Simulation of Ultra-large Communication Networks*

Dhananjai Madhava Rao and *Philip A. Wilsey*
Dept. of ECECS, PO Box 210030, Cincinnati, OH 45221–0030
phil.wilsey@uc.edu

Abstract

The steady growth in size and complexity of communication networks has necessitated corresponding advances in the underlying networking technologies including communication protocols. This multi-faceted growth has rendered analysis of today's ultra-large networks, a complex task. Simulations have been used to model and analyze communication networks. Complete models of the ultra-large networks need to be simulated in order to study crucial scalability and performance issues. Discrete event simulations of such ultra-large networks, with limited hardware resources is complex due to their sheer size. This paper presents the issues involved in the design of a framework to enable ultra-large simulations, consisting of millions of nodes. The parallel simulation techniques used, the application program interface needed for model development, and the results from experiments conducted using the framework are also presented.

1 Introduction

Computer and communication networks have grown in size and complexity in order to meet the demands of modern applications [6, 9]. These demands have necessitated corresponding growth in the underlying networking technology. Introduction of active networking techniques [1], that embed computational capabilities into conventional networking components, has furthered networking technology. Analyzing today's networks, that involve complex hardware and software interactions, is hard due to their size [6, 9]. Discrete event simulations have been employed to not only ease the analysis and exploration of complicated scenarios [6], but also play a vital role in developing intuitive ideas for real world networks [9].

Validation of the network topologies and components developed for simulation studies is also important. The network topologies should reflect the actual network sizes in order to ensure that crucial scalability issues do not dominate during validation of simulation results. Many networking protocols and mechanisms that work fine for small networks of tens or hundreds of computers may become impractical when the network sizes grow [9]. Events that are rare or that do not even occur in toy models may be common in the actual networks under study [9]. Detailed analysis using models of appropriate sizes is important to avoid what are popularly termed as *success-disasters* [9]. Paxson *et al* provide an excellent example to highlight the issues — "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" [9]. Since today's networks involve a large number of computers ranging from a few thousands to a few million nodes, modeling and simulating such ultra-large networks is necessary. However, modeling and simulation of ultra-large networks is hard due to their sheer size. The memory and computational resources needed to simulate such large networks, in acceptable time frames, are often beyond the limits of stand alone workstations. This paper presents the issues involved in the design and implementation of an ultra-large simulation framework, developed to ease modeling and simulation of ultra-large networks consisting of millions of nodes. The hurdles faced and the techniques adopted to alleviate them are presented in Section 2. A detailed description of the framework is presented in Section 3. Results obtained from the experiments performed using the framework are presented in Section 4. Section 5 provides some concluding remarks and pointers to future work.

2 Approach

The initial exploratory ultra-large network simulation studies conducted using the WARPED [10] parallel discrete-event simulator, indicated that the simulations place heavy demands on system resources such as processor cycles and

*Support for this work was provided in part by the Defense Advanced Research Projects Agency under contract DABT63–96–C–0055.

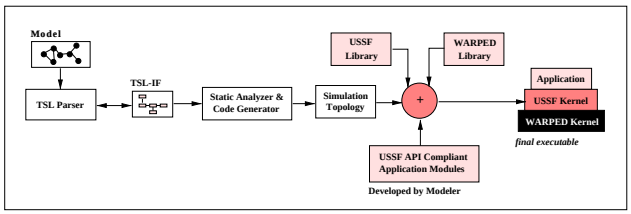

Figure 1. Overview of USSF

system memory [11]. Although the parallel simulation infrastructure of WARPED provided effective techniques to meet the computational needs, the demands on system memory continued to pose hurdles in enabling large simulations. Even though the virtual memory on the various workstations used for the parallel simulation studies were increased, it was observed that once the memory utilization crossed beyond the physical memory limits of the system, the simulations were unacceptably slow. In order to study the memory overheads of the simulations, the memory utilizations were classified into two categories: (i) memory requirements of the parallel simulation kernel; and (ii) memory requirements of the application models. The memory consumption of the simulation kernel arose due to the communication maps, the discrete events, and the state space that an optimistic kernel needs to maintain for its functionality [10]. The process descriptions and data used by the models contribute to the memory requirements of the application models. In order to effectively simulate ultra-large networks, the memory consumption of both the applications and the simulation kernel needed to be reduced.

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. Modification of the data structures used by the kernel would affect its performance and increase simulation time frames to unacceptable limits. Since many of the data structures needed to be duplicated at each parallel process, the memory gained by employing multiple workstations is modest. Regulating the memory consumption, by selectively maintaining data structures based on their demand, in a distributed system is complex and would considerably decrease the gains accrued by parallelizing the simulation. On the other hand, development of application models paying extreme care to memory utilization is also hard. The models of the nodes are complex [6] and often their memory consumptions cannot be reduced. Hence the only viable alternative to enabling large scale simulations was to regulate memory the usage of the application by building the necessary data structures on demand. Regulating the memory allocated by the application also regulates the memory usage of the underlying simulation infrastructure.

During our study of the network simulations [11], it was observed that many of the simulation processes share the same descriptions. The model descriptions, *i.e.*, the code to model the functionality, were the same but the states for each instance was different. If the model descriptions were decoupled from their data and states, the model descriptions could be reused. In other words, the same description could be associated with different states in order to simulate its various instances. Having only a single instance of the model description, instead of a few hundred thousands, provides a convenient mechanism to group similar objects together. This reduced the total number of process that the WARPED kernel needs to handle, which in turn reduced the size of its internal data structures. Decoupling of states from model descriptions enabled swapping of states in and out of the main memory. A cached state space was proposed in order to optimize the swapping of states. This dramatically decreased the run time memory requirements of the simulation. The aggregated processes present a very small state space (consisting of merely two offset pointers into a state queue) to the simulation kernel. This further reduced the memory requirements of the optimistic simulation kernel. The simulation kernel shares a single copy of the events between the source and destination processes, if they are present on the same processor. This helped prevent the exhaustion of main memory even though a large number of events exist in the simulation. All these techniques need to be carefully employed in order to simulate ultra-large network models. Developing applications paying special attention to these details is complex. The application writer would also need to perform complex book keeping activities. In order to ease development and simulation of ultra-large network models, an ultra-large simulation framework was proposed. The following section illustrates the design and working of the framework in detail.

3 An Overview of the USSF

To insulate the application developer from the intricacies needed to enable ultra-large simulations and to ease model development, an Ultra-large Scale Simulation Framework (USSF) was developed. As shown in Figure 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). TSL provides simple and effective techniques to specify hierarchical topologies [11]. Using TSL, large topologies can be built from smaller sub-topologies and sub-sub-topologies. The topology is parsed into an Intermediate Format (TSL-IF). TSL-IF not only provides a convenient mechanism to access information but also provides a well defined input to other modules in the system. Static analysis of the generated TSL-IF is performed to extract and collate common object definitions. The analyzed TSL-IF is then used to generate

```
design_file:
      ( include_clause )* ( tsl_topology )+ ENDFILE;
include_clause:
      INCLUDE QUOTE file_name QUOTE SEMI_COLON;
file_name: IDENTIFIER ( DOT IDENTIFIER )*;
tsl_topology:
      { label } object_definition_section
            object_instantiation_section net_list_section;
label: IDENTIFIER;
object_definition_section:
      OPEN_FLOWER ( object_definition )* CLOSE_FLOWER;
object_definition:
      object_name COLON url { parameter } SEMI_COLON;
object_name: IDENTIFIER;
parameter:
      QUOTE ( NUMBER | IDENTIFIER | STRING )* QUOTE;
url:
      IDENTIFIER ( DOT IDENTIFIER )*
            { COLON NUMBER DOT factory };
factory: IDENTIFIER ( DOT IDENTIFIER )*;
object_instantiation_section:
      OPEN_FLOWER ( object_instantiation )* CLOSE_FLOWER;
object_instantiation:
      IDENTIFIER COLON IDENTIFIER
            { NUMBER } { parameter } SEMI_COLON;
net_list_section:
      OPEN_FLOWER ( net_list )* CLOSE_FLOWER;
net_list:
      IDENTIFIER COLON ( IDENTIFIER )+ SEMI_COLON;
```

Table 1. TSL grammar

```
Basic { Router: HierarchicalRouter;
Node: SimpleNode; }

{ Switch: Router;
node1: Node; node2: Node; node3: Node; node4: Node;
node5: Node; node6: Node; node7: Node; node8: Node;
node9: Node; node10: Node; }

{ Switch: node1 node2 node3 node4 node5
node6 node7 node8 node9 node10 Basic;
node1: Switch; node2: Switch; node3: Switch;
node4: Switch; node5: Switch; node6: Switch;
node7: Switch; node8: Switch; node9: Switch;
node10: Switch; }

Level1 { Router: HierarchicalRouter; }

{ Level1Switch: Router;
subNet1: Basic; subNet2: Basic; subNet3: Basic;
subNet4: Basic; subNet5: Basic; subNet6: Basic;
subNet7: Basic; subNet8: Basic; subNet9: Basic;
subNet10: Basic; }

{ Level1Switch: subNet1 subNet2 subNet3 subNet4 subNet5
subNet6 subNet7 subNet8 subNet9 subNet10 Level1; }
```

Table 2. A Hierarchical Network (in TSL)

an optimal simulatable network topology. This technique helps to isolate the static optimization techniques from the implementation language. The current implementation of USSF, in conjunction with WARPED and the generated code, is in C++. The generated topology includes code to instantiate the necessary user defined modules that provide descriptions for the components in the topology. The models are developed using an simple and robust Application Program Interface (API), defined by USSF. The generated code is compiled along with the USSF library, WARPED library, and the application program modules to obtain the final executable. The following subsections describe the various components in detail.

3.1 Topology Specification Language (TSL)

The topology of the network to be simulated is provided to the framework in the Topology Specification Language (TSL) syntax. TSL provides simple and powerful constructs to describe complex networks using hierarchical descriptions. The LL(1) grammar of TSL, without semantic actions, is shown in Table 1. As illustrated in Table 1, a TSL description consists of three main sections, namely the *object definition section*, the *object instantiation section* and the *netlist section*. The object definition section contains information about the names of the various objects and their aliases in the description. Each object definition can be interpreted as defining a new type within the scope of a topology specification. The object instantiation section specifies the objects that are to be used to construct the topology. Every object instantiation should be associated with an object definition. A single object definition could be used by a number of object instantiations. This information is used by the static analyzer to identify and group together the objects built using the same model description. An optional set of parameters can be specified with each object instantiation. The netlist section specifies the interconnectivity (*netlist*) between the various instantiated components. The connectivity information is conveyed to the instantiated objects which may choose to use the information or ignore it. TSL permits a label to be associated with each topology specification. Using the labels, configurations can be combined together to build hierarchical topologies. This provides a powerful and convenient technique to specify ultra-large network topologies. An example of a TSL specification is shown in Table 2.

The input topology configuration is parsed by a TSL-Parser into a TSL Intermediate Format (TSL-IF). The current implementation of the TSL-Parser is built using the Purdue Compiler Construction Tool Set (PCCTS) [7]. The PCCTS framework uses the LL(1) TSL grammar (Table 1) to generate a parser that parses an TSL description. Output of the parser is the intermediate form of the input topology. TSL-IF forms the primary input to the static analysis and code-generation modules of USSF. TSL-IF is designed

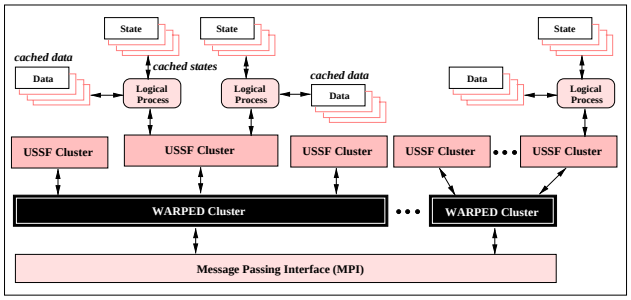

Figure 2. Layout of an USSF Simulation

to provide efficient access to related data from the various TSL sections. It closely reflects the structure of the TSL grammar. Object oriented techniques have been exploited to optimize memory consumption and ease representation of large hierarchical topologies. Generation of configuration information in TSL format can be automated to generate different inter-network topologies [2]. Further details on TSL is available in the literature [11].

3.2 Static Analysis & Code-Generation Modules

The static analysis and code-generation modules of USSF play an important role in reducing the memory consumed by the simulation. TSL-IF generated by the TSL parser forms the primary input. The static analyzer collates information on the modules repeated in the topology. This information is needed not only to group common instances together but also to reduce the memory consumed by the internal tables maintained by the USSF Kernel. The static analyzer is also responsible for allocating unique object identifiers or *id*s to each object instance in the topology. The static analyzer collates the information gathered from analysis into corresponding object definitions.

The code-generator uses the analyzed intermediate format to generate the final simulation topology. The generated topology is compliant with the USSF API. Depending on user specifications the generated code is directed to a single file or multiple files. Generation into multiple files enables compilation of the topology in parallel that reduces overall compilation times. The generated code is compiled and linked with the USSF library, the WARPED library, and user developed modules to obtain the final executable. Although the generated code is currently in C++ (to be compliant with the implementation of WARPED and USSF), the techniques employed are independent of the implementation languages.

3.3 WARPED

WARPED [10] is an optimistic parallel discrete event simulator developed at the Computer Architecture and Design Laboratory in the University of Cincinnati. It uses the Time Warp mechanism [5] for distributed synchronization. In WARPED, the logical processes (LPs) that represent the physical processes being modeled are placed into groups called "clusters". The clusters represent the operating system level parallel processes constituting the simulation. LPs on the same cluster directly communicate with each other without the intervention of the messaging system. This technique enables sharing of events between LPs which considerably reduces memory overheads. Communication across cluster boundaries is achieved using MPI [4]. LPs within a cluster operate as classical Time Warp processes; even though they are grouped together, they are not coerced into synchronization with each other. WARPED presents a simple and robust object-oriented application program interface (API). Further details on the architecture of WARPED and information on its API are available in the literature [10].

3.4 USSF Kernel

The core functionality of regulating the memory requirements of the application modules is handled by the USSF Kernel modules. The kernel is built on top of the WARPED simulator. The kernel modules present a interface similar to WARPED to the model developer. The core of the USSF kernel is the USSF cluster. The USSF cluster module represents the basic building block of USSF simulations. As specified by the WARPED API, each USSF cluster is assigned and addressed by a unique id. A USSF cluster performs two important functions. It not only acts as an LP to WARPED, it also acts as a cluster to the application programmer. As shown in Figure 2, 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 data and states of the processes are maintained in separate caches to satisfy concurrent accesses to the data and state spaces and to reduce cache misses. A Least-Recently-Used (LRU) cache replacement policy was adopted to swap the states into and out of the caches. Caching of the data and state spaces was adopted as they not only help in regulating the demands on main memory, they have been shown to improve performance of accessing data stored on file systems [8]. The necessary information needed to maintain cache coherence across rollbacks during simulation are present in the state information of the USSF cluster. This information is automatically restored by the WARPED kernel after rollbacks and the USSF cluster uses the restored data to ensure consistency of the various caches. Object oriented techniques have been used to decouple the various memory manage-

ment routines from the core. This design not 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 WARPED kernel into corresponding application process calls. It is also responsible for routing the various events generated by the application to the WARPED kernel. The WARPED kernel permits 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 ids. 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 ids 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 assigns contiguous ids 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 [5] that could occur in a Time Warp simulation. An simple incremental state saving mechanism with a fixed check-pointing interval is used for this purpose [3]. The states presented to WARPED by 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 simple application program interfaces (API). The API is illustrated in the following subsection.

3.5 USSF Application Program Interface

The USSF API closely mirrors the WARPED API [10]. This enables existing WARPED applications to exploit the features of the framework with very few modifications. 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 USSF Kernel presents an interface to the application developer for building local processes based on Jefferson's original definition [5] of Time Warp. Logical Processes (LPs) are modeled as entities which send and receive events to and from each other, and act on these events by applying them to their internal state. The basic functionality the kernel provides for modeling LPs are methods for sending and receiving events between the LPs

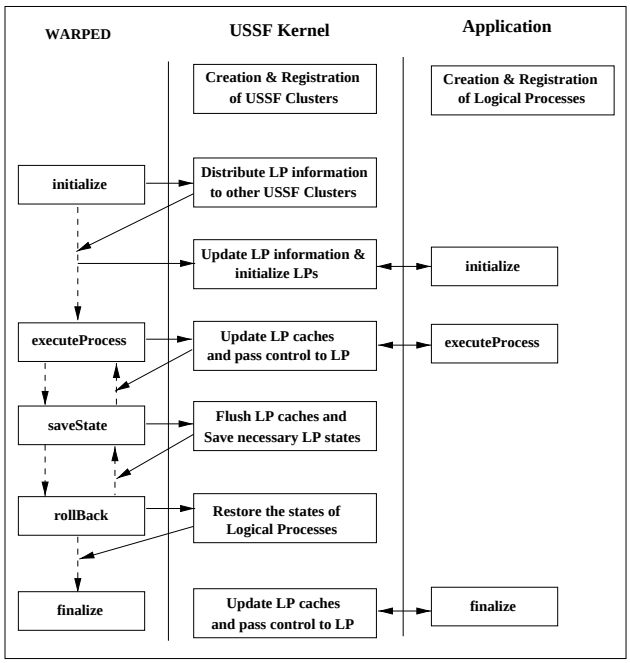

Figure 3. Flow of Control in USSF

and the ability to specify different types of LPs with unique definitions of state. On the other hand, the user is expected to override some of the kernel methods. The USSF kernel calls these methods 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 an 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. It is the responsibility of the modeler to assign unique ids to each LP. The static analysis modules in the USSF perform this functionality. Interfaces for constructing application data, states, and events are also specified. A detailed flow of control in the system via the API calls is presented in the following subsection.

3.6 Flow of Control in USSF

The ultra-large scale simulation framework is a parallel, distributed, and asynchronous system. It is necessary to illustrate the complex flow of control in order to fully highlight the issues involved in the various aspects of its design. Figure 3 attempts to illustrate the various activities that occur in the framework during simulation. The first phase of the simulation deals with setting up of the various USSF

clusters and the processes contained in them. 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 corresponding WARPED clusters. The WARPED kernel then invokes the `initialize` method of the various USSF cluster processes. The USSF clusters then exchange time stamped events distributing their internal tables to 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 WARPED kernel schedules USSF kernel events before the application events are scheduled. Processing the kernel events is crucial in order to ensure the internal data structures are updated before any of the application's processing begins. Once the updation of the internal data structures is complete, the USSF cluster 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. Any event generated by the applications is appropriately translated to USSF Cluster ids and despatched using the WARPED kernel's interfaces. The USSF cluster also saves the state of the various processes 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. Finally, when the `finalize` method 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.

4 Experiments

This section attempts to illustrate the various experiments conducted using the ultra-large simulation framework. The network model used to conduct the experiments was a hierarchical network topology. The topology was chosen as it is convenient to use the hierarchical nature of the network to construct larger networks. Verification of the network topology is also simplified. Validation of the simulations were done by embedding sanity checks at the various points in the models of the nodes constituting the network. The nodes representing the terminal points in the network generated traffic based on random Poisson distributions. The switches inter-connecting the nodes routed traffic by exploiting the fact that the topologies were hierarchical. The TSL specification of a two level hierarchical network consisting of hundred nodes is shown in Table 2. The network topology illustrated in the figure was recursively used in a hierarchical fashion in order to scale the network models to the required sizes.

The TSL parser was used to analyze the generated hierarchical topologies. Figure 4 illustrates the memory consumed by the parser. The memory consumption of the parser were monitored by overloading the `new` and `delete` calls in C++. The memory consumed by the raw input topology and the memory consumed after the topology had been statically analyzed are shown. The memory consumed by statically analyzed topology is much higher due to the fact that topology is flattened or elaborated during analysis. The flattening phase replicates the sub networks constituting the hierarchy. The replication involves creation of corresponding TSL-IFs for the sub networks. The identifiers associated with the nodes are hierarchically name-mangled to resolve any homo-graphs that could potentially arise. Name mangling increases the size of the identifiers. Hence the memory consumed by the statically analyzed topologies grows rapidly as the size of the networks grow.

The time taken to parse and build the TSL-IF, analyze the topology, and generate the simulatable topology are illustrated in Figure 5. Since the TSL-IF is constructed during parsing, the time for parsing and constructing the basic intermediate form cannot be individually measured. During time measurements, the memory monitors were turned off to ensure the overheads of monitoring memory usage was not included in the timings. The time taken to generate the topologies were measured on a workstation with dual Pentium II processors running Linux using the standard Unix time command. The low parsing times is due to the small size of the specifications. A small five level hierarchy with ten nodes at the lowest level (as shown in Table 2) is sufficient to specify a network with a million nodes. As the graphs illustrate, due to the hierarchical nature of the topology, the time required to statically analyze and generate the flattened topology increases rapidly as the size of the base topology increase.

Figure 6 presents the time taken for simulating the generated topologies with USSF and WARPED. Each node generates events to be routed to a random destination using a normal distribution. A constant packet size of 64 bytes was employed in the experiments. The data was collected using eight workstations inter-connected by fast Ethernet. Each workstation consisted of dual processor Pentium II processors with 128 MB of RAM (Random Access Memory) running Linux. The simulation time for the smaller configurations is high due to the initial setup costs. Figure 7 illustrates the average memory requirement at each node for the various topologies. The memory utilized by each WARPED cluster was monitored by overloading the `new` and `delete` calls of C++. The memory utilized was averaged at an interval of 10 seconds. The average memory consumed by each cluster was then averaged to yield the average memory consumed by the simulations. An aggressive GVT period

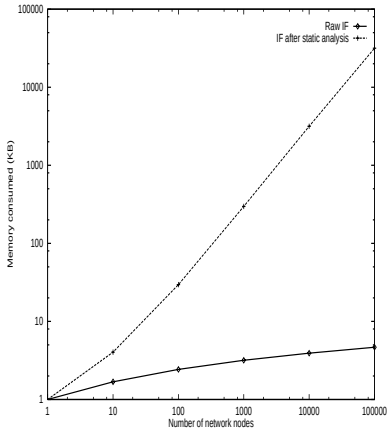

Figure 4. Memory consumption in TSL

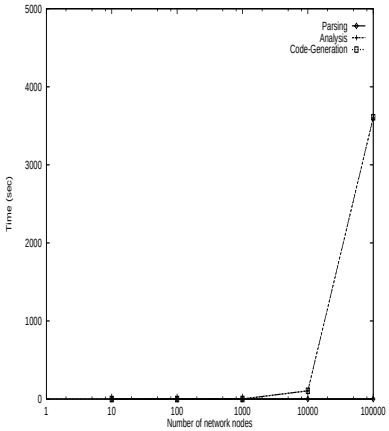

Figure 5. Timing information on TSL

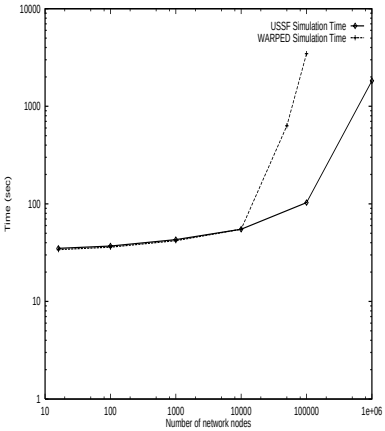

Figure 6. Simulation execution times

was used to ensure rapid garbage collection by WARPED. The routines to track memory usage were turned off during timing analyses. As illustrated by the figures, the memory consumption of the WARPED kernel increases rapidly as the simulation size increases. The experimental system ran out of memory while performing a million node simulation using WARPED, and hence the data for a million node simulation is not shown. As shown in Figure 6, the performance of WARPED deteriorates when the sizes of the various parallel processes exceed the physical memory limits. The careful study indicated that the degradation in performance was associated to operating system overheads involved with performing necessary input-output operations associated with maintaining virtual memory. The Figures 6 and 7 illustrate the advantages of employing USSF to reduce memory consumptions and improve performance. Figure 8 illustrates the memory consumption patterns obtained from a 100,000 network nodes simulation. The memory utilization gradually increases until garbage collection occurs in the system. The steep decrease in memory usage indicates the garbage collection phases in the system. Although garbage collection in the various parallel processes was triggered for every two simulation time units, the number of events in the simulation (due to concurrency and causal relationships in the model) contribute to the considerable variations in memory consumption.

In order to study the scalability issues of the framework, the hierarchical topology consisting of 100,000 was used. The number of processors used was varied between 2 and 16. The data was collected using a network of workstations. Each workstation consisted of dual processor Pentium II processors with 128 MB of RAM running Linux. Figure 9 presents the timing information obtained from this experiment. The times were measured using the standard Unix time command. The performance of the framework increases as the number of processors are increased since more computational resources are available for the concurrent processes to use. As illustrated by the experiments, the ultra-large scale simulation framework enables simulation of very large communication networks. The current implementation of USSF does not include a number of other proposed optimizations. The initial goal of the research was to enable such large-scale simulations. Studies are being conducted in order improve the performance of the framework.

5 Conclusion

Computer and communication networks have steadily grown in size and complexity to meet the demands of modern applications. The underlying networking components including the communication protocols have correspondingly increased in number and complexity. To ease the analysis of such complex networks, discrete event simulation have been employed. Simulation analysis with models built to reflect the ultra-large sizes of the networks is important to study scalability and performance issues. The size of such simulations place very high demands on memory which forms the primary bottleneck in simulation. Reducing memory consumption by the various models is hard if not impossible due to their inherent complexity. Hence techniques to efficiently regulate memory consumption have to be exploited to enable ultra large simulations. To insulate the application developer from such intricacies and to ease modeling and simulation of large networks, an ultra-large simulation framework was developed. The issues involved in the design and development of the frame-

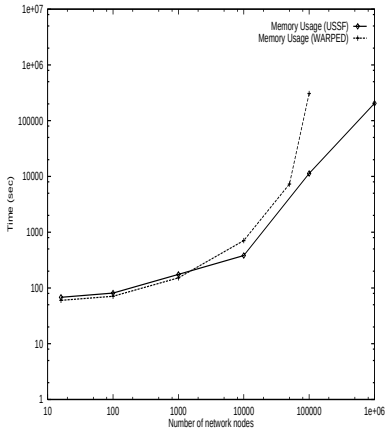

Figure 7. Average memory consumption

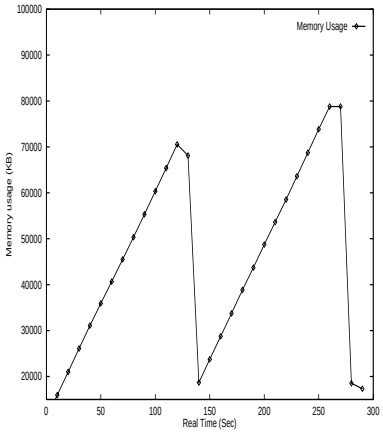

Figure 8. Typical memory usage

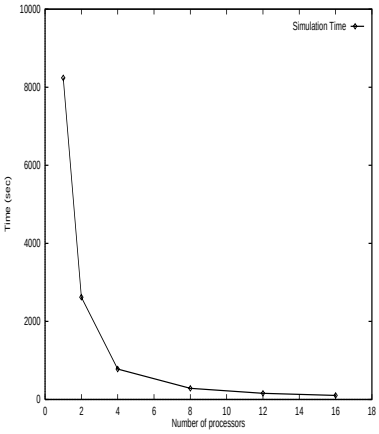

Figure 9. Framework's Scalability Data

work were presented. The experiments conducted using the framework were illustrated. From the experimental results the capacity to perform such large simulations in resource restricted platforms was demonstrated.

The initial goal of the research activity was to enable large scale simulations. Studies are being conducted to further improve the performance of the framework. Online monitoring and visualization tools have been proposed to aid analysis of the network models. Studies to explore alternative parallel simulation techniques and synchronization mechanisms are underway. USSF provides a convenient and effective means to model and study, present day and futuristic ultra-large communication networks.

References

[1] J. M. S. David L. Tennenhouse, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, Jan. 1997.

[2] K. C. E. Zegura and S. Bhattacharjee. How to model an internetwork. In *Proceedings of IEEE INFOCOM*, Apr. 1996.

[3] R. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, Oct. 1990.

[4] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA, 1994.

[5] D. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):405–425, July 1985.

[6] A. M. Law and M. G. McComas. Simulation software for communications networks: The state of the art. In *IEEE Communications Magazine*, pages 44–50, Mar. 1994.

[7] T. J. Parr. *Language Translation Using PCCTS and C++*. Automata Publishing Company, January 1997.

[8] D. A. Patterson and J. L. Hennessey. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc, San Mateo, CA, 2^{nd} edition, 1996.

[9] V. Paxson and S. Floyd. Why we don't know how to simulate the internet. In *Proceedings of the 1997 Winter Simulation Conference*, pages 44–50, Dec. 1997.

[10] 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)*, volume LNCS 1505, pages 13–23. Springer-Verlag, Dec. 1998.

[11] D. M. Rao, R. Radhakrishnan, and P. A. Wilsey. FWNS: A Framework for Web-based Network Simulation. In *1999 International Conference On Web-Based Modelling & Simulation (WebSim'99)*. Society for Computer Simulation, Jan. 1999.