ACADEMIC PRESS

J. Parallel Distrib. Comput. 62 (2002) 1670–1693

Journal of Parallel and Distributed Computing

www.academicpress.com

An ultra-large-scale simulation framework ☆

Dhananjai M. Rao and Philip A. Wilsey*

Experimental Computing Laboratory, Department of Electrical & Computer Engineering, Computer Science University of Cincinnati, Cincinnati, OH 45221-0030, USA

Received 20 July 2000; received in revised form 13 February 2001; accepted 3 March 2001

Abstract

Many modern systems involve complex interactions between a large number of diverse entities that constitute these systems. Unfortunately, these large, complex systems frequently defy analysis by conventional analytical methods and their study is generally performed using simulation models. Further aggravating the situation, detailed simulations of large systems will frequently require days, weeks, or even months of computer time and lead to scaled down studies. These scaled down studies may be achieved by the creation of smaller, representative, models and/or by analysis with short duration simulation exercises. Unfortunately, scaled down simulation studies will frequently fail to exhibit behaviors of the full-scale system under study. Consequently, better simulation infrastructure is needed to support the analysis of ultra-large (models containing over 1 million components)-scale models.

Simulation support for ultra-large-scale simulation models must be achieved using low-cost commodity computer systems. The expense of custom or high-end parallel systems prevent their widespread use. Consequently, we have developed an Ultra-large-Scale Simulation Framework (USSF). This paper presents the issues involved in the design and development of USSF. Parallel simulation techniques are used to enable optimal time versus resource tradeoffs in USSF. The techniques employed in the framework to reduce and regulate the memory requirements of the simulations are described. The API needed for model development is illustrated. The results obtained from the experiments conducted using various system models with two parallel simulation kernels (comparing a conventional approach with USSF) are also presented.

Keywords: Large-scale modeling; Parallel and distributed simulation; Time warp simulation; Unsynchronized simulation

1. Introduction

Modern systems such as microprocessors and communication networks have steadily grown in size and sophistication to meet the ever increasing needs and demands. For example, today's microprocessors are built using a few million

☆ Support for this work was provided in part by the Defense Advanced Research Projects Agency under Contract DABT63-96-C-0055.

*Corresponding author.

E-mail addresses: dmadhava@ececs.uc.edu (D.M. Rao), philip.wilsey@ieee.org (P.A. Wilsey).

0743-7315/02/$ - see front matter
PII: S0743-7315(02)00003-5

transistors [14] and the Internet, a global data network, now connects more than 16 million nodes [15]. These systems involve complex interactions between a few thousand to several million entities. The study and analysis of these systems is necessary in order to effectively design, manufacture, and maintain them [15,25]. Unfortunately, analytical methods of analysis are insufficient to study these systems and experimental techniques such as computer-based simulations are usually employed instead [21,25]. Furthermore, parallel simulation techniques are employed to enable simulation of large systems in acceptable time frames [15,21,25]. Simulation enables explorations of complicated scenarios that would be either difficult or impossible to analyze [15]. Due to its effectiveness, simulation has gained considerable importance and is widely used today.

Validity of the models plays central role in analyzing systems using simulation [23]. The models should reflect the size and complexity of the system in order to ensure that crucial scalability issues do not dominate during validation of simulation results. Many techniques, algorithms, and protocols that work acceptably for small models consisting of tens or hundreds of entities may become impractical when the size of the system grows [15]. Events that are rare or that do not even occur in small toy models may be common in the actual system under study. Detailed simulation of the complete system is necessary to study large-scale characteristics, long-term phenomena, and to analyze the system as a whole. Paxson et al., provide an excellent context from the networking domain to highlight this issue. They write, ''Indeed, the HTTP protocol used by the World Wide Web is a perfect example of a success disaster. Had its designers envisioned it in use by virtually the entire Internet—and had they explored the corresponding consequences with experiments, analysis or simulation—they would have significantly altered its design, which in turn would have led to a more smoothly operating Internet today'' [15]. Since today's systems involve a large number of entities ranging in the order of a few millions, modeling and simulating such ultra-large systems is necessary.

Simulation of large systems is complicated due to their sheer size. The memory and computational resources needed to simulate such large systems in acceptable time frames are often beyond the limits of a single stand alone workstation [18]. Developing large and complex models while paying special care to optimally utilize system resources (in particular, memory) is a tedious task demanding considerable expertise from the modeler. Parallel simulation techniques need to be efficiently exploited to meet the computational requirements. However, investing in large and expensive hardware components for a ''one time'' analysis of the system models is seldom economically viable. Hence, simulating large systems using modest hardware resources is an attractive and often the only alternative.

This paper presents the design and evaluation of an Ultra-large-Scale Simulation Framework (USSF) that was developed to enable and ease effective simulation of large systems. In particular, USSF was motivated by the need to support analyzing systems involving millions of entities using only modest computational resources. The framework utilizes parallel simulation techniques to harness the resources of conventional workstations to provide optimal time versus resource tradeoffs. Various software techniques have been employed to reduce and regulate the memory requirements of the simulations. USSF provides a flexible and robust object-oriented API for model development. The API also insulates the model developer from the intricacies of enabling large simulations.

The remainder of this paper is organized as follows. A brief description about the parallel simulation kernels that are used as the underlying synchronization kernels of USSF are presented in Section 2. In Section 3, brief descriptions about some of the earlier research activities related to large-scale simulations are presented. Section 4

outlines the software techniques used to alleviate the memory bottlenecks faced while enabling large-scale simulations. A detailed description of the USSF along with the API is presented in Section 5. The results obtained from the experiments using the framework with different models and parallel simulation kernels are presented in Section 6. Section 7 provides some concluding remarks with pointers to future work.

2. Background

The parallel simulation capability of USSF is enabled by developing the framework around a given parallel simulation engine. That is, the framework runs as an application on an underlying parallel kernel and utilizes its services. Object-oriented (OO) techniques have been employed to isolate the various modules of USSF from the underlying simulation kernel. This design was adopted in order to obtain a desired level of "separation of concerns" so that the design of the framework can focus on enabling large-scale simulations. The architecture of the framework can be viewed as extending the capabilities of the underlying parallel simulation engine. The design also enables USSF to be easily deployed on different simulation kernels. In this study, USSF was deployed on two different parallel simulation kernels; namely WARPED: an optimistic parallel discrete event simulation (PDES) kernel based on Time Warp [17]; and NOTIME: an unsynchronized PDES kernel [19,24]. The following subsections provide a brief description about these two simulation kernels.

2.1. WARPED

WARPED is an optimistic PDES kernel that uses the Time Warp [7] paradigm for distributed synchronization. A time warp synchronized simulation is organized as a set of asynchronous logical processes (LPs) that represent the different physical processes being modeled. The LPs share information by exchanging *virtual time* stamped event messages. Virtual time [9] is used to model the passage of time and defines a total order on the events in the system. Each LP processes its events by incrementing a local virtual time (LVT), changing its state, and generating new events. Although each LP processes local events in their correct time-stamp order, events are not globally ordered. Causality violations are detected when an event with time-stamps lower than the current LVT (a *straggler*) is received. On receiving a straggler event a rollback mechanism is invoked to recover from the causality error. The rollback process recovers the LP's state prior to the causal violation, canceling the erroneous output events generated, and reprocessing the events in their correct causal order. Each LP maintains a queue of state transitions along with lists of input and output events corresponding to each state to enable the recovery process. A periodic garbage collection technique based on global virtual time (GVT) is used to prune the queues by discarding history items that are no longer needed. The distributed simulation is deemed to have terminated when all the events in the system have been processed in their correct causal order.

The WARPED kernel presents an interface to build LPs. The kernel provides an application program interface (API) to build different LPs with unique definitions of state. The basic functionality for sending and receiving events between LPs using a message passing system is supported by the kernel. In WARPED, LPs are placed into groups called "clusters". LPs on the same cluster communicate with each other without the intervention of the message passing system, which is faster than communication through the message passing system. Although LPs are grouped

together into clusters they are not coerced into synchronizing with each other. Control is exchanged between the application and the simulation kernel through cooperative use of function calls. Further details on the API and working of WARPED is available in the literature [10,11,17].

2.2. NOTIME

NOTIME is an unsynchronized PDES kernel [19,24]. Unsynchronized simulations have been successfully employed for simulation of stochastic models such as queueing models and communication networks. NOTIME simulations provide considerable improvements in performance, when compared to WARPED simulations, with negligible loss in accuracy of the simulation results. A brief description of NOTIME is presented in the following paragraph and further discussions on unsynchronized simulations are available in the literature [19,24].

The NOTIME PDES kernel provides necessary support to develop applications modeled as discrete event simulations. NOTIME mirrors the API utilized by WARPED. The design enables models developed for WARPED to be run using NOTIME without changes to the application. Similar to WARPED, the LPs are grouped into clusters. Processor level parallelism occurs at the cluster level and each cluster is responsible for communication management and scheduling the LPs contained by the cluster. NOTIME utilizes the message passing interface (MPI) libraries for communicating between the parallel clusters. Communication between LPs on the same cluster occur without intervention of the communication layer. Since the parallelism occurs at the cluster level, simulation objects that execute relatively independent of each other can be placed on different clusters to maximize parallelism. Conversely, simulation objects that frequently communicate with each other should be placed on the same cluster to exploit the fast intra-cluster communication. Each cluster uses a single input queue that contains the events for all the LPs associated with it in order to optimize scheduling and intra-cluster communication. NOTIME uses a first-in–first-out (FIFO) scheduling scheme. The parallel simulation terminates when all the events in the system have been processed. The kernel uses a circulating token scheme for termination detection. Additional details on NOTIME are available elsewhere [19,24].

3. Related research

This section presents some of the other techniques that have been employed to enable large-scale simulations. Simulation of large-scale models has received considerable attention in the past. Various combination of software and hardware techniques have been used to improve the capacity and efficiency of simulators. Huag et al. present a novel technique to selectively abstract details of the network models and to enhance performance of large simulations [6]. Their technique involves modification of the network models in order to achieve abstraction [6]. Premore and Nicol present issues involved in development of parallel models in order to improve performance [16]. In their work, they convert source codes developed for *ns*, a sequential simulator into equivalent descriptions in telecommunications description language (TeD) to enable parallel simulation [16]. Coupled with meta languages (such as TeD) [16], parallel network libraries and techniques to transparently parallelize sequential simulations have been employed [22]. Riley et al. present a federated approach to enable parallel simulation in order to improve the capacity for simulating large models. In their technique, existing sequential

simulators are extended to enable parallel simulation using conservative synchronization techniques [22].

Relaxation and even elimination of synchronization, a large overhead in parallel simulations, has been explored [19,26]. The relaxation techniques attempt to improve performance at the cost of loss in accuracy of the simulation results [19,24]. Martini et al. [12] propose a novel synchronization protocol called "tolerant synchronization" in which a conservative synchronization protocol is allowed to optimistically process events that are within a tolerance level. As the protocol optimistically processes events, causality violations may occur but these violations are ignored and no recovery process is initiated to rectify the errors. Fall exploits a combination of simulation and emulation in order to study models with large real world networks [4]. This method involves real time processing overheads and necessitates detailed model development. Carothers et al. present a novel technique called "Reverse Computation" that eliminates the need for state saving in Time Warp simulations and reduces the memory requirements of the simulations [3]. However, not all computations can be reversed and this technique necessitates development of additional simulation code to undo computations.

On the other hand, USSF employs a different approach to enable simulation of large systems using resource constrained platforms. The framework exploits the presence of replicated modeling constructs and the reuse of component (or LP) descriptions to reduce the actual size of the simulations. Replicated structures are identified through static analysis of the model. The framework utilizes a single copy of each unique component (or LP) to mirror its different replicated instances. This approach reduces the overall size of the simulation which in turn reduces the resource consumption of the simulation. USSF also employs a number of techniques to reduce and regulate (i.e., improve efficient utilization of main memory) the overall memory requirements of the simulation in order to improve performance. Furthermore, the techniques employed in the framework are independent of the underlying synchronization mechanism and can be applied to any discrete event simulator. Use of the framework does involves only minor changes to the application modules—development of applications is straightforward and it does not require any change in the modeling methodology. In other words, existing applications can be easily ported to exploit the features of the USSF. A more detailed description of USSF is available in the following sections.

4. Approach

The initial exploratory simulation studies of large systems were conducted using WARPED [18,20]. The studies indicated that the memory requirements of such large simulations posed the primary bottleneck in enabling large-scale simulations. The memory requirements of a parallel simulation can be classified into two main categories: namely (i) the *static* memory requirements that do not change over the lifetime of the simulation; and (ii) the *dynamic* memory requirements that continuously change during simulation. The static memory requirements of the simulation arise due to the LP descriptions (application code), the kernel code, and the various data structures that need to be maintained during the lifetime of the simulation. Many of the kernel data structures need to be duplicated at each of the parallel clusters to enable parallel simulation [17]. Hence, irrespective of the number of workstations employed for parallel simulations, the static memory requirements of the kernel, for a given number of LPs, remains almost constant.

The discrete events and the states of the various LPs contribute to the dynamic memory requirements of the simulation. The dynamic memory requirements are governed by the state sizes of the application modules and their event generation characteristics. The static and dynamic memory requirements of the simulation must be reduced in order to enable large-scale simulations using modest hardware resources. Reducing the memory consumed by the parallel simulation kernel, by modifying its data structures, is complex due to the intricate mechanics of optimistic parallel simulations [17,21]. Modification of the data structures used by the kernel would affect its performance and increase simulation time [21,20]. Hence, techniques to reduce the static memory requirements and regulate the dynamic memory usage of the application were pursued.

The initial hurdle in enabling large-scale simulations was the static size of the application models and their data structures. The application development languages (described in Section 6) provided hierarchical constructs that ease specification of large systems. The hierarchical models were statically elaborated (i.e., at compile time) or "flattened" to a single hierarchical level prior to generating the simulatable code (as illustrated in Section 6). The static elaboration (or compile time elaboration) technique did not scale well in terms of the size of the generated code (the volume of generated code was too large), compilation time (the time taken to compile the generated code was unacceptably long), and static size of the resulting executable. Therefore, a runtime elaboration technique was proposed, in order to reduce the static size of the application modules and in turn its static memory requirements. A *runtime elaboration library* (REL) was developed as a part of USSF to ease runtime elaboration. The runtime elaboration technique and the REL are presented in Section 5.1. A comparative analysis between static elaboration and runtime elaboration is also presented in Section 5.1.

During our exploratory studies of large-scale simulations, it was observed that many of the LPs share the same descriptions. The LP descriptions (the code to model the functionality of the LPs) were the same but the data and states were different. The model descriptions could be reused by decoupling them from their data and state. In other words, the same description could be associated with different data and states in order to emulate the various instances of a particular LP. Using this approach a few thousand instances of a given LP could be aggregated into one. This approach reduces the total number of LPs and in turn would reduce the size of the internal data structures of the simulation kernel. Decoupling the data and states also provided a convenient design for swapping data and states in and out of the main memory based on demand, freeing up the memory to contain the discrete events which strongly influence the performance of the simulations [17,20,21]. These techniques regulate the dynamic memory consumption of the simulations. The USSF utilizes these techniques to enable effective simulation of large system using limited computational resources. The API of USSF insulates the model developer form the intricacies of enabling ultra-large simulations. A detailed description on the design and implementation of USSF is presented in the following section.

5. Ultra-large-scale simulation framework

The USSF was developed to ease simulation of large systems. An overview of the framework along with the applications used in this study is shown in Fig. 1. As illustrated in the figure the *networks simulation framework* (NSF), the *performance and scalability analysis framework* (PSAF), and the queueing models generate USSF API compliant code from the corresponding high-level model specifications. As

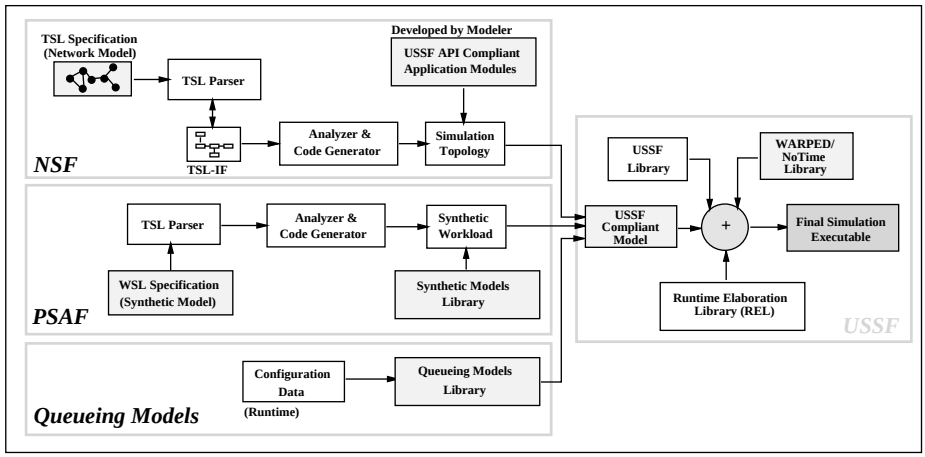

Fig. 1. System overview.

mentioned earlier, the compile time elaboration technique used in NSF and PSAF did not scale well when employed for large models involving millions of LPs. Hence, the elaboration technique was modified to employ the runtime elaboration technique (presented in Section 5.1). The necessary *static* analysis to identify and collate the various LPs that share a common object description (as required by USSF API), was also coupled along with code-generation. LPs that share the same description are identified using each object's definition. The object definitions are a part of the high-level model specifications and are extracted from the frontend modeling language supported by the application frameworks [1,21]. The collated information is embedded along with the generated runtime elaboration code and is utilized by the REL modules. The current implementation of USSF in concordance with WARPED and NOTIME is in C++. Accordingly, the application frameworks also generate code in C++. As shown in Fig. 1, the generated code is compiled with the appropriate libraries to obtain the final simulation executable. A brief description about the applications is available in Section 6 and a detailed description about the various components of USSF are presented in the following subsections.

5.1. Runtime elaboration library

Hierarchical language constructs provide convenient techniques to specific large systems by reusing the specification for smaller subsystems [20,21]. The frontend modeling languages of the application frameworks provide hierarchical constructs to ease specification of large models. However, the hierarchical constructs have to be elaborated or "flattened" prior to simulation. Elaboration is the process in which each hierarchical level is broken down to its constituting components. The basic steps involved in elaborating a hierarchical specification are shown in Fig. 2. As illustrated in the figure, the elaborator starts with an user-specified hierarchy and recursively traverses the various components in the model and creates new instances of the sub-hierarchies and the objects. Elaboration of sub-hierarchies is done before they are imploded into the enclosing hierarchy. Imploding hierarchies involves inclusion of all necessary object definitions, object instantiations, and corresponding data structures.

Elaboration may be done *statically* or at *runtime*. Static elaboration occurs prior to code generation while runtime elaboration occurs prior to simulation, when the

step 1: Initialize elaborator

1. Initialize new symbol table and IF
2. Search input IF and locate node corresponding to user specified top level hierarchy
3. Call elaboration (**step 2**) with new hierarchy

step 2: Elaboration subroutine (*parameter* `hierarchy`)

1. Process the list of modules specified in the `hierarchy`. If the module is an object instantiation perform **step 3**. If the module is a sub-hierarchy perform **step 4**.

step 3: Elaborate object instantiation

1. Create new instance of the object instantiation with name-mangled labels.
2. Create new object definition for the new instance with mangled labels and add to new symbol table, if necessary. Collate information about objects sharing the same object definition.
3. Add new object instantiation to the new topology and update necessary data structures.

step 4: Elaborate `sub-hierarchy`

1. Instantiate a temporary symbol table and IF
2. Recursively call elaboration with the `sub-hierarchy`
3. Implode new sub-hierarchy into current hierarchy

Fig. 2. Phases in elaborating a hierarchical design.

generated code is executed. Since static elaboration occurs prior to code generation, the volume of code generated for large models is considerably higher when compared to the code generated when using runtime elaboration. For example, for a network model consisting of a million nodes the number of lines of C++ code generated is a few million. The few million lines of code occupy large volumes of disk space and the compilation times are unacceptably long. As illustrated by the experiments presented in Section 6.4, the static elaboration technique does not scale well. In contrast, the code generated for runtime elaboration merely captures the hierarchical structure of the network model and passes the information to the REL. The volume of the generated C++ code is considerably less and hence compilation times are reduced. The static size of the executable for runtime elaboration is also smaller. However, runtime elaboration involves additional overheads prior to simulation. Runtime elaboration provides a tradeoff between the size of the generated code, the compilation time of the generated code, and overall simulation time.

The REL was developed to ease elaboration of large models. The modules of the REL interact with the various modules of USSF kernel during elaboration and construct the simulation at runtime. Fig. 3 illustrates the important classes that constitute the API of the REL. The `Elaboration` class, the `BasicElabContainer` class, and the `BasicModel` class form the core infrastructure of the REL. The `BasicElabContainer` class is the base class for all the elaboration classes that are used to instantiate the actual LPs that constitute the simulation. For each unique simulation module, the code-generator is responsible for generating a corresponding container class with the `BasicElabContainer` class as its parent. As shown in Fig. 3, the generated container class overloads the necessary pure virtual method in the `BasicElabContainer` class. The `Elaboration` class is the base class for each of the generated elaboration classes. For each level in a hierarchical design, an unique class is generated by the code-generator. The generated code contains the necessary calls to the various methods in the `Elaboration` class. On instantiating an elaboration class for a given hierarchy, pointers to the underlying sub-hierarchies and objects,

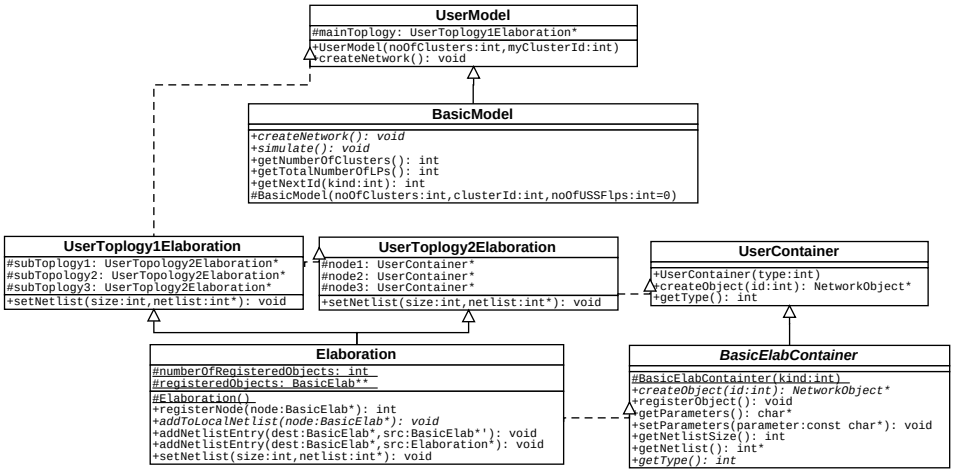

Fig. 3. UML diagram for runtime elaboration API.

code generated as member objects, are suitably instantiated. Runtime elaboration proceeds in a depth-first manner. The `BasicModel` class acts as the base class for generating the top level elaboration class. The generated code triggers runtime elaboration by instantiating the elaboration class corresponding to the top most hierarchy in the design. The elaboration data structures constructed in memory are deleted as the recursive decent unwinds in order to ensure minimal memory usage. The REL also includes support for elaborating models directly for WARPED and NOTIME without necessitating any changes to the generated code.

5.2. USSF kernel

The core functionality of regulating the memory requirements of the application modules are handled by the USSF kernel modules. The kernel modules present an interface similar to WARPED and NOTIME to the model developer. The USSF API is presented in the following subsection. The core of the USSF kernel is the USSF cluster. The USSF cluster represents the basic building block of USSF simulations. Each USSF cluster is assigned and addressed by an unique *id*. A USSF cluster performs two important functions. It not only acts as a LP to WARPED and NOTIME, it also acts as a cluster to the application programmer. As shown in Fig. 4, the USSF cluster is used to group a number of LPs that use the same description together. A single copy of an user process is associated with different data and states to emulate its various instances. The USSF cluster uses file-based caches to maintain the data and states of the various processes. The caching helps in regulating the demands on main memory. Separate data and state caches are maintained to satisfy concurrent accesses to data and state spaces and to reduce cache misses. On encountering rollbacks, the kernel flushes all the caches to maintain cache consistency and coherence. OO techniques have been used to decouple the various memory management routines from the core of the USSF kernel. This design not only provides a simple mechanism to substitute various memory management algorithms but also insulates the USSF cluster from their intricacies.

The USSF cluster is also responsible for scheduling the various application processes associated with it. The USSF cluster appropriately translates the calls made by the underlying simulation kernel into corresponding application process calls. It is also responsible for routing the various events generated by the application

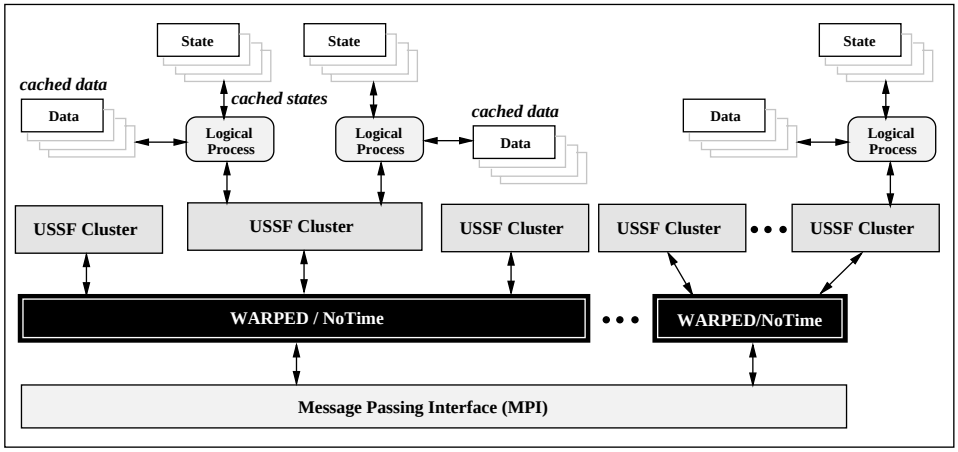

Fig. 4. Layout of an USSF simulation.

to the simulation kernel. The WARPED and NOTIME kernels support exchange of events between the USSF clusters. To enable exchange of events between the various user LPs, the USSF cluster translates the source and destination of the various events to and from USSF cluster *id*'s. In order for USSF kernel to perform these activities, a table containing the necessary information is maintained by the kernel modules. The table is indexed using the unique process *id*'s that need to be associated with each user LP. To reduce the number of entries in this table, a single entry is maintained for a group of LPs sharing a process description. The static analysis phase of the various application frameworks assigns contiguous *id*'s to processes constructed using the same simulation objects. This fact is exploited to efficiently construct and maintain the table. The USSF cluster also maintains a file-based state queue in order to recover from rollbacks [7] that could occur in a Time Warp simulation. An incremental state saving mechanism with a fixed (at compilation time) check-pointing interval is used for this purpose [5]. The states space of the USSF cluster contain the corresponding offsets of the checkpoint and state information in the state queue. The offsets are used to restore the states efficiently after a rollback. A simple garbage collection mechanism triggered by the garbage collection routines in WARPED is used to prune the state queues. Access to the various methods in the USSF kernel is provided via a set of application program interfaces, illustrated in Section 5.3. Further details on the design and implementation of the USSF kernel is available in the literature [20,21].

5.3. USSF application program interface (API)

The API presented by USSF closely mirrors the WARPED API [17]. This enables existing WARPED and NOTIME applications to exploit the features of USSF with few modifications to USSF. The API has been developed in C++ and the object oriented features of the language have been exploited to ensure it is simple and yet robust. The API plays a critical role in insulating the model developer from the intricacies involved with enabling ultra-large parallel simulations. The interface has been carefully designed to provide sufficient flexibility to the application developer and enable optimal system performance. Fig. 5 presents an overview of the important classes that constitute the API.

The USSF kernel presents an interface to the application developer for modeling a set of communicating LPs. The LPs are modeled as entities which send and receive

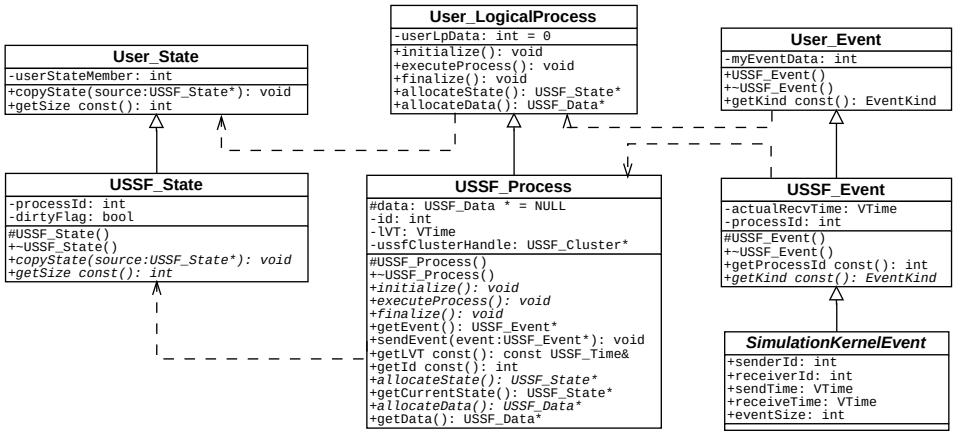

Fig. 5. Overview of USSF API.

events to and from each other, and act on these events by applying them to their internal state. The `USSF_Process` class forms the base class for all the LPs. The basic functionality that the `USSF_Process` class provides for modeling LPs are methods for sending and receiving events between the LPs and the ability to specify different types of LPs with unique definitions of state. The `USSF_State` and `USSF_Event` form the base classes for the states and events in the system, as shown in Fig. 5. The user is expected to override some of the kernel methods that are invoked at various times through out the simulation. Each method in this set has a specific function. The `initialize` method gets called on each LP before the simulation begins. This gives each LP a chance to perform any actions required for initialization. The method `finalize` is called after the simulation has ended. The method `executeProcess` of a LP is called by the USSF kernel whenever the LP has at least one event to process. The kernel calls `allocateState` and `allocateData` when it needs the LP to allocate a state or data on its behalf. The various interface classes along with the inheritance hierarchies for specifying application data, states, and events are shown in Fig. 5. Although it is the responsibility of the modeler to assign unique *id*'s to each LP, the static analysis modules in the USSF perform this functionality. Interfaces for constructing application data, states, and events are also specified. Control is exchanged between the application and the USSF kernel modules through cooperative use of function calls. A detailed flow of control in the system via the API calls is presented in the following subsection.

5.4. Flow of control in USSF

The flow of control in the USSF is illustrated in order to fully highlight the issues involved in the various aspects of its design. Fig. 6 shows the various interactions between USSF, the underlying parallel simulation kernel, and the application during simulation. The first phase of the simulation deals with setting up of the various USSF clusters and the processes contained in them. The runtime elaboration modules perform the task of elaborating the topology and instantiating the necessary LPs. As the processes are created and registered with the USSF cluster, the internal tables are updated. At the end of this phase, the various USSF clusters register themselves with the underlying simulation kernel. The `initialize` method of the various USSF cluster processes are invoked by the underlying simulation kernel. The USSF clusters then exchange time stamped events distributing their internal tables to

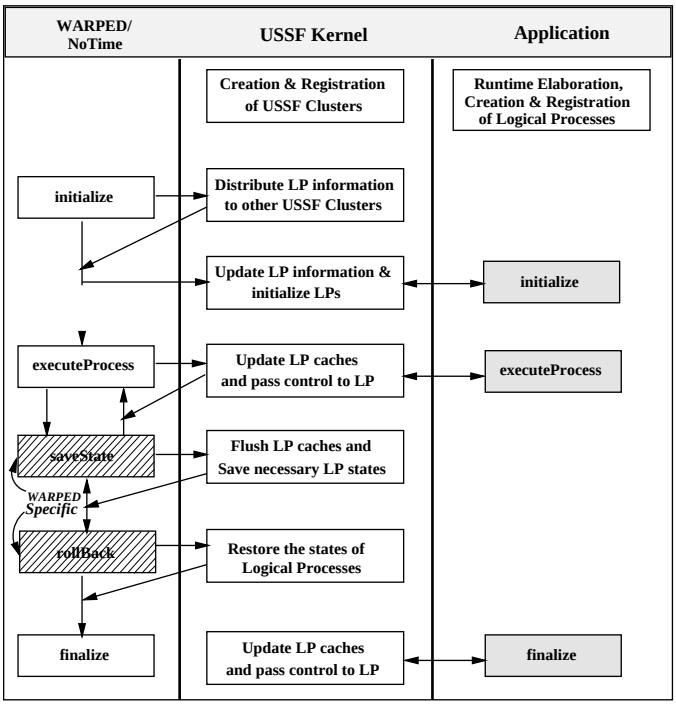

Fig. 6. Flow of control in USSF.

other clusters. It is important to note that the USSF kernel events have a time stamp that is lower than those of the applications. This is necessary to ensure that the underlying kernels schedule USSF kernel events before the application events are scheduled. Processing the kernel events is crucial in order to ensure that the internal data structures are updated before any of the application's processing begins. Once updating of the internal data structures is complete, the USSF clusters call the `initialize` methods of all the LPs associated with them. When the `executeProcess` method of the USSF cluster is invoked, it updates the data and state caches of the corresponding LP and in turn calls the LP's `executeProcess` method. The events generated by the application are appropriately translated to USSF Cluster *id*'s and dispatched using the underlying kernel's interfaces. The USSF cluster also saves the state of the various processes as when the state saving methods are triggered. The saved states are used to restore the states of the various LPs when a rollback occurs during simulation. Garbage collection is done when the routines are triggered by WARPED or NOTIME. Finally, when the `finalize` method of the USSF cluster is called, the USSF cluster calls the `finalize` method of the various LPs associated with it and clears all its data structures and the simulation terminates.

6. Experiments

The experiments conducted to evaluate the performance of USSF and the results obtained are presented in this section. All the experiments were conducted on a network of shared memory multi-processor (SMP) workstations. Each workstation consisted of two 300 MHz Pentium II processors with 128 MB of main memory. The workstations were networked using fast Ethernet. The experimental analysis of the USSF was conducted using three applications; namely the network simulation

framework [20], the PSAF [1], and queueing models [19]. Fig. 1 presents an overview of the interaction between the application frameworks and USSF. The following subsections present brief descriptions about the applications.

6.1. Network simulation framework

The NSF provides a collection of tools to ease modeling and simulation of large-scale networks. As shown in Fig. 1, the primary input to the framework is the topology to be simulated. The syntax and semantics of the input topology is defined by the topology specification language (TSL), which provides simple and effective techniques to specify hierarchical topologies [18]. A TSL specification consists of a set of interconnected topologies. A topology consists of a set of interconnected object instantiations (net-lists) along with necessary object definitions. The topology is parsed into an OO intermediate format (TSL-IF) using a TSL parser. The parser is generated using the purdue compiler construction tool set (PCCTS) [13]. Hierarchical TSL-IFs are elaborated or "flattened" (as illustrated in Section 5.1) prior to code generation. The elaborated TSL-IF is used to generate necessary C++ code that conforms to WARPED's API for simulation. The NSF in conjunction with WARPED is implemented in C++. The generated topology includes code to instantiate the necessary user-defined modules that provide descriptions for the components in the topology. The generated code is compiled along with the WARPED library, and the application program modules to obtain the final simulation executable. Further details on NSF are available in the literature [21,20].

The network models used in the experiments were constructed by interconnecting a set of subnetworks (representing a local area network) to form a larger network using the hierarchical modeling techniques supported by TSL. The subnetworks were modeled as a set of `nodes` interconnected by a `router`. Each `node` in the network model is driven using a `TrafficGenerator`. The `TrafficGenerator` can be used to generate traffic patterns (such as constant bit rate (CBR)) for modeling different network applications or workloads. Different random number generators based on statistical distributions (such as Poisson distribution and normal distribution) may also be used to generate network traffic. The `router` component is used to model a simple router (or a switch) in a network. It forwards the packets generated by a `node` to the corresponding destination `node` or adjacent `router` as the case may be. Information necessary for routing is established at the time of initialization of simulation. The `routers` build the tables for routing by exchanging information between interconnected `routers`. Interconnections between subnetworks was achieved by suitably interconnecting the `routers`. In the experiments conducted as a part of this study, the `routers` at the higher hierarchical level were interconnected with each other to model different routing domains such as a *stub* domains and *transit* domains [2]. The *transit–stub* network model (used in the experiments) has been shown to be a good model of the Internet [2,27]. The characteristics of the network models used in the experiments is shown in Table 1.

6.2. Performance and scalability analysis framework

The PSAF is a simulation-platform independent tool that can be used to analyze the scalability and performance of any discrete event simulator [1]. The centerpiece of the framework is a platform-independent *workload specification language* (WSL). WSL permits the characterization of simulation model using a set of fundamental parameters that influence the performance of a discrete event simulator [1]. The language provides constructs that can be used to describe synthetic as well as real

Table 1
Characteristics of TSL models used in experiments

Models	Number of components					Lines
	Nodes	Traffic generators	Routers	PacketSinks	Total	of TSL
L0N	6	6	2	6	20	53
L1N	30	30	11	30	103	76
L2N	300	300	111	300	1011	99
L3N	3000	3000	1111	3000	10,111	122
L4N	30,000	30,000	11,111	30,000	101,111	145
L5N	300,000	300,000	111,111	300,000	1,011,111	168

Table 2
Characteristics of WSL models used in experiments

Models	Number of components			
	Sources	Sinks	LPs	Total
WSL1	10	10	80	100
WSL2	100	100	800	1000
WSL3	1000	1000	8000	10,000
WSL4	10,000	10,000	80,000	100,000

world benchmarks. PSAF includes a *synthetic workload generator* (SWG) that can be used to generate synthetic benchmarks in WSL. WSL also provides hierarchical constructs that can be used to specify large workloads. The hierarchical constructs provide an efficient technique to scale models to desired proportions using the synthetic constructs of the language. PSAF also provides support to specify platform-specific translation routines that are used to generate a set of simulation models, specific for a given simulation environment, from a WSL description. PCCTS is used to generate the WSL parser. Hierarchical WSL specifications are elaborated (as described in Section 5.1) prior to generation of the simulation models. The generated models can be collectively or individually used as a benchmarking suite to explore the effects of different parameters on the performance of the targeted simulator. The synthetic models generated using PSAF closely reflect the characteristics of several *real-world* models [1]. Further details on the design and implementation of PSAF are available in the literature.

The characteristics of the synthetic models used in the experiments is shown in Table 2. These models were developed using the hierarchical WSL constructs. The models consisted of a set of interconnected synthetic LPs with fixed state size and event granularities. A set of event sources and sinks were used to exercise the various LPs constituting the synthetic model. The events were generated using random distributions supported by WSL. The WSL translator was used to generate the synthetic models from the corresponding WSL specification.

6.3. Queueing models

The queueing models used in this study were built using a library of components developed to explore the usefulness of unsynchronized simulations [19,24]. The stochastic nature of queueing models make it an interesting case for unsynchronized

simulation. Since WARPED and NOTIME present the same API, the queueing models developed using the library can be simulated using either of the parallel kernels without necessitating any changes to the models. The primary components of the queueing library are random sources, queues, servers, and statistics collectors. The components can be used to model any $\mathcal{G}/\mathcal{G}/m$ queueing system [24]. The $\mathcal{G}/\mathcal{G}/m$ queueing systems represent a generic class of queueing systems that play a central role in a number of physical systems [8]. In such a queueing system the distribution of interarrival times and service times are completely arbitrary. The system has m servers and the order of service is also arbitrary. Accordingly, the queueing library permits different random distributions to be associated with the source and servers. The model and scenario of the queueing system to be simulated is specified through configuration files. The queuing models used in the experiments are presented in Section 6.6.

6.4. Comparison between static and runtime elaboration

The statistics obtained from the experiments conducted to evaluate static and runtime elaboration techniques, using the network models shown in Table 1, are presented in Tables 3 and 4, respectively. The values shown in the tables are average values that were computed using the statistics from ten test runs. Fig. 7 shows a comparison between the time taken for code generation and compilation using static versus runtime elaboration techniques. The memory usage of the TSL parser was monitored by overloading the `new` and `delete` class of C++. As shown in Fig. 7, the time for static elaboration, code generation, and compiling the generated code for small network models is lower than the time for runtime elaboration. As the size of the network model increases the time for code generation and compilation increases exponentially with respect to the number of nodes in the network. The time for compiling the generated code for network model `L4N` consisting of 100,000

Table 3
Statistics on static elaboration

Model	Parsing time (s)	Elaboration time (s)	Peak memory usage (KB)	Code gen. time (s)	Lines of C++	Compile time (s)
L0N	0.00290	0.0079	10	0.00142	173	3.142
L1N	0.00349	0.0150	29	0.00396	733	4.319
L2N	0.00400	0.3993	267	0.03250	7103	19.832
L3N	0.00475	62.1024	2809	0.32100	70,803	957.079
L4N	0.00563	7698.5800	29,880	3.72000	707,803	N/A

Table 4
Statistics on runtime elaboration

Model	Parsing time (s)	Elaboration time (s)	Peak memory usage (KB)	Code gen. time (s)	Lines of C++	Compile time (s)
L0N	0.00290	0.021	1.33	0.0040	340	6.062
L1N	0.00349	0.031	1.99	0.0052	380	6.641
L2N	0.00400	0.123	2.66	0.0073	482	7.675
L3N	0.00475	1.480	3.33	0.0257	584	9.286
L4N	0.00563	15.800	4.66	0.2080	686	10.749
L5N	0.00626	65.010	4.92	2.0140	788	12.244

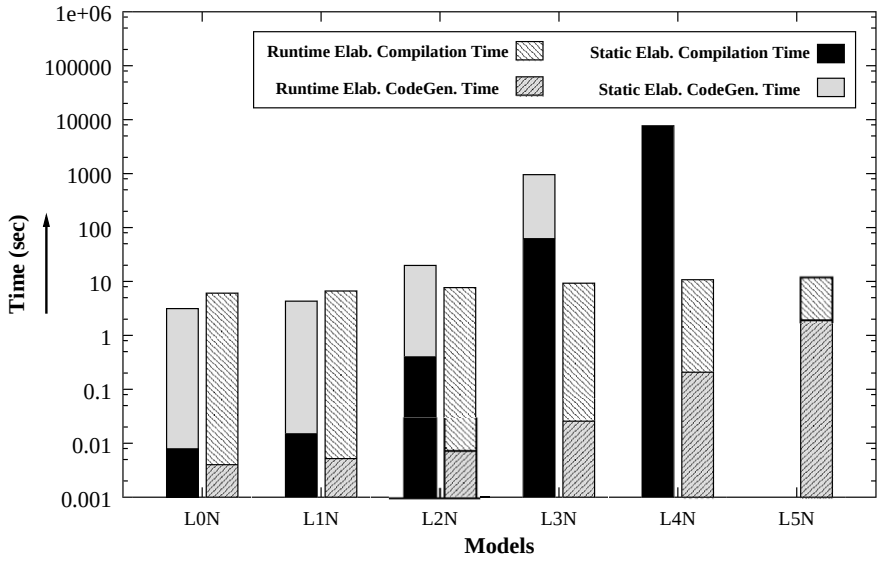

Fig. 7. A comparison between certain static and runtime elaboration parameters.

network components is not shown since the C++ compiler ran out of memory while compiling the generated code. Static elaboration technique was not employed for the model LN5 (consisting of more than a million nodes) since the technique failed for a considerably smaller network model (LN4 network model).

In contrast, the corresponding times for runtime elaboration are considerably smaller. The time for generating code for runtime elaboration is small since the network model is not elaborated prior to code generation. The time for compiling the code generated for runtime elaboration is lower since the size of the generated code (in terms of number of lines of C++ is shown in Table 4) is smaller. The reduction in the size of the generated code also reduces the size of the static executable. The drawback of runtime elaboration is that elaboration must be done each time the simulation is run. The time taken for runtime elaboration for the various models is shown in Table 4. Runtime elaboration occurs in each of the parallel clusters. Hence, irrespective of the number of parallel clusters used in the simulation, the time for runtime elaboration remains the same for a given network model.

6.5. Comparison between WARPED and USSF

The network and synthetic models shown in Tables 1 and 2 were simulated using WARPED and USSF with WARPED as the underlying simulation kernel. A very aggressive GVT computation was used so as to ensure rapid garbage collection to reduce memory consumption of the simulations. Runtime elaboration was utilized for simulating the models. The simulations were run in parallel using a varying number of processors. The LPs were randomly partitioned onto the parallel clusters. The simulation times of the network models shown in Table 1 are shown in Fig. 8(a) and (b). The parallel simulation times for the synthetic models using WARPED are shown in Fig. 9(a) and (b). The graphs also show the time for simulating the models using a sequential simulator. The sequential simulations were conducted using the sequential simulator that is available as a part of WARPED. The sequential simulator also uses WARPED's API and hence the models were run using the

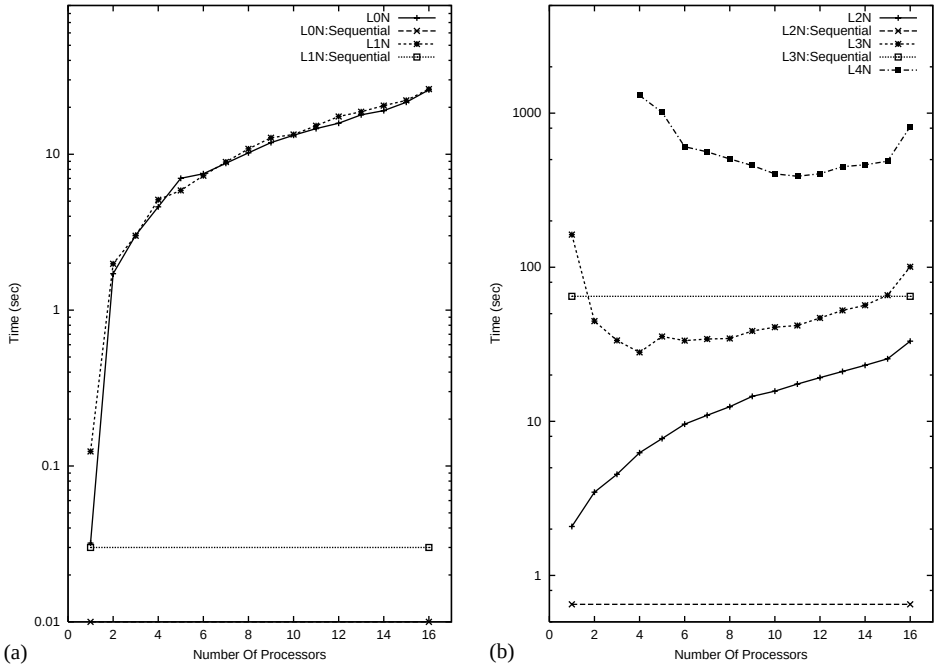

Fig. 8. Simulation time for TSL models with WARPED: (a) smaller models; (b) larger models.

sequential kernel without any changes. The sequential simulator could not be used to simulate the large models (such as L4N, L5N, and WSL4) as the simulations exceeded the memory limits of the systems and did not complete successfully. Fig. 10(a) and (b) present the corresponding simulation times of the models obtained using USSF. The data plotted in the graphs are the average simulation time values computed from 10 simulation runs.

As illustrated by the graphs shown in Figs. 8, 9, and 10; for both WARPED and USSF the simulation time for small models (such as L0N, L1N, WSL1, and WSL2) increases as the number of processors utilized in the simulation are increased. The increase in simulation time is due to the limited amount of parallelism available in the small models. Since parallelism is limited, increasing the number of processors utilized in the simulations merely increases the overheads of parallel simulation and the overall simulation times increase. In the case of medium sized models (such as L2N, L3N, WSL3, and WSL4) the performance improves as the number of processors are increased up to a certain threshold where the gains accrued by parallel simulation outweigh the overheads. Beyond the threshold point, the overheads of parallel simulation dominate as the number of processors are increased and the performance deteriorates. However in the case of the large models (such as L4N and WSL4) the simulation time improves as the number of processors are increased. The improvement in simulation time occurs because the models are large (consisting of 100,000 LPs or more) and sufficient amount of workload and parallelism is available to exploit the parallel processors. The simulation times for the large models using few processors is not shown either because the simulations took unreasonably long time or they did not complete as they exceeded the memory limits of the system. For example the L4N model could not be run with WARPED using fewer than 4 processors and the L5N model (consisting of more than a million LPs) could be run only using USSF on 16 processors.

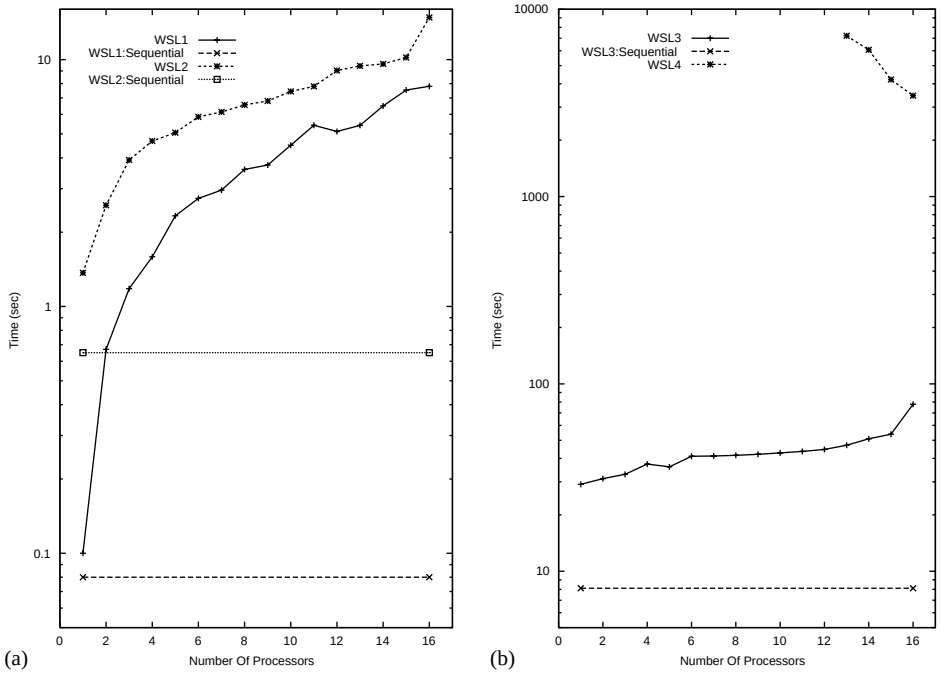

Fig. 9. Simulation time for WSL models with WARPED: (a) smaller models; (b) larger models.

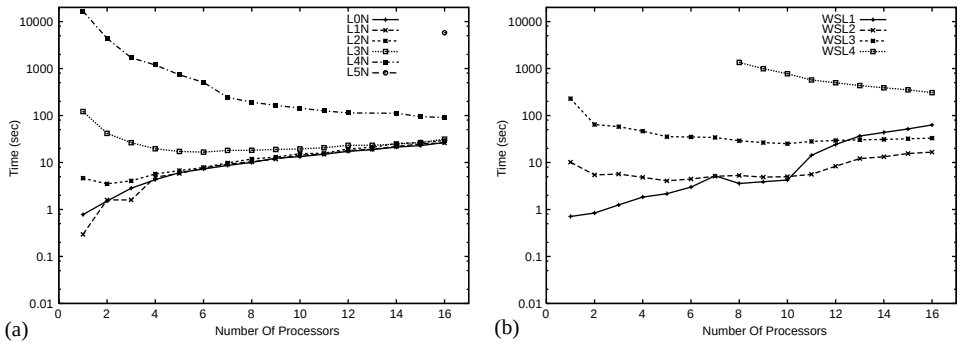

Fig. 10. Simulation time with USSF: (a) TSL models; (b) WSL models.

The graph in Fig. 11(a) presents a comparative picture between the simulation execution times of small models using WARPED and USSF. Fig. 11(b) presents the peak dynamic memory utilization of the models. The dynamic memory consumption of the simulations was measured by tracking the memory allocated and deallocated by overloading the C++ `new` and `delete` function calls. The peak memory shown in the graph represents the maximum of the memory consumed by any of the WARPED clusters. As illustrated by the graphs, the memory consumption of WARPED is well within the memory limits of the system and the WARPED performs better than USSF. The simulation times with USSF is higher due to additional overheads of USSF. Some of the USSF simulations performed better than WARPED simulations due to fewer rollbacks. The USSF simulations experienced fewer rollbacks because the overheads of the framework inherently throttle the simulation curtailing the aggressiveness of the optimistic simulations. It was also

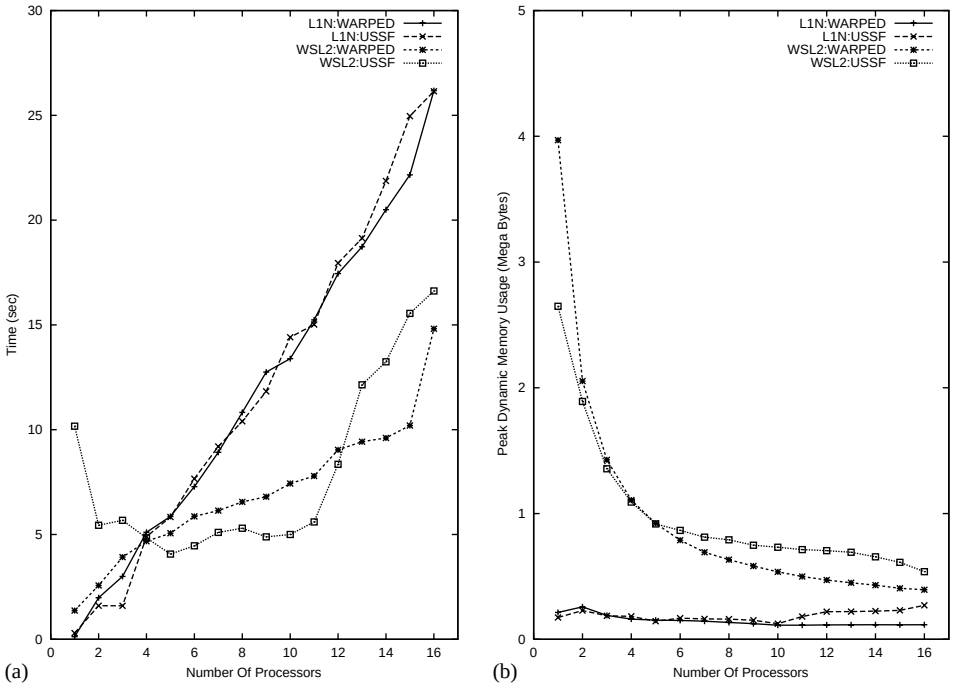

Fig. 11. (a) Simulation time and (b) peak dynamic memory usage of WARPED and USSF for smaller models.

observed that the performance of USSF simulations deteriorate more rapidly than WARPED simulations as the number of rollbacks increase. The two primary factors that increase the cost of rollbacks were: (i) since a number of LPs are aggregated into a single USSF cluster, for each rollback the cluster experiences, all the LPs in the cluster need to be rolled back to ensure consistency of the simulations; and (ii) since incremental state saving is used the overheads of restoring the state after a rollback is higher when compared to that of WARPED. The observation indicates that aggregation of LPs into USSF clusters results in a trade off between memory usage and simulation overheads. Although aggressive aggregation decreases memory usage it increases simulation overheads.

The graphs in Fig. 12(a) and (b) presents a comparison between the simulation time and peak dynamic memory consumption of the WARPED and USSF simulations. As shown by Fig. 12(b) the memory consumption of the simulations steadily decrease as the number processors increases since the memory requirements gets distributed across the processors. As illustrated by the graphs, the simulation times using WARPED and USSF are comparable when the memory consumption of WARPED falls well within the physical memory limits of the workstations used in the simulations. For example, WARPED simulation the `L4N` model on 4 processors has a peak memory consumption of about 90 MB. In contrast, the peak memory requirements of the USSF simulations were considerably lower (about 68 MB) and hence they perform better. However, as the number of processors are increased the size of the simulations fall well within the physical memory limits of the system and the performance of WARPED simulations steadily increase.

As shown by the graph in Fig. 12(a), although the memory requirements of WARPED drops well below the memory limits of the workstations, as the number of processors are increased, the performance of WARPED simulations do not

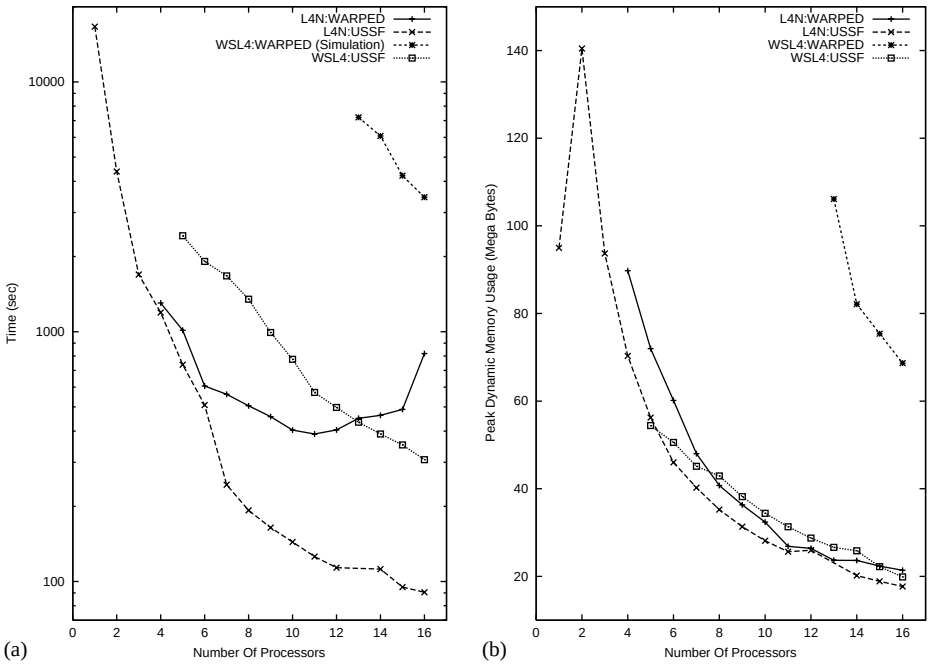

Fig. 12. (a) Simulation time and (b) peak dynamic memory usage of WARPED and USSF for larger models.

considerably improve. An analysis of the various parameters that influence the performance of the parallel simulations revealed that the primary factor that contributed to increase in simulation times was the cost of the initialization phase of the simulations. The initialization cost was high because of the overheads involved in distributing LP information to the various parallel clusters. For each LP, the WARPED kernel sends an event to all the other clusters in the simulation providing them the necessary information. For example in the case of the `L4N` model, consisting of 106,444 LPs and running on 16 processors, the 16 clusters exchange a total of 1,596,660 MPI messages. Hence, for large models the number of MPI messages used in the simulations increase as the number of processors are increased and hence the communication overheads dominate the initialization phase. On the other hand, since the USSF kernel collapses a large number of user-defined LPs in a single USSF cluster (or a WARPED process), the number of WARPED LPs are few and the number of MPI events used are few. Therefore, the USSF simulations have considerably smaller startup times compared to the WARPED simulations. Consequently, as shown in Fig. 12(a), the overall simulation time for USSF simulations is considerably lower than that of WARPED simulations.

6.6. Comparison between NOTIME and USSF

The experiments with USSF using NOTIME as the underlying parallel simulation kernel were performed using different queueing models. The queueing models were built in a hierarchical fashion by randomly cascading generic queueing systems to form larger systems. The queuing models were developed using the queueing model library described in Section 6.3. The runtime elaboration library was used to ease development of the queueing models. Table 5 presents the characteristics of the queuing models used in the experiments. The various queueing models were

Table 5
Characteristics of queueing models used in the experiments

Models	Number of components					
	Event generators	Sources	Queues	Servers	Statistic collectors	Total
Queue1	30	31	4	40	1	106
Queue2	300	312	41	410	1	1063
Queue3	3000	3122	4110	411	1	10,644
Queue4	30,000	31,222	41,110	4111	1	106,444

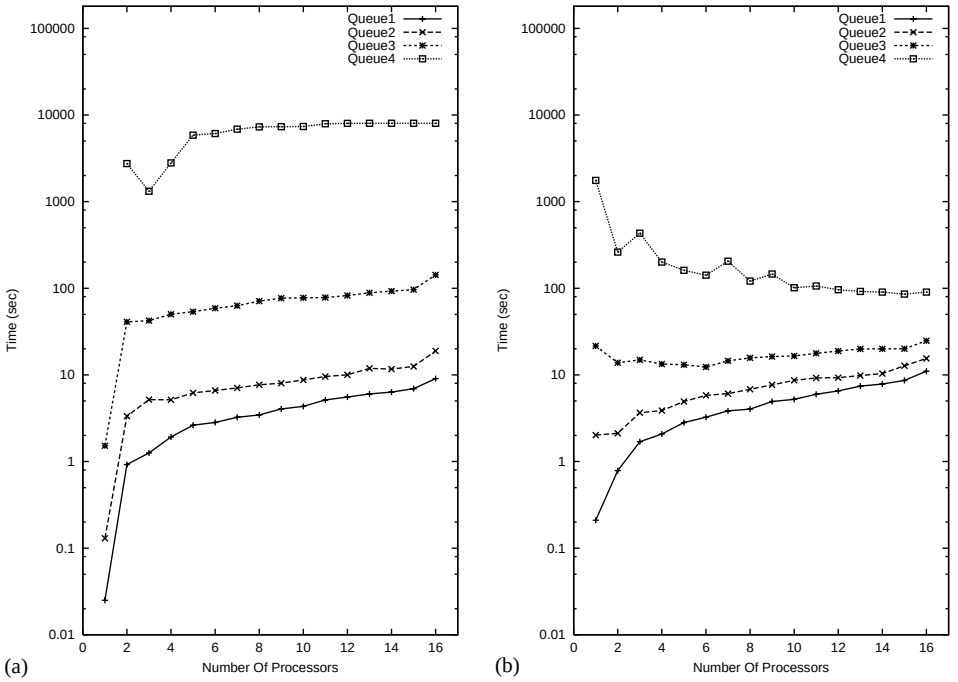

Fig. 13. Simulation times for queueing models using NOTIME and USSF: (a) using NOTIME; (b) using USSF.

simulated in parallel using a varying number of processors on a network of workstations. The LPs were randomly partitioned onto the parallel clusters. Figs. 13 and 14 present the simulation times and peak dynamic memory usage of the various queueing model simulations performed using, respectively, NOTIME and USSF (with NOTIME as the underlying simulation kernel). The data plotted in the graphs are the average simulation time values computed from 10 simulation runs.

As illustrated by Fig. 13(a) and (b), the parallel simulation times using USSF and NOTIME for small queueing models (such as `Queue1` and `Queue2`) increase as the number of process are increased. However, for the larger queueing model, namely `Queue4`, the performance of parallel simulations using `ussf` improves as the number of processors are increased. The statistics reflect the nature of the computation to communication ratio of the queueing models. In the case of small models, communication dominates computation, and hence the communication costs increase as the number of processors are increased and the overall simulation time

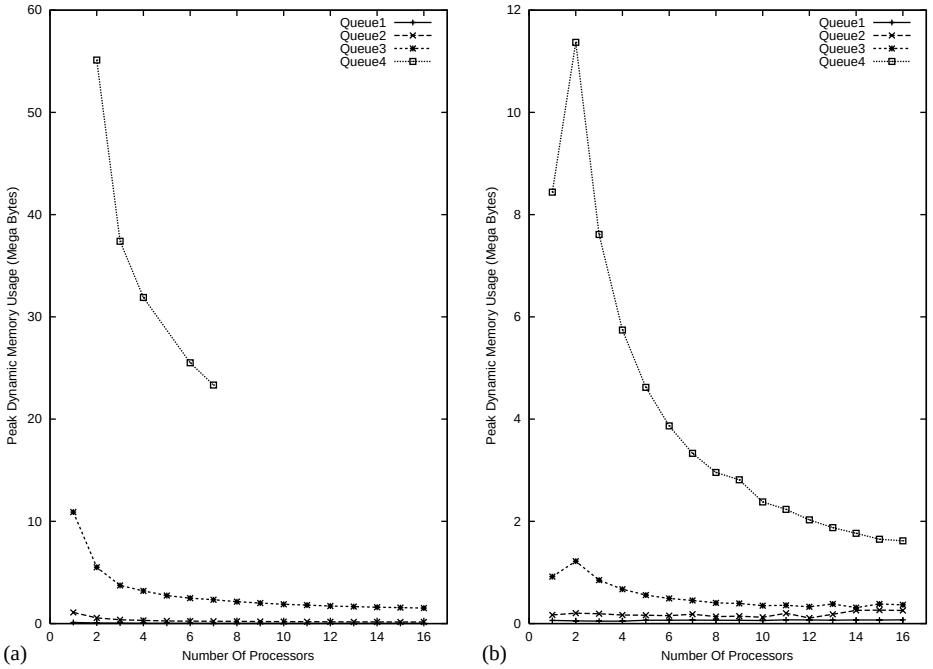

Fig. 14. Peak dynamic memory usage using (a) NOTIME and (b) USSF.

increases. In the case of the large models, computation dominates communication. Hence, as the number of processors are increased the computational overheads are distributed across the parallel processes and the simulation time decreases. The NOTIME simulation data for the `Queue4` model on one processor is not shown since the simulation exceeded the memory capacity of the workstation and did not complete successfully. The peak dynamic memory requirements of the simulations are shown in Fig. 14. As illustrated by the graphs, the memory consumption of the USSF simulations is considerably lower than the memory consumption of the NOTIME simulations. The significant difference in memory usage between 1 and 2 processor USSF simulations is indicative of the additional overheads necessary to enable parallel simulation.

Similar to the initialization overheads of WARPED, the initialization phase of NOTIME simulations involved exchanging a large number of messages to distribute information on the configuration of the simulations. Hence, the time for initializing NOTIME simulations increases as the number of processors are increased. In the case of NOTIME the cost of initialization phase is more pronounced due to the reduced overhead of the simulation kernel. However, since the USSF kernel collapses a number of user-defined LPs into a few NOTIME LPs, the number of messages exchanged during initialization is reduced. Therefore, as illustrated by the graphs in Fig. 13, the initialization time and overall simulation time for USSF simulations (using NOTIME) are considerably smaller than those with NOTIME.

7. Conclusions and future work

The steady growth in size and complexity of modern systems has required their simulation with modest hardware resources to enable detailed yet cost-effective study and analysis. An USSF was developed to ease simulation of ultra-large models

with limited hardware resources. The issues involved in the design and implementation of USSF were presented in this paper. The techniques used to reduce the static and dynamic memory requirements of large simulations were presented. An API for the runtime elaboration library was presented. Runtime elaboration was shown to out perform static elaboration for large models. A comparison between the performance of USSF using two parallel simulation kernels, namely WARPED and NOTIME, with raw WARPED and NOTIME simulations were presented. The experiments conducted indicate that USSF simulations perform better for large models. The experiments also demonstrate the capacity of the framework for simulating ultra-large systems using resource constrained platforms.

The USSF provides a tradeoff between memory requirements and simulation overheads by varying the number of LPs aggregated into each USSF cluster. LPs that share a common description are aggregated together. Therefore, the number of LPs that share a common description is a critical factor that determines the overall efficiency of the solution provided by USSF. USSF is an ideal candidate for simulating large applications which contain a number LPs that share a common description. Such models are typical in the domains of network modeling and very large-scale integrated-circuits (VLSI) design. It must be noted that USSF is a general purpose discrete event simulation framework and does not place restrictions on the nature of the discrete event model being simulated. It is also independent of the underlying synchronization mechanism.

The design and development of the USSF is a part of an ongoing research to improve the efficiency of large-scale simulations. Further studies are underway to improve the efficiency of USSF. Research is being conducted to determine an optimal level of aggregation for each model based on the availability of hardware resources. Techniques to dynamically (i.e., during the course of simulation) change the degree of aggregation and the number of USSF clusters used in a simulation are also being investigated. The effectiveness of USSF to enable large-scale simulations using conservative simulation techniques needs to be explored. Application of the techniques, used in USSF, for simulation of large-scale mixed technology system also provides an excellent avenue for further research.

References

[1] V. Balakrishnan, P. Frey, N. Abu-Ghazaleh, P.A. Wilsey, A framework for performance analysis of parallel discrete event simulators, in: Proceedings of the 1997 Winter Simulation Conference, Atlanta, GA, December 1997.

[2] K. Calvert, M. Doar, E.W. Zegura, Modeling internet topology, IEEE Comm. Mag. 35 (6) (June 1997) 160–163.

[3] C.D. Carothers, K.S. Perumalla, R.M. Fujimoto, Efficient optimistic parallel simulations using reverse computation, in: Proceedings of the 13th Workshop on Parallel and Distributed simulation, PADS'99, Atlanta, GA, May 1999, pp. 126–135.

[4] K. Fall, Network emulation in the Vint/NS simulator, in: Proceedings of the Fourth IEEE Symposium on Computers and Communications, Red Sea, Egypt, July 1999.

[5] R. Fujimoto, Parallel discrete event simulation, Comm. ACM 33(10) (October 1990) 30–53.

[6] P. Huang, D. Estrin, J. Heidemann, Enabling large-scale simulations: selective abstraction approach to the study of multicast protocols, in: Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Networks, Montreal, Canada, October 1998.

[7] D. Jefferson, Virtual time, ACM Trans. Programming Languages Systems 7 (3) (July 1985) 405–425.

[8] L. Kleinrock, Queueing Systems, Wiley, New York, NY, 1975.

[9] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Comm. ACM 21 (7) (July 1978) 558–565.

[10] D.E. Martin, T. McBrayer, P.A. Wilsey, WARPED: a time warp simulation kernel for analysis and application development, 1995, available on the www at http://www.ece.uc.edu/~++paw/warped/.
[11] D.E. Martin, T.J. McBrayer, P.A. Wilsey, WARPED: a time warp simulation kernel for analysis and application development, in: H. El-Rewini, B.D. Shriver (Eds.), 29th Hawaii International Conference on System Sciences (HICSS-29), Big Island, HI, January 1996, Computer Science Press, Vol. I, pp. 383–386.
[12] P. Martini, M. Rümekasten, J. Tölle, Tolerant synchronization for distributed simulations of interconnected computer networks, in: Proceedings of the 11th Workshop on Parallel and Distributed Simulation, PADS 97, Lockenhaus, Austria, June 1997, Society for Computer Simulation, pp. 138–141.
[13] T.J. Parr, Language Translation Using PCCTS and C++, Automata Publishing Company, San Jose, CA, January 1997.
[14] D.A. Patterson, J.L. Hennessey, Computer Architecture: A Quantitative Approach, 2nd Edition, Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1996.
[15] V. Paxson, S. Floyd, Why we don't know how to simulate the internet, in: Proceedings of the 1997 Winter Simulation Conference, Atlanta, GA, December 1997, pp. 44–50.
[16] B.J. Premore, D.M. Nicol, Parallel simulation of TCP/IP using TeD, in: Proceedings of the 1997 Winter Simulation Conference, Atlanta, GA, December 1997, pp. 437–443.
[17] R. Radhakrishnan, D.E. Martin, M. Chetlur, D.M. Rao, P.A. Wilsey, An object-oriented time warp simulation kernel, in: D. Caromel, R.R. Oldehoeft, M. Tholburn (Eds.), Proceedings of the International Symposium on Computing in Object-Oriented Parallel Environments, ISCOPE'98, Lecture Notes in Computer Science, Vol. 1505, Springer, Berlin, December 1998, pp. 13–23.
[18] D.M. Rao, R. Radhakrishnan, P.A. Wilsey, FWNS: a framework for web-based network simulation, in: A.G. Bruzzone, A. Uhrmacher, E.H. Page (Eds.), 1999 International Conference On Web-Based Modelling & Simulation, WebSim'99, San Francisco, CA, January 1999, Vol. 31, Society for Computer Simulation, pp. 9–14.
[19] D.M. Rao, N.V. Thondugulam, R. Radhakrishnan, P.A. Wilsey, Unsynchronized parallel discrete event simulation, in: Proceedings of the 1998 Winter Simulation Conference, Washington D.C., December 1998, pp. 1563–1570.
[20] D.M. Rao, P.A. Wilsey, An object-oriented framework for parallel simulation of ultra-large communication networks, in: Proceedings of the Third International Symposium on Computing in Object-Oriented Parallel Environments, San Francisco, CA, November 1999.
[21] D.M. Rao, P.A. Wilsey, Simulation of ultra-large communication networks, in: Proceedings of the Seventh International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS'99, College Park, MD, October 1999, pp. 112–119.
[22] G.F. Riley, R.M. Fujimoto, M.H. Ammar, A generic framework for parallelization of network simulations, in: Proceedings of the Seventh International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, College Park, MD, October 1999, pp. 128–135.
[23] S. Robinson, Simulation model verification and validation: increasing the users' confidence, in: Proceedings of the 1997 Winter Simulation Conference, Atlanta, GA, December 1997.
[24] N.V. Thondugulam, D.M. Rao, R. Radhakrishnan, P.A. Wilsey, Relaxing causal constraints in PDES, in: 13th International Parallel Processing Symposium, IPPS/SPDP '99, San Juan, Puerto Rico, April 1999, pp. 696–700.
[25] P.A. Wilsey, Modeling, analysis and simulation of computer and telecommunication systems, in: A. Kent, J.G. Williams (Eds.), Encyclopedia of Computer Science and Technology, Vol. 41, Marcel Dekker, Inc., New York, 1999, pp. 147–160.
[26] P.A. Wilsey, A. Palaniswamy, Rollback relaxation: a technique for reducing rollback costs in an optimistically synchronized simulation, in: International Conference on Simulation and Hardware Description Languages, Tempe, AZ, January 1994, Society for Computer Simulation, pp. 143–148.
[27] E. Zegura, K. Calvert, S. Bhattacharjee, How to model an internetwork, in: Proceedings of the IEEE INFOCOM, San Francisco, CA, April 1996, pp. 594–602.