

PLATFORM-INDEPENDENCE AND SCHEDULING IN A MULTI-THREADED REAL-TIME SIMULATION

Paul C. Sugden*

Melissa A. Rau[†]

Unisys Corporation
NASA Langley Research Center
Mail Stop 169
Hampton, VA 23681

P. Sean Kenney[‡]

Systems Development Branch
NASA Langley Research Center
Mail Stop 125B
Hampton VA 23681

Abstract

Aviation research often relies on real-time, pilot-in-the-loop flight simulation as a means to develop new flight software, flight hardware, or pilot procedures. Often these simulations become so complex that a single processor is incapable of performing the necessary computations within a fixed time-step. Threads are an elegant means to distribute the computational workload when running on a symmetric multi-processor machine. However, programming with threads often requires operating system specific

calls that reduce code portability and maintainability. While a multi-threaded simulation allows a significant increase in the simulation complexity, it also increases the workload of a simulation operator by requiring that the operator determine which models run on which thread. To address these concerns an object-oriented design was implemented in the NASA Langley Standard Real-Time Simulation in C++ (LaSRS++) application framework. The design provides a portable and maintainable means to use threads and also provides a mechanism to automatically load balance the simulation models.

Introduction

Aviation research often relies on real-time, pilot-in-the-loop flight simulation as a means to develop new flight software, flight hardware, or pilot procedures. Concepts involving pilot interaction may require concurrent simulation of multiple, independent or loosely coupled simulation models. A single processor may not be sufficient to perform all of the necessary computations of these complex flight simulations. A means to alleviate this problem is to distribute the execution of independent simulation models among multiple processors. Symmetric multi-processor

* Software Engineer, Member AIAA.

[†] Software Engineer.

[‡] Aerospace Engineer, Member AIAA.

Copyright © 2001 by the American Institute of Aeronautics and Astronautics, Inc. No copyright is asserted in the United States under Title 17, U.S. Code. The U.S. Government has a royalty-free license to exercise all rights under the copyright claimed herein for Governmental purposes. All other rights are reserved by the copyright owner.

machines allow the independent simulation models to be distributed among event loops executing in parallel. Each processor must be able to complete the computations of the models assigned to it before the end of the frame. Therefore, the number of simulation models that can be executed in a real-time multi-processor simulation is limited by the number of processors available for real-time use and by the computational requirements of the simulation models.

Traditionally, multi-processor computers are used by starting separate processes on each processor. The processes interact with each other via shared memory, message-passing, or some other form of inter-process communication (IPC). While multi-process implementations are functional, they are notoriously complex, difficult to maintain, and limit the use of polymorphism in certain modern object-oriented programming languages⁶. Multi-threaded solutions, on the other hand, allow multiple threads of execution to operate within the same address space, eliminating the need for IPCs, and eliminating restrictions on the use of polymorphism. Threads are supported by most modern operating systems and thereby enable the development of an elegant, platform-independent solution to the problem of how to simulate multiple vehicles concurrently.

In previous work¹ a generic design was presented for the use of threads in the Langley Standard Real-Time Simulation in C++ (LaSRS++) framework on a multi-processor machine. While functional, the design had several shortcomings. First, the design was implemented with only the IRIX operating system from Silicon Graphics Incorporated in mind. This restricted the framework to only being able to perform multi-threaded operations on the IRIX platform. Second, the design unnecessarily tied the thread classes to the IRIX barrier and lock implementations. This coupling increases the difficulty of implementing these features on different platforms and maintaining the framework during operating system upgrades. Clearly a more portable and maintainable design is needed. The design also failed to provide a mechanism to determine a feasible simulation model distribution for the number of threads available during a real-time simulation. If the distribution selection is

left to the simulation developer or user, experience shows that even in a small-scale simulation, this process can be costly, involving educated trial and error guesses when a feasible distribution may not even exist. A more effective solution is to provide automatic scheduling of each model for execution by a particular thread.

To address these issues, the existing design evolved into a new form that is both portable and maintainable and provides the mechanisms for an automatic scheduling algorithm to distribute a set of simulation models across a number of threads, such that the load is as uniform as possible. The new design blends the pre-existing thread design with the portability abstraction used in the LaSRS++ application framework for other operating system services².

While the application of these techniques in the LaSRS++ framework is used for the special purpose of flight simulation, the thread design and the automatic scheduling algorithm are independent of the purpose of the application. The design is a simple, user-friendly solution for the execution of any sequential, real-time application, running on a symmetric multi-processor architecture, whose iterations can be resolved into two or more independent sequences of computation.

Design Requirements

A preexisting design for multi-threaded, real-time simulation¹ was extended to be platform independent and incorporate automatic scheduling of models to processors. The design had the following requirements for threads:

1. The design must support a variable number of threads selected at run-time.
2. The design must be portable.
3. The design must provide a mechanism for thread synchronization that does not jeopardize real-time deadlines.

The scheduling algorithm had the following requirements:

1. The algorithm must allow a variable number of simulation models to be processed on each thread.

2. The algorithm must automatically distribute independent components of the workload amongst available threads in such a way that real-time deadlines are not jeopardized.
3. The algorithm must recognize if a feasible workload distribution does not exist.

A design that met the above requirements was implemented in the LaSRS++ application framework. Object-oriented programming techniques were used extensively to hide implementation details and maximize code reuse. Well-known design patterns were used where appropriate to promote simplicity and readability of code, and encapsulate specific platform dependencies. The Bridge, Singleton, Factory and Mediator patterns were used extensively³. Using the Bridge pattern, the implementation of an abstraction can be changed without affecting clients. The Singleton pattern ensures that there is a single, globally accessible instance of a class. The Factory pattern is a specialization of the Singleton pattern that is used to encapsulate all platform-dependent, conditional compilation. The Mediator pattern is an encapsulation of object interaction that promotes loose coupling by preventing objects from interacting directly.

Operating System Requirements for Real-Time

In order to build a multi-threaded, real-time simulation for a particular symmetric multi-processor platform running a general-purpose operating system, the operating system must provide certain capabilities. Traditional general-purpose time-sharing operating systems schedule processes using a preemptive scheduling policy. This means that a process is forced to relinquish control of the processor once its time slice has expired so that other processes can get processor time. Since hard real-time deadlines could be jeopardized by preemption, an operating system that can support real-time simulations must provide a mechanism for disabling the preemptive scheduler on processors reserved for real-time use.

Traditional time-sharing operating systems also place the address space of processes in virtual memory. Virtual memory allows the operating system to handle loads that are more demanding in terms of space than the machine's random access memory (RAM) will

allow. This is accomplished by swapping pages of RAM in and out of secondary storage. Since secondary storage access is significantly more costly in terms of time than access to RAM, a suitable operating system must provide the capability of locking the text and data segments of a process into RAM.

Conventional operating systems for symmetric multi-processors perform a task called load balancing. Load balancing refers to the migration of processes amongst the processors in such a way that the total system load is distributed evenly across all available processors. Since a real-time thread must never share processor resources with other standard processes, the operating system must provide a mechanism for forcing a thread to run on a particular processor.

By default, the operating system is usually unaware of multiple threads of execution within an address space, leaving the scheduling of these threads as a task for the process. This type of thread is commonly referred to as a process-scope thread. However, for multi-threaded real-time simulation applications, multiple threads must be executing in parallel, which by definition is impossible for process-scope threads. The operating system must provide a mechanism for creating threads that are schedulable by the kernel rather than from within the process. This type of thread is commonly called a system-scope or kernel thread.

If a general-purpose operating system has all of these capabilities, it is suitable for multi-threaded, real-time simulation. Each thread of such a simulation must be a system-scope thread within a process that has been locked into RAM, set to run on a dedicated processor, with preemptive scheduling disabled.

Object-Oriented Approach to Threads

The C++ language does not provide any direct (object-oriented) support for multi-threaded programming. Two types of classes are required for developing object-oriented, multi-threaded applications. These are classes that represent threads, and classes representing thread synchronization mechanisms. Thread synchronization mechanisms include mutexes and barriers. The mutex is a mechanism that ensures

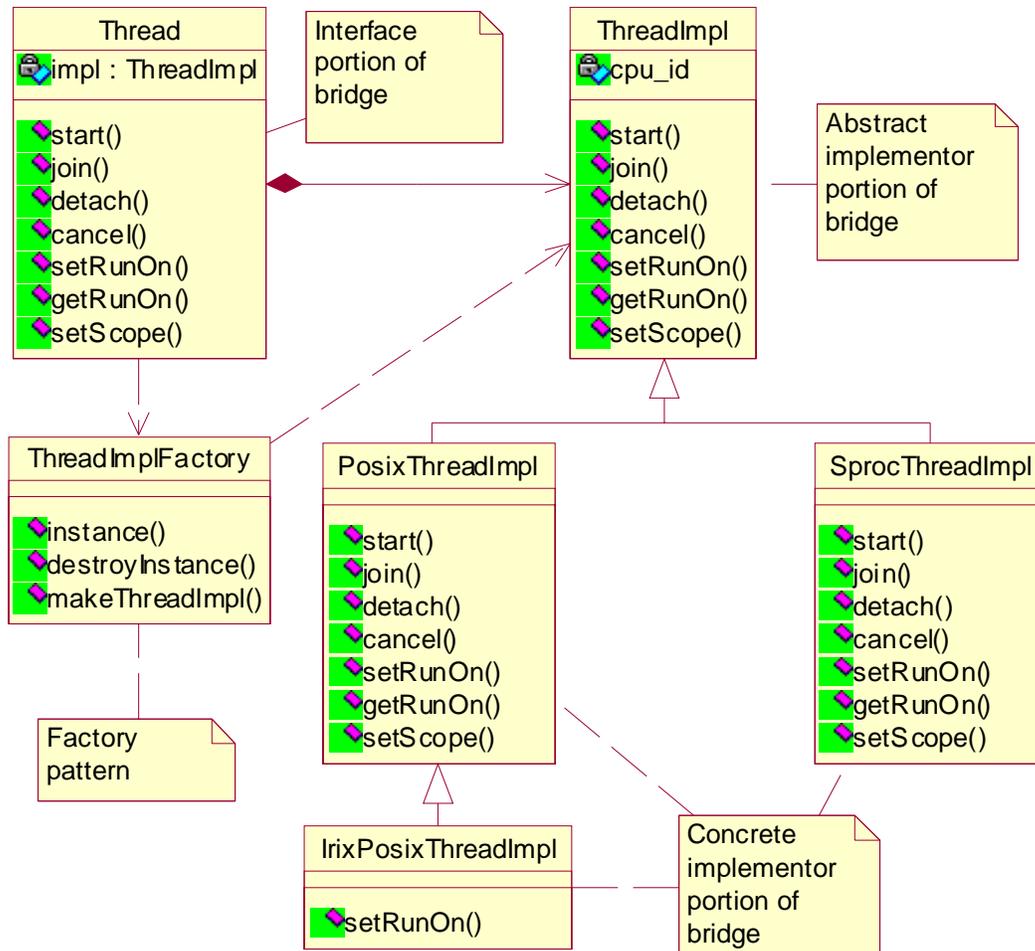


Figure 1 Thread Design

mutually exclusive access to data or to non-thread-safe portions of code (critical sections) in a multi-threaded process. The barrier provides a collection point for some specified number of threads of execution, and ensures that none continue until the specified number of threads have arrived.

Classes were developed for the LaSRS++ application framework that provide a platform-independent interface to thread, mutex, and barrier facilities. For each of these services, the bridge and factory design patterns were used. The bridge pattern prevents clients of thread-related services from being affected by implementation details. This allows the implementation portion of these classes to be changed without requiring any modification to client components. The factory design pattern encapsulates

all platform-specific and build-option-determined conditional compilation. This significantly improves the portability and maintainability of the LaSRS++ framework code.

Figure 1 shows a UML class diagram of the LaSRS++ thread implementation. When a client constructs a new *Thread*, *ThreadImplFactory* constructs the appropriate implementation (*ThreadImpl*). The specific type of implementation is transparent to the client. The LaSRS++ design for *Barrier* and *Mutex* is analogous to the design for *Thread*.

To further increase the maintainability of the thread implementation, the interfaces of the thread-related abstraction classes were structured after the Portable Operating System Interface (POSIX) standard, commonly called pthreads. Pthreads are currently

supported under several variants of the UNIX operating system including Solaris, Digital UNIX, IRIX, and Linux⁴. As a result, the use of pthreads allows us to build on any of these platforms using the *PosixThreadImpl* as the concrete implementation portion of the bridge. This source-code portability is beneficial in an environment of potentially frequent architecture changes⁸.

A requirement of the multi-threaded simulation uncovers a deficiency in the POSIX standard. There is nothing in the POSIX application programming interface (API) for specifying a particular processor on which a thread must run. This eliminates the possibility of constructing a completely platform-independent concrete thread implementation for multi-threaded, real-time applications. A system-scope pthread may be scheduled on a specific processor for its lifetime, but the call to do this is platform-specific. However, due to the use of the factory and bridge design patterns, the work required to implement platform-specific pthreads is minimal. The *PosixThreadImpl* class has a virtual method *setRunOn()* that takes the processor number as an argument. To create a pthread that will run successfully on the IRIX operating system, a class named *IrixPosixThreadImpl* was created. This class inherits from *PosixThreadImpl* and overrides the *setRunOn()* method with the IRIX-specific system call for running a thread on a given processor. In this way, the reuse of the existing thread classes is maximized, thus minimizing the work required for implementing platform-specific thread models.

Real-Time Thread Synchronization

While each thread in a multi-threaded real-time simulation is operating on independent tasks, there is often some portion of the event loop that cannot be executed before all threads have completed a certain portion of their task. An example of this is computation of relative geometry between simulation models. Regardless of how relative geometry computations are distributed, the state of all models must have been updated before any of these computations can take place. A barrier mechanism is used to implement this sort of rendezvous.

Operating systems supporting multi-threaded programs also support barrier mechanisms. Programs that are not subject to strict real-time deadlines should generally use the operating system supplied barrier facility. However, if a thread blocks on a barrier system call, it may not be guaranteed to resume execution within an acceptable amount of time to meet a real-time deadline. The alternative is to implement a spin-loop barrier mechanism. In this barrier, a primary thread waits at the barrier for all remaining threads to arrive. As each thread arrives, it locks itself and waits in a spin-loop conditioned on the value of that lock. Once all threads have arrived, the primary thread releases the lock for each remaining thread, and all threads may continue.

Scheduling Algorithm

Efficient thread synchronization alone does not guarantee that real-time deadlines will be met. The computational load must be balanced across the processors such that no deadlines are missed during real-time execution. Before any automatic scheduling assignments can be made, an approximation for the maximum execution time of each independent job must be determined. Assuming no a priori knowledge of the job execution times, the following algorithm is used to accomplish this task in as efficient a manner as possible.

1. Jobs are randomly distributed across all real-time simulation threads.
2. Each thread performs any necessary initialization for all jobs assigned to its processor.
3. Each thread performs its jobs a fixed number of times, tracking the maximum execution time of each. It is assumed that the observed maximum is a good estimate.

Once this phase has been completed, the timing information can be used as input to one of several load-balancing algorithms that attempt to find a valid distribution of jobs. The load-balancing problem that must be solved can be stated as follows:

Let $P = \{p_1, p_2, \dots, p_n\}$ be the set of n processors and $J = \{j_1, j_2, \dots, j_m\}$ be the set of m jobs with associated

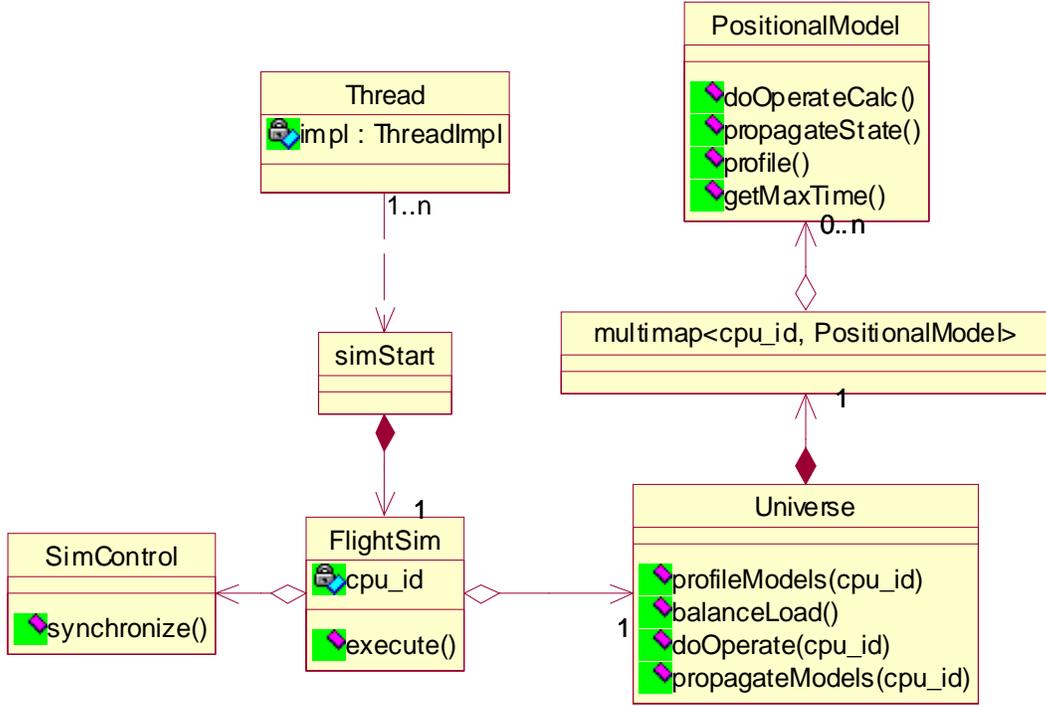


Figure 2 Relationships Between LaSRS++ Core Classes

compute times $C = \{c_1, c_2, \dots, c_m\}$. Let $L = \{l_1, l_2, \dots, l_n\}$ be the total compute time on each processor, and let T be the period of a real-time iteration. Determine a mapping $f : J \rightarrow P$ such that:

1. $\sum_{i=1}^n \left(l_i - \frac{\sum_{j=1}^m c_j}{n} \right)^2$ is minimized
2. $\forall l_i \in L, l_i < T$

Where $l_i = \sum c_j \mid f(j) = i$.

These requirements ensure that the workload is balanced as evenly across the processors as possible, and that no processor is overloaded such that deadlines are missed. This is under the assumption that it is possible for the given workload to be executed by the processor set.

While this problem is **NP**-complete⁹, an algorithm was developed that yields near-optimal solutions in a reasonable amount of time. Jobs are sorted by non-increasing execution time. Starting with the largest

job, each job is assigned to the processor that is currently supporting the smallest workload. If a job cannot be placed, it is assumed that the current workload cannot be supported in real-time by the given processor set.

The computational requirements of this algorithm do not jeopardize real-time deadlines because the algorithm is initially executed prior to running the simulation, and may be repeated at non-critical points of execution after starting (e.g., in a RESET mode). Within larger scale simulations where faster load-balancing algorithms are desirable, any algorithm may be substituted. Much research has been done with various load-balancing techniques, such that an effective algorithm can be applied within any arena.

Application of these Techniques to LaSRS++

Both the platform-independent thread abstraction and the load-balancing algorithm presented above were incorporated into the LaSRS++ framework. The relationships between various objects that are central to the framework are depicted in figure 2. The sequence of execution, from construction of certain

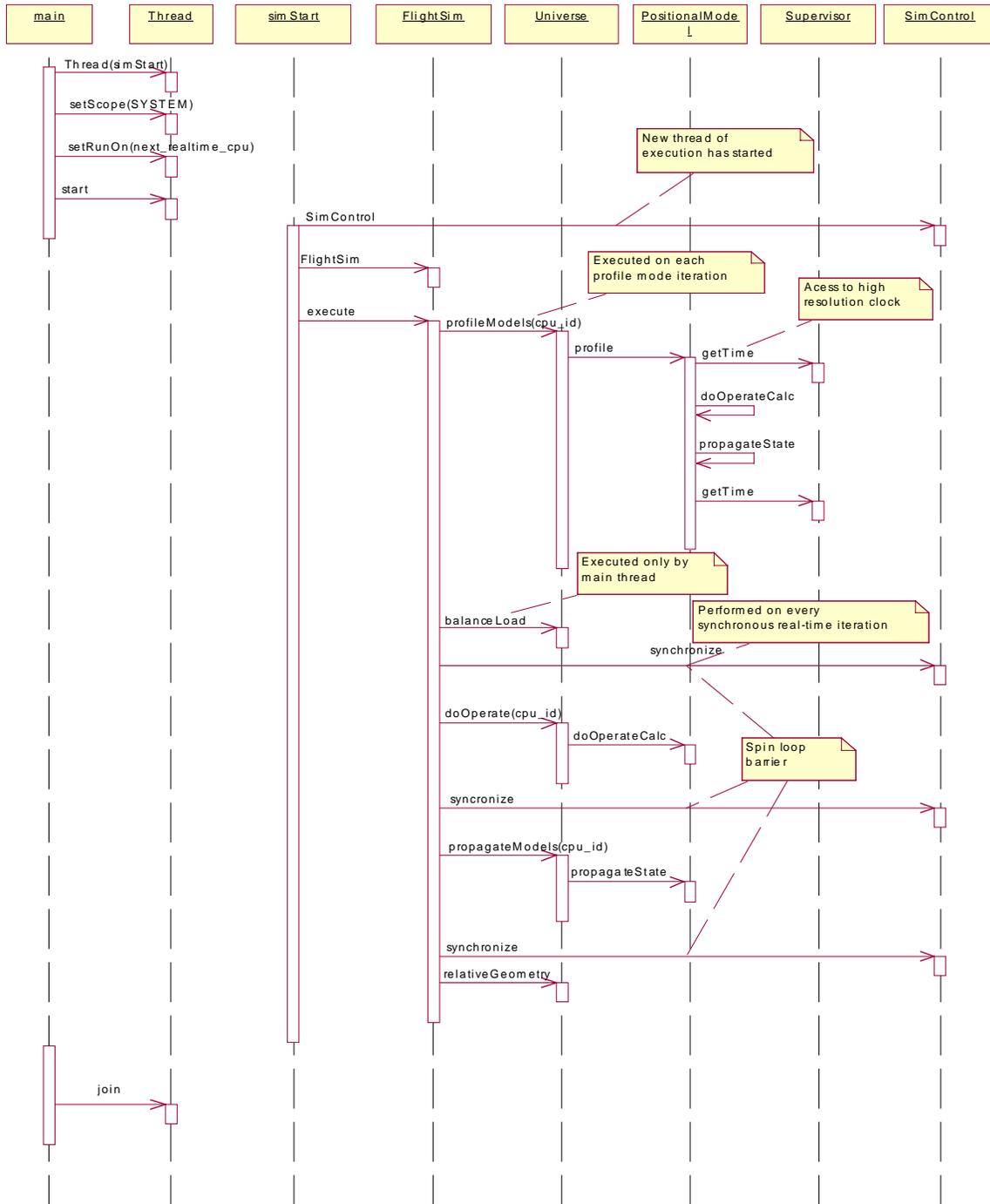


Figure 3 Core LaSRS++ Sequence Diagram

core objects to synchronous real-time execution, is depicted in figure 3.

After being launched, the LaSRS++ simulation process locks the segments of its address space into RAM; sets its priority to the highest level available on the system;

and isolates, restricts, and disables preemptive scheduling on each available processor. For each dedicated processor, the process constructs one system-scope thread and starts it on this processor. These threads each begin execution in a function that

constructs and executes a *FlightSim* object. At construction time, the *FlightSim* object is given the processor number on which the constructing thread is running.

In the LaSRS++ framework, the real-time simulation event loop is defined by the *FlightSim execute()* method. Construction and execution of multiple *FlightSim* objects by separate *Threads* results in multiple simulation event loops executing in parallel. The event loop consists of several distinct phases, some of which may not begin until all computations from the previous phase are complete (i.e. updating states of positional models must have been completed before calculation of relative geometry between models can begin). Each such phase begins with a call to *SimControl::synchronize()*. This method is an implementation of a spin-loop barrier, as previously discussed, that will not return until it has been called by all simulation threads. This ensures that all *FlightSim* objects are executing the same portion of the event loop at the same time.

While *FlightSim* objects represent the encapsulation of the periodic sequence of events in LaSRS++, *Universe* is the maintainer of the simulation model data. All simulation model objects are derived from *PositionalModel*, and are contained within the *Universe* singleton. *Universe* mediates between *PositionalModels* and all other components of the simulation framework, meaning that a *PositionalModel* may only be modified through action taken by the *Universe*.

To manage the active models during execution, the *Universe* has a member data structure that maps each model to the processor on which it is to be executed. The structure is a *multimap* container from the C++ Standard Template Library (STL)⁵. The *multimap* maintains a sorted list of key and value pairs. The pairs stored in the *multimap* contained by *Universe* are the processor identifiers and *PositionalModel* objects. Use of the STL provides a portable and robust data structure, which has been thoroughly tested and whose interface is well defined.

Within the *multimap*, *PositionalModels* are found using the processor identifier as the search key.

Correspondingly, all of the *Universe's* methods that act on *PositionalModels* take a processor number as an argument. When one of these methods is called, only the *PositionalModels* associated with the specified processor are affected. This allows multiple *FlightSim* objects to cause the *Universe* to modify disjoint subsets of *PositionalModels* in parallel.

Among *Universe's* methods are *profileModels()*, *doOperate()*, and *propagateModels()*. *ProfileModels()* is used by *FlightSim* to execute models in order to gain an estimate of the maximum amount of compute time that each will take. This information is required to complete the load-balancing operation. *DoOperate()* is used by *FlightSim* to update the outputs (e.g., forces and moments) of a model during synchronous real-time. *PropagateModels()* is used by *FlightSim* to integrate a simulation object's state and to compute outputs from the new state that belong to the next frame.

After the simulation process has been constructed and initialized, the workload is profiled and balanced across the available processors. To do this, the *Universe* randomly associates an equal number of *PositionalModels* with each available real-time processor. The simulation is initially in PROFILE mode. In this mode, *FlightSim* objects call the *Universe's profileModels()* method. Since there is a *FlightSim* object for each real-time processor, this results in all models getting profiled. The main simulation thread then calls the *Universe balanceLoad()* method. Using the algorithm discussed in the previous section, the *Universe* modifies the mapping of *PositionalModels* to processors within its *multimap* such that the load is as evenly distributed as possible. If a feasible workload distribution does not exist, the user is informed, and the simulation will not be allowed to transition to the real-time OPERATE mode.

After the simulation has transitioned into the synchronous real-time OPERATE mode, the *FlightSim* objects perform the following periodic sequence of steps:

1. *SimControl::synchronize()*

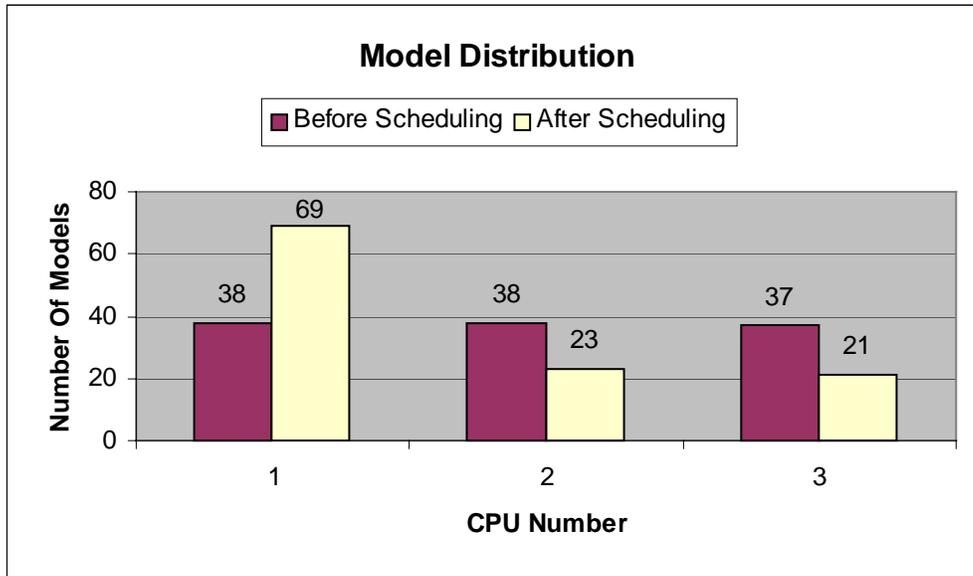


Figure 4 Workload Distribution

2. *Universe::doOperate()*
3. *SimControl::synchronize()*
4. *Universe::propagateModels()*
5. *SimControl::synchronize()*
6. *Universe::relativeGeometry()*

Experimental Results

The capabilities of the multi-threaded simulation and the scheduling algorithm were tested with a mixed workload of transport aircraft, fighters, and missiles. The models and their associated maximum compute times are presented in Table 1.

Model Type	Cost (ms)
B757	4.35 ms
Table 1 Positional Models and Associated Costs	
F/A-18E	6.59 ms
F-15A	0.99 ms
F-16A	0.87 ms
Launch Envelope Missile	0.01 ms

The LaSRS++ real-time simulation runs on an SGI Onyx with eight processors, three of which are available for the parallel execution of simulation threads. The simulation is run at a frame rate of 50 Hz, corresponding to a 20 ms frame time. To

demonstrate the effectiveness of the use of multiple threads and the load-balancing algorithm, a workload was created comprised of three B757s, two F/A-18Es, 10 F-15As, and 10 F-16As. Each fighter is equipped with four launch envelope missiles. Note that without the use of multiple threads, this workload could not be executed within real-time constraints.

Figures 4 and 5 demonstrate the effectiveness of the load-balancing algorithm on the given workload. Prior to balancing, the number of models for each processor is similar, but the workload on the first processor is approaching the upper bound of 20 ms. A slight variation in the model computations or the system performance in this configuration could result in a frame overrun – an unacceptable occurrence in a real-time simulation. The load-balancing algorithm then successfully schedules the models in a configuration that ensures that the load on all processors is well within the boundaries of the real-time deadlines.

Concluding Remarks

The design has allowed the framework to be compiled and run on the SGI and the Sun platforms, and it will be ported to the Linux and Win2000 platforms in the near future. Moving the framework to a new platform only requires the development of several

implementation classes that can be unit tested before use with the framework.

The abstractions found in the thread design make the framework easy to maintain. Any modifications to the operating system that might require changes to thread, barrier or lock classes would require changes only to a platform specific implementation. The modifications could then be verified using the unit test for the modified class and would not require extensive re-testing of the framework itself.

The scheduling algorithm alleviates unnecessary burden on the simulation operator by providing an effective solution to the load distribution problem.

Although the design presented in this paper was originally designed to support flight simulation at NASA Langley Research Center, the design could be used in any object-oriented framework to more effectively utilize the capabilities of symmetric multi-processor machines.

Bibliography

- [1] David Geyer. *The Use of Multiple Threads in an Object Oriented Real-Time Simulation*, Paper Number AIAA 99-4338, August, 1999.
- [2] P. Sean Kenney, et al. *Using Abstraction to Create a Portable Object-Oriented Simulation*, Paper Number AIAA 99-4340, August, 1999.
- [3] Gamma E., Helm R., Johnson R., Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.
- [4] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1997.
- [5] Nicolai M. Josuttis. *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1999.
- [6] Michael Madden, et al. *Constructing a Multiple-Vehicle, Multiple-CPU Using Object-Oriented C++*. Paper Number AIAA-98-4530, August, 1998.
- [7] Richard A. Leslie, et al. *LaSRS++ An Object-Oriented Framework for Real-Time Simulation of Aircraft*. Paper Number AIAA-98-4529, August, 1998.
- [8] Bill O. Gallmeister. *POSIX.4: Programming for the Real World*. O'Reilly & Associates, Inc. Sebastopol, California, 1995.

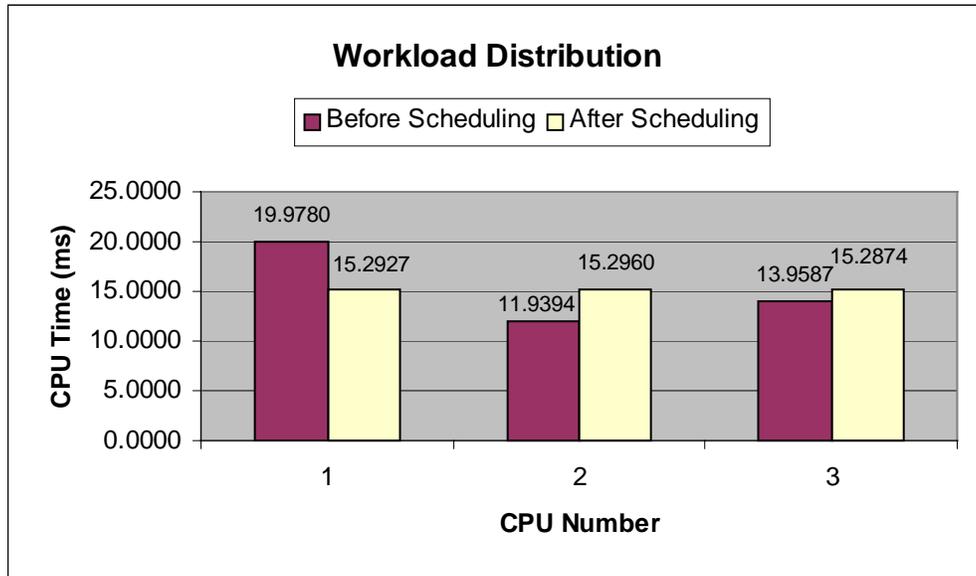


Figure 5 Model Distribution

[9] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1994.