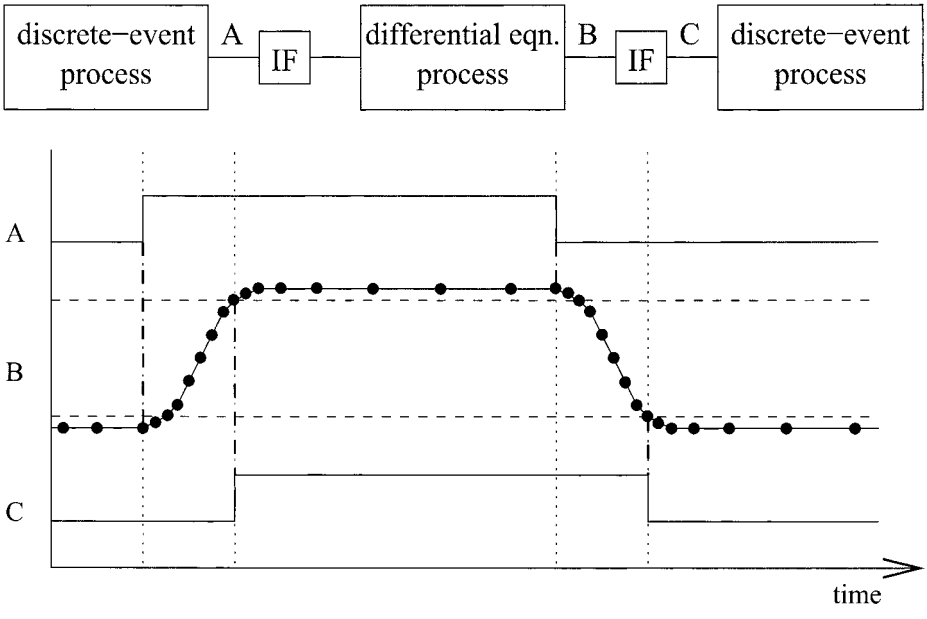

FIG. 4. (a) Example mixed-mode simulation model, (b) signals from example mixed-mode simulation model.

simulation model is defined as interacting processes. Discrete-event processes de fine the behavior of the discrete-event model whereas differential equation processes define the behavior of the differential equation model. The interface functions are entrusted with the task of the communication between these processes; the different notions of time remain the responsibility of the simulator. Resolving the different notions of time is critical for correct simulation. Figure 5 illustrates the temporal behavior of a discrete-event process and a differential equation process. As seen in Fig. 5, execution of a discrete-event process is instantaneous. Time is not advanced during execution. On the other hand, a differential equation process's execution may advance the simulation time during execution. This means that the execution of a differential equation process may result in the execution influencing itself. This continuous time increment is why a differential equation process is referred to as a *self-advancing* process. In the remainder of this section, differential equation processes are denoted as self-advancing processes.

A mixed-mode simulator must coordinate between the two different simulation processes. In current implementations of integrated mixed-mode simulators, this coordination is achieved by partitioning the self-advancing process into a large set of discrete-event processes [1]. However, this results in high communication costs, which should be avoided when targeting execution to a parallel machine. As this study is restricted to parallel mixed-mode simulation on distributed networks of workstations, intelligent partitioning is essential for reducing the communication costs. Current implementations accomplish this partitioning by combining a set of processes belonging to the same self-advancing process and executing this partition on one processor. The same goal is achieved if the self-advancing process is considered as a single heavyweight process with respect to network synchronization.

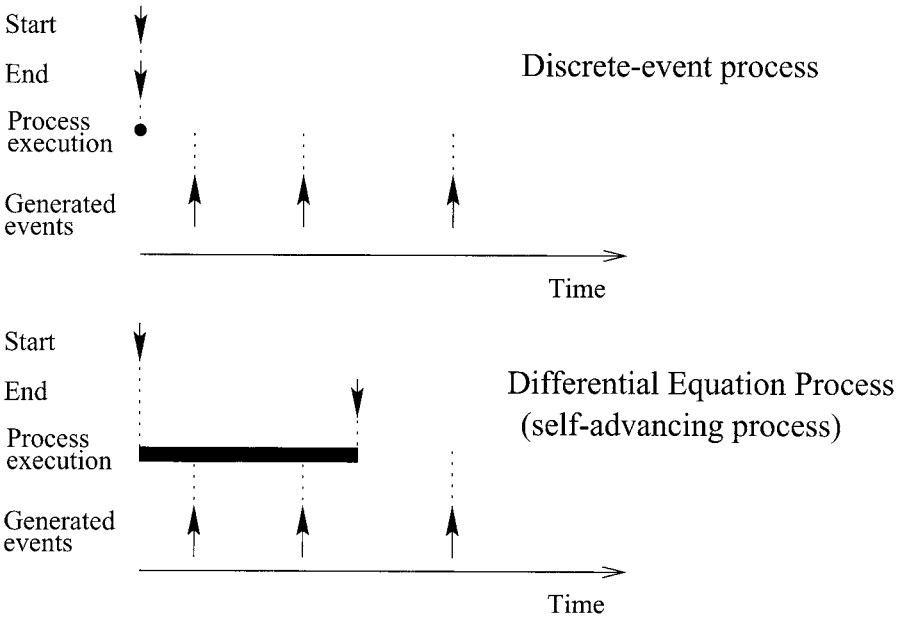

FIG. 5. Mixed-mode processes.

But this assumes that the simulator can synchronize the two different types of processes. In our work to to facilitate the synchronization of the self-advancing process with other discrete-event processes, a new synchronization scheme is proposed [19]. This novel scheme adopts a process based approach to synchronization and is called *process synchronization.*

Process synchronization of mixed-mode simulation processes involves the design of a synchronization interface that handles the interactions between self advancing processes and discrete-event processes. In the implementation of process synchronization reported in this document, the analog sub-assemblies are grouped into `n` analog processes (each called an *analog island*) for distribution and parallel execution on `n` processors. Process synchronization protocols then bridge the interfaces between each analog island and the remainder of the simulation. This has several advantages. Since traditional optimistic discrete-event simulation stores states after each activation of a process, memory requirements are reduced in the case of a single self advancing process. This is because the self-advancing process advances in its own time domain, such that state saving is only required at synchronization points. Since synchronization is limited to specific simulation intervals, the number of states saved will be much lower than the number of states saved in the traditional optimistic discrete-event simulation. It is important to note that state and event histories are the major memory intensive components of the discrete-event simulation and by saving state only at synchronization points, a considerable amount of memory consumption is reduced. This is essentially a trade off between process granularity and memory requirements [17]. Communication is reduced to interface function communication in the case of the self-advancing processes. As self-advancing processes have high internal communication demands, this communication is kept local. Tahawy *et al.* [47] were the first to investigate synchronization of

processes for mixed-mode simulation. However, the process synchronization scheme introduced by Tahawy only considered sequential event-driven simulation kernels. Due to the requirement of parallel simulation methods of today's large scale simulation models, the utility of Tahawy's scheme is limited. A more detailed description of the process synchronization protocol is available in the literature [19].

With the growing trend of hardware designs that contain significant analog and digital sections, comprehensive design environments that seamlessly integrate analog and digital circuitry are necessary. Toward this end, the VHDL language (for which there are several design tools) was extended to support (in addition to digital) analog and mixed-signal simulation. These extensions to VHDL have resulted in a new language called VHDL-AMS [25]. In addition to the electrical-domain, VHDL-AMS also supports mixed-domain modeling. There are several ongoing endeavors in the academic and industrial worlds to develop design environments to create and simulate models written in VHDL-AMS. Simulation Environment for VHDL-AMS (SEAMS) [18], is one such mixed-signal simulator that is the product of our research. SEAMS [18] consists of several modules that are integrated into a single simulation system. Figure 6 shows the architecture of SEAMS. The input to the system is a VHDL-AMS model that is to be simulated. The model is parsed and analyzed (the syntax and the semantics of the system are verified). The *scram* analyzer (augmented with support for VHDL-AMS) converts the VHDL-AMS code to an intermediate format from which C^{++} code is automatically generated. Once again the intermediate format and the generated code have been augmented to support the analog and mixed-signal language extensions to VHDL. The code that is generated is compliant to two simulation kernels. To correctly handle the aspects of digital logic simulation, the TyVIS kernel API (augmented to support VHDL-AMS types) is used. To correctly handle the aspects of analog logic simulation, the *anaKernel*'s API is used. The generated C^{++} program is then compiled and linked with the different kernels in the system, namely the TyVIS VHDL (and VHDL-AMS) Kernel [51], the *anaKernel* analog simulation kernel, and the WARPED optimistic discrete-event simulation kernel [37]. In experiments carried out with simple mixed-signal circuits such as clock generators

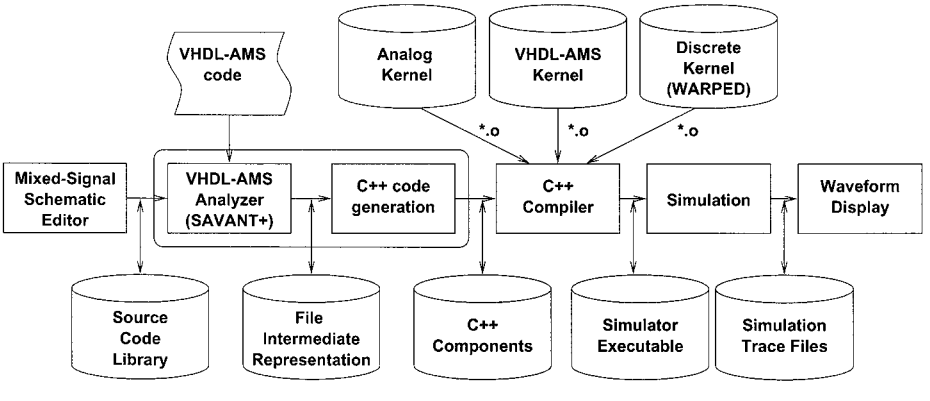

FIG. 6. Architecture of SEAMS.

consisting of an analog resistor-capacitor (RC) circuit connected to two digital NOT logic gates, speedups of more than 60% were obtained using the process synchronization approach [19].

7. CONCLUSIONS

Designing and implementing a simulation environment that is capable of simulating complete mixed-technology (VHDL and VHDL-AMS) models is a daunting task. If this is not challenging enough, the requirement of executing this simulation environment on a distributed network of workstations efficiently is sure to push the difficulty and complexity level of the task up a few notches. In this paper, we described in detail the design and implementation of such an environment for the simulation of mixed-technology systems. Specifically, the design of the SAVANT/TyVIS/WARPED simulation environment was described and some of the issues faced during the implementation were discussed. In addition, optimizations to the simulation environment were also discussed. This demonstrated the success and utility of applying PDES techniques to the simulation of electronic CAD systems. Finally, we showed how the environment was extended to handle the simulation of analog and digital VLSI circuit designs (in the form of VHDL-AMS model descriptions). To our knowledge, the SAVANT/TyVIS/WARPED and the SEAMS environments are the only ones of its kind that allow the parallel discrete-event simulation of mixed-technology systems.

ACKNOWLEDGMENTS

This research was supported in part by Wright Laboratory under USAF Contract F33615-95-C-1638 and by DARPA under Contracts J-FBI-93-116 and DABT63-96-C-0055. This research has been conducted with the participation of many investigators. We acknowledge and thank them for all their efforts. While not a complete list, the following individuals have made notable direct and/or indirect contributions to this effort (in alphabetical order): Jeff Carter, Harold W. Carter, Chetput L. Chandrashekar, Praveen Chawla, Peter Rey, John Hines, Herb Hirsch, Ramesh S. Mayiladuthurai, Timothy J. McBrayer, Kathiresan Nellayappan, Greg Peterson, Phani Putrevu, Umesh Kumar V. Rajasekaran, Al Scarpelli, Vasudevan Shanmugasundaram, Mike Shellhause, Swaminathan Subramanian, Narayanan V. Thondugulam, and John Willis.

REFERENCES

1. Eduardo L. Acuna, James P. Dervenis, Andrew J. Pagones, Fred L. Yang, and Resve A. Saleh, Simulation techniques for mixed analog/digital circuits. *IEEE J. Solid-State Circuits* **25**(2) (April 1990), 353–363.
2. P. Agrawal, Concurrency and communication in hardware simulators, *IEEE Trans. Comput.-Aided Design* (October 1986).
3. P. Agrawal and W. J. Dally, A hardware logic simulation system, *IEEE Trans. Comput.-Aided Design Integrated Circuits Systems* **9**(1) (January 1990), 19–20.
4. Brian A. A. Antao and Arthur J. Brodersen, Behavioral simulation for analog system design verification, *IEEE Trans. VLSI Systems* **3**(3) (September 1995), 417–429.
5. M. L. Bailey, How circuit size affects parallelism, *IEEE Trans. Comput.-Aided Design* **11**(2) (February 1992), 208–215.

6. M. L. Bailey, J. V. Briner, Jr., and R. D. Chamberlain, Parallel logic simulation of VLSI systems, *ACM Comput. Surv.* **26**(3) (September 1994), 255–294.
7. H. Bauer and C. Sporrer, Distributed logic simulation and an approach to asynchronous GVT-calculation, *in* "6th Workshop on Parallel and Distributed Simulation," pp. 205–208, Society for Computer Simulation, January 1992.
8. H. Bauer and C. Sporrer, Reducing rollback overhead in Time-Warp based distributed simulation with optimized incremental state saving, *in* "Proc. of the 26th Annual Simulation Symposium," pp. 12–20, Society for Computer Simulation, April 1993.
9. H. Bauer, C. Sporrer, and T. H. Krodel, On distributed logic simulation using Time Warp, *in* "VLSI 91" (A. Halaas and P. B. Denyer, Ed.), pp. 127–136, Edinburgh, Scotland, August 1991 [IFIP TC 10/WG 10.5].
10. J. V. Briner, Jr., "Parallel Mixed-Level Simulation of Digital Circuits Using Virtual Time," Ph.D. thesis, Duke University, Durham, North Carolina, 1990.
11. CAD Benchmarking Lab, NCSU, "ISCAS '89 Benchmark Information," available at http://www.cbl.ncsu.edu/www/CBL_Docs/iscas89.html.
12. Christopher D. Carothers, Kalyan S. Perumalla, and Richard M. Fujimoto, Efficient optimistic parallel simulations using reverse computations, *in* "Proceedings of the Thirteenth Workshop on Parallel and Distributed Simulation, PADS '99," pp. 126–135, May 1999.
13. François E. Cellier, "Continuous System Modeling," Springer-Verlag, Berlin/New York, 1991.
14. K. M. Chandy and J. Misra, Asynchronous distributed simulation via a sequence of parallel computations, *Comm. ACM* **24**(11) (April 1981), 198–206.
15. M. Chetlur, N. Abu-Ghazaleh, R. Radhakrishnan, and P. A. Wilsey, Optimizing communication in Time-Warp simulators, *in* "12th Workshop on Parallel and Distributed Simulation," pp. 64–71, Society for Computer Simulation, May 1998.
16. J. Cloutier, E. Cerny, and F. Guertin, Model partitioning and the performance of distributed time warp simulation of logic circuits, *in* "Simulation Practice and Theory," pp. 83–99, 1997.
17. J. Fleischmann and P. A. Wilsey, Comparative analysis of periodic state saving techniques in Time Warp simulators, *in* "Proc. of the 9th Workshop on Parallel and Distributed Simulation (PADS 95)," pp. 50–58, June 1995.
18. P. Frey, K. Nellayappan, V. Shanmugasundaram, R. S. Mayiladuthurai, C. L. Chandrashekar, and H. W. Carter, SEAMS: Simulation Environment for VHDL-AMS, *in* "1998 Winter Simulation Conference (WSC '98)," December 1998.
19. P. Frey, R. Radhakrishnan, H. W. Carter, and P. A. Wilsey, Parallel mixed-technology simulation, *in* "Proceedings of the Fourteenth Workshop on Parallel and Distributed Simulation (PADS '00)," pp. 7–14, May 2000.
20. R. Fujimoto, Parallel discrete event simulation, *Comm. ACM* **33**(10) (October 1990), 30–53.
21. R. M. Fujimoto, The virtual time machine, *in* "International Symposium on Parallel Algorithms and Architectures," pp. 199–208, June 1989.
22. A. Gafni, Rollback mechanisms for optimistic distributed simulation systems, *in* "Distributed Simulation," pp. 46–53, Society for Computer Simulation, January 1988.
23. W. Gropp, E. Lusk, and A. Skjellum, "Using MPI: Portable Parallel Programming with the Message-Passing Interface," MIT Press, Cambridge, MA, 1994.
24. "IEEE Standard VHDL Language Reference Manual," IEEE, New York, NY, 1993.
25. IEEE Computer Society, "IEEE Draft Standard VHDL-AMS Language Reference Manual," IEEE, 1997.
26. D. Jefferson, Virtual time, *ACM Trans. Programming Languages Systems* **7**(3) (July 1985), 405–425.
27. George Karypis and Vipin Kumar, "Multilevel *k*-Way Partitioning Scheme for Irregular Graphs," Technical Report TR 95–055, University of Minnesota, Computer Science Department Minneapolis, MN 55414, August 1995.

28. S. A. Kravitz and B. D. Ackland, Static vs. dynamic partitioning of circuits for a MOS timing simulator on a message-based multiprocessor, *in* "Proceedings of the SCS Multi-conference on Distributed Simulation," 1988.
29. L. Lamport, Time, clocks, and the ordering of events in a distributed system, *Comm. ACM* **21**(7) (July 1978), 558–565.
30. B. D. Lubachevsky *et al.*, Rollback sometimes works... if filtered, *in* "Winter Simulation Conference," pp. 630–639, Society for Computer Simulation, December 1989.
31. J. Misra, Distributed discrete-event simulation, *Comput. Surv.* **18**(1) (March 1986), 39–65.
32. Isreal Mitrani, "Simulation Techniques for Discrete Event Systems," Cambridge Univ. Press, New York, 1982.
33. A. Palaniswamy and P. A. Wilsey, An analytical comparison of periodic checkpointing and incremental at saving, *in* "Proc. of the 7th Workshop on Parallel and Distributed Simulation (PADS '93)," pp. 127–134, Society for Computer Simulation, July 1993.
34. Terence J. Parr, "Language Translation Using PCCTS and C^{++}," Automata Publishing Company, January 1997.
35. Kalyan S. Perumalla and Richard M. Fujimoto, Efficient large-scale process-oriented parallel simulation, *in* "Proceedings of the 1998 Winter Simulation Conference (WSC '98)," pp. 459–466, 1998.
36. Kalyan S. Perumalla, Richard M. Fujimoto, and Andrew T. Ogielski, "The TeD Language Manual," 1998, available at www.cc.gatech.edu/computing/pads/ted.html.
37. R. Radhakrishnan, D. E. Martin, M. Chetlur, D. M. Rao, and P. A. Wilsey, An object-oriented Time Warp simulation kernel, *in* "Proceedings of the International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE '98)" (D. Caromel, R. R. Oldehoeft, and M. Tholburn, Eds.), Lecture Notes in Computer Science, Vol. 1505, pp. 13–23, Springer-Verlag, Berlin/New York, 1998.
38. R. Radhakrishnan, T. J. McBrayer, K. Subramani, M. Chetlur, V. Balakrishnan, and P. A. Wilsey, A comparative analysis of various Time Warp algorithms implemented in the WARPED simulation kernel, *in* "Proceedings of the 29th Annual Simulation Symposium," pp. 107–116, 1996.
39. U. K. V. Rajasekaran, M. Chetlur, G. D. Sharma, R. Radhakrishnan, and P. A. Wilsey, Adressing communication latency issues on clusters for fine grained asynchronous applications—A case study, *in* "International Workshop on Personal Computer Based Network of Workstations, PC-NOW '99," April 1999.
40. Resve Saleh, Shyh-Jye Jou, and A. Richard Newton, "Mixed-Mode Simulation and Analog Multilevel Simulation," Kluwer Academic, Dordrecht/Norwell, MA, 1994.
41. S. P. Smith, B. Underwood, and M. R. Mercer, An analysis of several approaches to circuit partitioning for parallel logic simulation, *in* "Proceedings of the 1987 International Conference on Computer Design," pp. 664–667, IEEE, New York, 1987.
42. L. P. Soulé, "Parallel Logic Simulation: An Evaluation of Centralized-Time and Distributed-Time Algorithms," Technical Report CSL-TR-92–527, Computer Systems Laboratory, Stanford University, Stanford, CA, June 1992.
43. L. P. Soulé and T. Blank, Statistics for parallelism and abstraction level in digital simulation, *in* "Proceedings of the 24th Design Automation Conference," pp. 588–591, ACM/IEEE, New York, 1987.
44. L. P. Soulé and A. Gupta, An evaluation of the Chandy–Misra–Bryant algorithm for digital logic simulation, *ACM Trans. Modeling Comput. Simulation* **1**(4) (October 1991), 308–347.
45. Christian Sporrer and Herbert Bauer, Corolla partitioning for distributed logic simulation of VLSI-circuits, *in* "Proceedings of the 7th Workshop on Parallel and Distributed Simulation," pp. 85–92, May 1993.
46. Swaminathan Subramanian, Dhananjai M. Rao, and Philip A. Wilsey, Study of a multilevel approach to partitioning for parallel logic simulation, *in* "International Parallel and Distributed Processing Symposium, (IPDPS '00)," IEEE Comput. Society Press, May 2000, forthcoming.

47. H. El Tahawy, D. Rodriguez, S. Garcia-Sabiro and J.-J. Mayol, Vhd$_e$ldo: A new mixed mode simulation, *in* "EURO DAC 1993," September 1993.

48. K. Umamageswaran, K. Subramani, P. A. Wilsey, and P. Alexander, Formal verification and empirical analysis of rollback relaxation, *J. Systems Architecture* **44**, 473–495 (1998).

49. D. West, "Optimizing Time Warp: Lazy Rollback and Lazy Re-evaluation," Master's thesis, University of Calgary, Calgary, Alberta, 1988.

50. J. C. Willis, P. A. Wilsey, G. D. Peterson, J. Hines, A. Zamfriescu, D. E. Martin, and R. N. Newshutz, Advanced intermediate representation with extensibility (AIRE), *in* "VHDL Users' Group Fall 1996 Conference," pp. 33–40, October 1996.

51. P. A. Wilsey, D. E. Martin, and K. Subramani, SAVANT/TyVIS/WARPED: Components for the analysis and simulation of VHDL, *in* "VHDL Users' Group Spring 1998 Conference," pp. 195–201, March 1998.

52. B. P. Zeigler, "Theory of Modelling and Simulation," Wiley, New York, 1976.