Fall 2007

An open framework for highly concurrent hardware-in-the-loop simulation

Ryan C. Underwood

Recommended Citation

Underwood, Ryan C., "An open framework for highly concurrent hardware-in-the-loop simulation" (2007). *Masters Theses*. 4591.
https://scholarsmine.mst.edu/masters_theses/4591

AN OPEN FRAMEWORK FOR HIGHLY CONCURRENT HARDWARE-IN-THE-LOOP SIMULATION

by

RYAN C. UNDERWOOD

A THESIS

Presented to the Faculty of the Graduate School of the

UNIVERSITY OF MISSOURI-ROLLA

In Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE IN COMPUTER SCIENCE

2007

Approved by

Dr. Bruce M. McMillin, Advisor

Dr. Daniel R. Tauritz

Dr. Mariesa L. Crow

ABSTRACT

Hardware-in-the-loop (HIL) simulation is becoming a significant tool in prototyping complex, highly available systems. The HIL approach allows an engineer to build a physical system incrementally by enabling real components of the system to seamlessly interface with simulated components. It also permits testing of hardware prototypes of components that would be extremely costly to test in the deployed environment. Key issues are the ability to wrap the systems of equations (such as Partial Differential Equations) describing the deployed environment into real-time software models, provide low synchronization overhead between the hardware and software, and reduce reliance on proprietary platforms. This thesis introduces an open source HIL simulation framework that can be ported to any standard Unix-like system on any shared-memory multiprocessor computer, requires minimal operating system scheduler controls, provides a soft real-time guarantee for any constituent simulation that does likewise, enables an asynchronous user interface, and allows for an arbitrary number of secondary control components. The availability of multiple processor cores significantly simplifies the framework by reducing complex scheduling. The framework is implemented in FACTS Interaction Laboratory (FIL) which provides a real-time HIL simulation of a power transmission network with physical Flexible AC Transmission System (FACTS) devices. Performance results are given that demonstrate a low synchronization overhead of the framework.

ACKNOWLEDGEMENTS

I would like to thank Dr. Bruce M. McMillin, my advisor, for his support and guidance during this project, Dr. Mariesa L. Crow for her invaluable knowledge and excellent coursework, and Dr. Daniel R. Tauritz for his advice and support. I would also like to thank Julia Albath, who provided the necessary guidance and motivation for me to conclude the work on this project.

This work was supported in part by DOE/Sandia under contract number 291871, in part by NSF MRI award CNS-0420869 and CSR award CCF-0614633, and in part by the UMR Intelligent Systems Center. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

TABLE OF CONTENTS

Page

ABSTRACT iii
ACKNOWLEDGEMENTS iv
LIST OF ILLUSTRATIONS vi
GLOSSARY vii
SECTION
- 1 INTRODUCTION 1
- 2 REVIEW OF LITERATURE 3
- 3 APPROACH OF THIS THESIS 6
- 4 OVERVIEW OF HIL SIMULATION FRAMEWORK 9
  - 4.1 REQUIREMENTS 9
  - 4.2 ASYNCHRONOUS HARDWARE MANAGEMENT 9
  - 4.3 INTERPROCESS COMMUNICATION ALTERNATIVES 11
- 5 DISCUSSION OF HIL SIMULATION FRAMEWORK 15
  - 5.1 SYSTEM BLOCK DIAGRAM 15
  - 5.2 SYSTEM HARDWARE/SOFTWARE 15
  - 5.3 SOFTWARE VERSIONS 17
  - 5.4 HAL (HARDWARE ABSTRACTION LAYER) 20
  - 5.5 SIMULATION CORE (FLATTEN4) 20
  - 5.6 SIMULATION CORE TASK (SIM_DRIVER) 23
  - 5.7 PI (PROPORTIONAL-INTEGRAL) CONTROLLER(S) 23
  - 5.8 LOAD BANK CONTROLLER 24
  - 5.9 SIMULATION CONSOLE (SIMGUI) 27
  - 5.10 SYSTEM STATE TRANSMITTER (SIMLISTENER) 28
- 6 RESULTS AND DISCUSSION 30
- 7 CONCLUSIONS 35
- 8 FUTURE DIRECTIONS 36

BIBLIOGRAPHY 37
VITA 41

LIST OF ILLUSTRATIONS

Page

Figure 4.1: Comparison of hardware approaches. 10
Figure 4.2: The locking scheme, with stale flags. 13
Figure 5.1: High level diagram of the simulation system. 16
Figure 5.2: The FIL laboratory setup. 18
Figure 5.3: A FACTS device with embedded computer. 19
Figure 5.4: Internals of the FACTS device. 19
Figure 5.5: Automata of the HAL. 21
Figure 5.6: Data flow for an interface variable being sampled into the system... 21
Figure 5.7: Simulation driver automata. 23
Figure 5.8: The programmable load bank console. 25
Figure 5.9: The static load banks. 25
Figure 5.10: The load bank power storage array. 26
Figure 5.11: The simulation computer. 27
Figure 5.12: The power system simulation GUI. 28
Figure 6.1: Lock acquisition latency, HAL delay 100 microseconds. 31
Figure 6.2: Lock acquisition latency, HAL delay 50 microseconds. 31
Figure 6.3: Lock acquisition latency, HAL delay 5 microseconds. 32
Figure 6.4: Lock acquisition latency, no HAL delay. 32
Figure 6.5: Critical section latency, HAL delay 100 microseconds. 33
Figure 6.6: Critical section latency, HAL delay 50 microseconds. 33
Figure 6.7: Critical section latency, HAL delay 5 microseconds. 34
Figure 6.8: Critical section latency, no HAL delay. 34

GLOSSARY

118-bus, 179-bus systems: Test power system data, each representing a real-world power distribution network.

A/D, D/A: Analog to Digital, Digital to Analog, Analog to Digital Conversion/Convertor, Digital to Analog Conversion/Convertor.

HAL: Hardware Abstraction Layer. In this thesis, it refers to a process that acts as a hardware I/O aggregator in order to remove the direct hardware access responsibilities from the other processes in the system.

Hardware In the Loop (HIL): A technique for testing a hardware prototype that involves either simulation of the hardware prototype or simulation of the physical system with which the finished product would interface.

Hardware Under Test (HUT): The physical hardware prototype that is being developed and validated by the HIL technique.

Interface variable: A variable which is sampled externally from A/D hardware and then injected back into the system. A real-world variable can be updated by the simulation, but the simulated value is always overwritten by the external value obtained from A/D subsystem.

Jacobian: A square matrix of the partial derivatives of the variables in a system evaluated at the current time step.

Mismatch: The error term in Newton's method. Used to determine whether or not the iteration has converged on a solution yet.

Newton-Raphson (NR) iteration (Newton's method): An approach for iteratively solving a non-linear system of equations.

Nyquist rate: When discretizing an analog variable, the non-inclusive lower bound on the sampling rate that would be sufficient to capture all of the changes in slope of the variable; the Nyquist rate is equal to twice the waveform's frequency. An analog variable must be sampled at a frequency greater than the Nyquist rate in order to capture all of the variable's dynamics.

PI Controller: Proportional-Integral Controller, a type of controller that maintains an output set point based on a feedback loop.

Power Hardware In The Loop (PHIL): A hardware-in-the-loop technique with the specific property that the simulated system and real system are coupled by the exchange of real electric power. Also referred to in the literature as “Sim-Stim” interface.

Rest Of System (RoS), Rest Of Power System (RoPS): The part of the power system which is being computed by the HIL simulation.

SMP: Symmetric Multi-Processing, a general term for a multiprocessor computer that has a unified memory architecture.

System variable: A variable such as voltage or phase angle that is either given as a precondition to the simulation (as in slack or PV buses) or a dependent variable calculated internally to the simulation (for PQ buses); a system variable is only directly updated by the simulation, not by external events.

1 INTRODUCTION

Hardware-in-the-loop (HIL) is a technique in which a portion of a given system is a simulation algorithm and a portion of the same system is a hardware implementation. The real hardware is connected via a digital interface to a computer-simulated system model. HIL is beneficial in testing prototypes of those devices which have complex internal algorithms, those which would cause catastrophe if failed in the field, and those for which building a laboratory test environment is difficult or impossible [1].

In the HIL, an important goal is that the simulated system demonstrate dynamics approximating those of the real system as closely as possible, with respect to the hardware under test (HUT). The accuracy of these dynamics depends both on the mathematical model of the simulation and on the latency of the simulation's response to changes in the system state. The simulation response latency can have many sources such as wire propagation delay, A/D and D/A interface delay, device drivers, computer load on a multiuser or general purpose system, user interface management, control programs for auxiliary hardware, task switching latency, and computational complexity of the simulation model itself. While the implementation of the simulation model by itself dominates the processor power consumed by the simulation, an efficient simulation framework with real-time bounded internal delays is required to provide a predictable and minimal external latency.

Minimizing the amount of time the HIL simulation spends communicating with other processes, and thus minimizing operating system interference with the simulation task, allows the simulation to have a real time response in the average case. If the simulation processes are assigned to particular processors in a multiprocessor shared memory system, the simulation can be guaranteed to reflect an input change onto its outputs in bounded time, and can then be considered a hard real time system.

A characteristic feature of the surveyed HIL simulation systems, with the exception of RT-VTB [2], is that the computer-based simulation component, whether it is programmed to simulate the hardware under test or the rest of the system, is built upon a commercial, proprietary simulation platform such as RTDS [3]. This dependency on a proprietary software platform presents a barrier for independent scientists to reproduce the results that have been reported in those papers and causes the simulation software itself to depend on the commercial platform product continuing to exist and be supported by its sponsor, limiting the useful life of the otherwise independent simulation software. This may not be such a problem because of the typically quick obsolescence cycle of the hardware developed with the aid of that simulation, but it presents an obstacle when a research team wishes to revive and build upon the work of earlier researchers in developing a similar product rather than starting completely from scratch.

On the surface, it appears possible to construct an efficient, open real-time simulation framework that runs on a general purpose operating system and on the general purpose computing hardware available in the FACTS Interaction Laboratory. Such a framework would permit HIL experiments to be carried out on generic computer systems, reducing the cost of instrumenting and running the experiments while at the same time increasing access to independent reproduction of experimental results.

In this thesis, a HIL simulation framework consisting of a real time simulation of an electric power transmission system coupled with several external hardware control processes has been constructed. The framework implements a graphical user interface using Unix-style inter-process communication and shared memory techniques, as well as vendor-supplied proprietary real-time extensions to the Linux kernel to allow shielding processors from hardware interrupts. The framework should be easily adaptable to any Unix-like system which has support for System V IPC, support for process pinning, a sufficient number of processor cores, and in which CPUs can be selectively shielded from hardware interrupts. Unlike most previous efforts, this thesis describes a simulation framework that meets the open source definition, and can thus be used for any purpose with no restriction [4]. This framework allows for greater access to HIL simulation techniques by enabling HIL simulation to be performed on easily-accessible general purpose operating systems and computing hardware.

2 REVIEW OF LITERATURE

In order for the result of the HIL simulation to be useful, the results of the simulation must be sufficiently accurate to apply to the real world. HIL simulations involve an inevitable tradeoff between accuracy and real-time response. While it is given that inaccuracies in the simulated model will produce uncertain results, metrics have recently been developed for gauging the actual robustness of the HIL simulation results in terms of uncertainties in the model [1]. In the FACTS Interaction Laboratory, the HIL simulation is used strictly to validate an external prototype of a FACTS device, so error in the simulation has a more limited scope of effect.

The choice of integration time step in the HIL simulation algorithm is important for similar reasons. An excessively long integration time step introduces inaccuracy into the results as system dynamics are omitted, but an excessively short integration time step means less simulated time can be computed in a given block of real time. If the integration time step is shorter than the longest integration time step that would provide sufficient accuracy in the result, real-time response could be needlessly sacrificed.

With respect to the accuracy of the HIL simulation, the sampling method of the HIL interface is an important consideration. If the Nyquist rate is not observed, the simulation will miss external dynamics occurring at a higher frequency, and the result will be rendered inaccurate. However, rapidly polling for external events can consume processor time and potentially invoke peripheral driver code that is not real-time safe, leaving the simulation with not enough processor cycles to run in real time. One approach to this issue is to offload the real world variable sampling to external hardware that averages the variable's state over a given time period (e.g., the simulation time step); the simulation then receives the averaged value as an input for each time step [5][6]. The approach of this framework is simply to sample the real world values asynchronously so that they can be sampled as frequently as necessary. A 1ms sampling rate would provide the simulation with sufficient dynamics with respect to the 300 Hz filter on the FACTS [7].

Several methods have been developed for interfacing the simulated (software-based) and hardware-based components of the HIL system. The simplest method is to couple the systems using low-voltage Digital-Analog Converter (D/A) and Analog-Digital Converter (A/D) interfaces [8]. Control signals are sent to the hardware in digital or analog form and the analog state of the hardware's outputs are sampled back into the simulation. A more complex scheme, and one that allows for better validation of the HUT when applied appropriately, is referred to as PHIL (Power-Hardware-In-the-Loop), or as a "Sim-Stim" (Simulation-Stimulation) interface. A PHIL method implies that real electric power is being exchanged at the interface boundary between the simulated system and the HUT, thus simulating as closely as possible the real environment in which the HUT will exist [9][10][11][12][13]. A MIL (Model-In-the-Loop) interface is very similar to PHIL, but instead of the simulation driving amplifiers directly to generate the power and sampling from transducers as in the Sim-Stim interface, in the MIL approach an external conversion black-box is implemented that converts the A/D and D/A signals on the simulation side to the real power flow on the device side using voltage-source or current-source converters [14]. One unique approach to the real-time HIL interface question is to implement the interface across a USB (Universal Serial Bus) bus using the isochronous transfer mode of USB, which provides for real-time bounded transfers across the interface [15]. The interface used in the FACTS Interaction Laboratory most closely resembles the MIL approach because low-voltage A/D and D/A interfaces are utilized between the simulation and the HUT.

In the context of power systems, HIL has been used in the simulation of power systems where the goal is to analyze failure scenarios of components such as controllers and turbines [16]. It has also been proposed as a method for determining optimal control parameters for power system components such as STATCOM load banks [17]. The HIL technique has also been proposed as a way to refine the parameters of existing systems, so there are potential applications for the HIL simulation technique beyond new system design [17]. The HIL simulation in this work is used to validate the performance of the FACTS device, part of a distributed power grid control system.

Previous work was done in creating an open source, freely redistributable HIL platform called RT-VTB (Real-Time Virtual Test Bed) [2][18][6][19][20][21]. The approach of RT-VTB is to use RTAI, a RTLinux-derivative real-time scheduling extension to the Linux kernel, along with Comedi, a library of A/D and D/A board software drivers that includes real-time extensions. The authors of RT-VTB built on the existing work called VTB (Virtual Test Bed). VTB is a C++ framework that implements several types of solvers, such as simulated analog computer (SAC) and signal extension resistive coupling (SRC). The solver and simulated system parameters can be configured through the creation of a ".vts" file. The approach of RT-VTB is to implement a real-time process in the kernel side of RTAI that sends a periodic "tick" to the simulation userspace process, which in turn polls the kernel process using the RTLinux FIFO mechanism. One tick of the simulation userspace process updates the real world variables from A/D sampling, computes the next time step, and outputs any control signals via D/A. Because Comedi is a real-time library, and because one computational time step of the VTB simulation userspace process always runs much faster than real-time, a hard real-time response is guaranteed in the RT-VTB implementation. In contrast, the FIL power system simulation only makes a soft real-time guarantee, due both to the general purpose operating system requirement and the nature of solving a non-linear system such as a power system.

Existing HIL simulations such as VTB have implemented a Graphical User Interface (GUI), either for visualizing the design of the simulation model as in [22], or in the modification of the control parameters of a running simulation as in [23]. Similarly, a GUI was implemented for control and monitoring of the power system simulation and power generator.

3 APPROACH OF THIS THESIS

The approach of this thesis is to enable a soft-real-time HIL simulation to be built by first relaxing the real-time constraints on the simulation algorithm itself. It cannot always be guaranteed that a non-linear system with an arbitrary set of values for the system's real world variables will have a bounded number of steps to convergence with an iterative solver such as Newton-Raphson (NR) [24]. This framework was designed for implementation of a power system simulation, which is modeled as a system of non-linear equations and is solved using NR. If a non-linear system such as a power system were specified to run in hard real-time, it would not necessarily be useful because whether the constraint is met or not depends on the inputs to the system. Once the power system has been allowed to converge to a relatively steady state, convergence of future time steps will be very fast (2 or 3 iterations); small changes in the real world values will not impact this fast convergence. However, a power system solver under a hard real-time deadline may produce no results at all, since if it is not given enough real time to converge to a solution in a given time step, it may never have enough time to converge to a new steady state solution on any future time step.

It has been shown that only a fixed integration time step can satisfy hard real time constraints, and thus a variable step should be only cautiously employed [24]. Methods of accounting for multiple switching events during a timestep and recalculating the step appropriately also have been developed, but the computational expense may preclude usage in a real-time environment because the usual approach is to restart the current timestep, switch to a variable timestep to solve until the point where the events occur, and then switch back to the nominal time step when the simulation has caught up. One way to reduce the computational expense is by employing linear interpolation between the states preceding and following the event in question [25]. A more refined approach is to combine a variable timestep with interpolation to arrive at a more accurate result, although at higher computational expense [26]. The FIL power system simulation uses a fixed time step simulation and omits any attempt to account for missed external dynamics, since any interesting external dynamics in the system occur at a relatively slow rate.

Regarding the simulation itself, it might be tempting to implement the simulation as a single, normal process. After all, if the program is in ready-to-run state most of the time and the system is not loaded with other tasks, normal operating system scheduling should provide a reasonable response time. It would then be up to the user to prevent the computer from becoming loaded with other tasks while the simulation is running. This approach unfortunately cannot offer any sort of real-time guarantee because a general purpose operating system's scheduler is real-time nondeterministic; there is no way to predict how a scheduler will behave without auditing and certifying a particular scheduler for real-time response. System calls can also have quite unpredictable blocking characteristics. A single-process design would also require that all hardware I/O and user interface management be performed synchronously. Depending on the real-time capability of the hardware drivers, this may not be feasible. Additionally, a synchronous user interface is not good practice from the perspective of human-computer interaction.

Another approach, as in the approach of RT-VTB, is to implement the simulation as a real-time process under a certified real-time operating system (RTOS). Several problems with this approach are a lack of certified real-time device drivers for peripheral hardware as well as a general difficulty of coordinating hardware controllers and user interface through a RTOS. It can be argued that a certified real-time operating system is an unacceptable requirement for a general purpose real-time simulation framework, because it limits the availability of the framework to only those hardware platforms for which a certified real-time operating system is available. Due to the generally poor portability of code between real-time operating systems, it would also leave all subsequent researchers with little choice but to utilize the same real-time operating system. For these reasons, the idea of implementing the simulation under an open source hard RTOS like RTLinux [27], RTAI [28], or KURT-Linux [29] was rejected.

A multi-process design would also have to endure several challenges, not the least of which is that the several processes would have to coordinate access to shared data and avoid race conditions through the use of locks, which are synchronization primitives provided by the operating system. Misuse of locks or the accidental introduction of a lock ordering bug could lead to seemingly-random deadlock – a partial or total halt of the system. Misuse of locks could also foil the real-time performance of the system by stalling an important process for longer than otherwise necessary.

The decision was made to use a multi-process design requiring at least N+1 processor cores in the system, where N is the number of concurrently running processes that hold a lock. Those lock-holding processes are pinned to distinct CPUs so that no other process can be scheduled on the CPU that a lock-holding process owns. Along with shielding critical processors from hardware interrupts and disallowing calls to non-realtime-safe code inside critical sections, this scheme guarantees that the time that any process holds the lock is bounded, ensuring that the simulation process time step – which takes a lock itself in order to read the asynchronously updated values of its real world variables – is bounded. The framework described in this thesis utilizes a shared memory architecture, where the

A/D is sampled and the D/A state is updated in a process that runs concurrently with the simulation, and where that process shares the simulation state memory as in the approach of [15].

4 OVERVIEW OF HIL SIMULATION FRAMEWORK

In this work, a framework has been produced that enables soft real-time simulation with no specialized hardware support and no specialized operating system support for real-time scheduling, meaning it can be run on an off-the-shelf Unix system such as Linux as long as it complies with the respective standards. The simulation algorithm itself has no requirement beyond the fact that it must run fast enough by itself on the given system's CPU and memory architecture to satisfy the real time constraints of the system. The framework imposes few additional constraints on the hardware and operating system beyond those imposed by the simulation core.

4.1 REQUIREMENTS

For any type of real-time computation, the driving goal is to maximize the proportion of time that the computation process is in "running" or "ready to run" state compared to the time that it is in other states, such as "waiting on hardware" or "waiting on lock". Thus, the primary goals of this framework were to decouple the management of A/D and D/A hardware from the simulation process itself and to use inter-process communication techniques that cause minimal blocking in the simulation process.

4.2 ASYNCHRONOUS HARDWARE MANAGEMENT

Unix drivers typically include a userspace library that abstracts the kernel driver interface and which is called directly by the application wishing to use the hardware instead of the application calling the kernel interface directly. The API of the library thus insulates the application from changes in the implementation of the driver, which can be quite volatile.

The goal of decoupling the management of A/D and D/A hardware from the simulation process is to eliminate the overhead caused by peripheral device I/O. This overhead comes from calling the userspace library, transitioning to kernel mode, and communicating with the peripheral through its registers. Sometimes the kernel mode transition can be eliminated if the device supports memory mapped I/O, but in all cases the sequence of device communication must be repeated every time analog data is to be captured or sent. Placing the burden of hardware communication on the simulation process is unreasonable in two ways. First, it adds to the baseline latency of a simulation time step due to the layers of driver code described above, ensuring that one iteration of the simulation loop can never be completed in less real time than the hardware I/O requires. It also requires making a difficult decision about the location of the code performing the analog data updates relative to the code implementing the system solver (NR) loop. If this code is placed outside the NR loop, the most recent updates to the system variables are unavailable to the solver until a time step has passed, and a latency between the solver updating the system state and the real world outputs reflecting the updated state occurs. However, placing the update code inside the NR loop requires more indirection of memory access, slowing the solver; the complexity of the inner solver code itself is also increased, potentially causing instruction cache misses. Decoupling the A/D and D/A and having a separate process asynchronously merge the simulated system state with the real world state avoids these problems and allows as fine a granularity of A/D and D/A sweeps as the application requires (Figure 4.1).

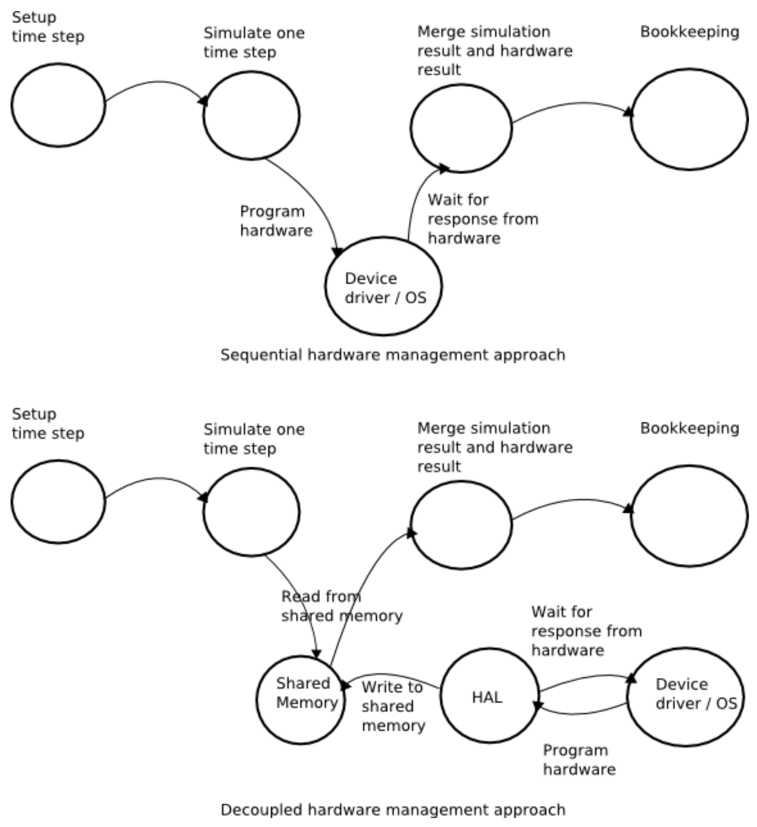

Setup time step
Simulate one time step
Merge simulation result and hardware result
Bookkeeping
Program hardware
Wait for response from hardware
Device driver / OS
Sequential hardware management approach
Setup time step
Simulate one time step
Merge simulation result and hardware result
Bookkeeping
Read from shared memory
Wait for response from hardware
Shared Memory
Write to shared memory
HAL
Device driver / OS
Program hardware
Decoupled hardware management approach

Figure 4.1: Comparison of hardware approaches.

For the remainder of this thesis, interface variables refer to the variables that are sampled externally from A/D hardware and then injected back into the system.

4.3 INTERPROCESS COMMUNICATION ALTERNATIVES

In contrast to the Windows platform, a Unix environment implements a diverse, although not always orthogonal, set of inter-process communication (IPC) primitives for message passing and synchronization. Available mechanisms are signals, pipes, sockets, message queues, semaphores (POSIX and SysV implementations), and shared memory (either shared pages external to the process as in POSIX shared memory or a shared program area and heap memory as in POSIX threads).

Many of these IPC primitives have conditions under which they block (suspend) the process utilizing them. This would be unacceptable for the simulation process since it is under a real-time constraint. For example, attempting to receive a message in a currently empty queue, or attemping to read a pipe or a socket which currently has no waiting data will block the calling process. Some system calls such as read() have non-blocking variants. A non-blocking system call is guaranteed to not block the process; non-blocking Unix system calls return EAGAIN ("try again") as an error if the call would have otherwise been blocked. This approach has the problem of requiring the application to periodically poll for changes in the status of the pipe or socket. This polling incurs a performance penalty above and beyond a normal timed-interval wait because each poll also costs the overhead of a system call. A Linux system call flushes the CPU pipeline since system calls are implemented with a designated software interrupt. Additionally, the entire CPU register set must be saved to memory before kernel code is executed, and before returning from the system call, the registers must be reloaded with the application's CPU context. The biggest problem of all is that the read() system call is not real-time safe on a general purpose operating system.

Some sort of shared memory approach was thus required, so that IPC could occur without a system call. A thread-based approach was rejected because one of the system requirements was that the components of the simulation needed some sort of process prioritization, or at the very least CPU affinity, so that the scheduling of the core simulation task was not potentially affected by the user interface, peripheral controllers, or other components. A method of manually controlling thread scheduling in the required fashion did not appear to be available. An obvious approach that remained was to use the combination of System V semaphores for synchronization and SysV/POSIX shared memory for message passing.

It should be noted that the choice of using shared memory provides no inherent synchronization or serialization of memory read and write operations, so special care must be taken to avoid the case where a reader reads a piece of data which has been only partially written to the shared memory region. This additional problem, caused by the choice of shared memory as the message passing medium, can be addressed by the use of System V semaphores.[1]

The most general usage of a semaphore is to implement Dijkstra's P() (hold) and V() (release) operations [30]. The framework abstracts the relatively complex and obscure C code required for holding and releasing a System V semaphore into relatively simple DOWN() and UP() macros that are used throughout the codebase. These macros can then be used whenever it is necessary for a system component to wait for exclusive access to some shared resource, such as when a system component needs to update a data object in shared memory and the update cannot be performed atomically. This is a common situation since usually a processor can only atomically update data types that it supports in hardware – such as integers up to and including the word size of the processor as well as 32-bit and 64-bit floating point values – and many objects in real-world systems are sets of such types and thus cannot be atomically updated as a set.

Since it would be best to avoid at all costs the event where the process bearing the computational load is rescheduled or even blocked at any point, the framework must ensure that system calls which could block the process are kept to a minimum, if not eliminated. (It is not possible to create an exhaustive list of all blocking system calls to avoid because many system calls specified in standards have unspecified blocking behavior and are implemented in differing fashions depending on the operating system's design.) I/O involving device drivers would usually block due to hardware response latency and recovery times, but this is not a problem since device I/O is performed asynchronously in this framework. Unfortunately, in order to utilize System V semaphores there is no choice but to use the semop() system call. Due to the nature of semaphores the semop() call will be blocked if the semaphore is already held. It is possible to specify the IPC_NOWAIT flag to avoid blocking, but this would allow the process to proceed without acquiring the semaphore and without reading the updated data. If the timing between processes ends in the worst case, the computational process may never be able to read updated data captured from the hardware, and thus the simulated state would eventually diverge from the real state sufficiently to invalidate the result of the simulation.

This problem was solved by utilizing a set of "new data" flags, in an approach similar to the general double-checked locking approach [31]. To be consistent with the implemented framework code, these flags will hereafter be referred to as "stale" flags. (A "stale" flag can simply be regarded as an active-low "new data" flag.) Each stale flag X_f for

[1]System V semaphores are used by allocating a semaphore set which is accounted for externally to the process, with a per-user quota. When the processes using the semaphores are destroyed, the semaphore set still exists in the operating system, so the simulation must manage its semaphores to avoid leaving behind orphans.

interface variable X exists as an atomically-updatable data type and corresponds to the data item or set that is protected by a lock P(X). When that data set is updated by a writer holding the corresponding semaphore, the stale flag is also reset before releasing the semaphore. An interested reader can check the stale flag before attempting to take the lock P(X). If the stale flag X_f is set, there is no reason to take the lock P(X), since the data set X has not yet been updated since the last time it was read, and in this case a system call is avoided. If the stale flag X_f is reset (not set), then the computational process takes the lock P(X), reads the interface variable X, sets the stale flag X_f, and finally releases the lock P(X) (Figure 4.2). Note that the operating system is only called when lock P(X) is taken or released, so avoiding unnecessary locking in this fashion is an important real-time advantage.

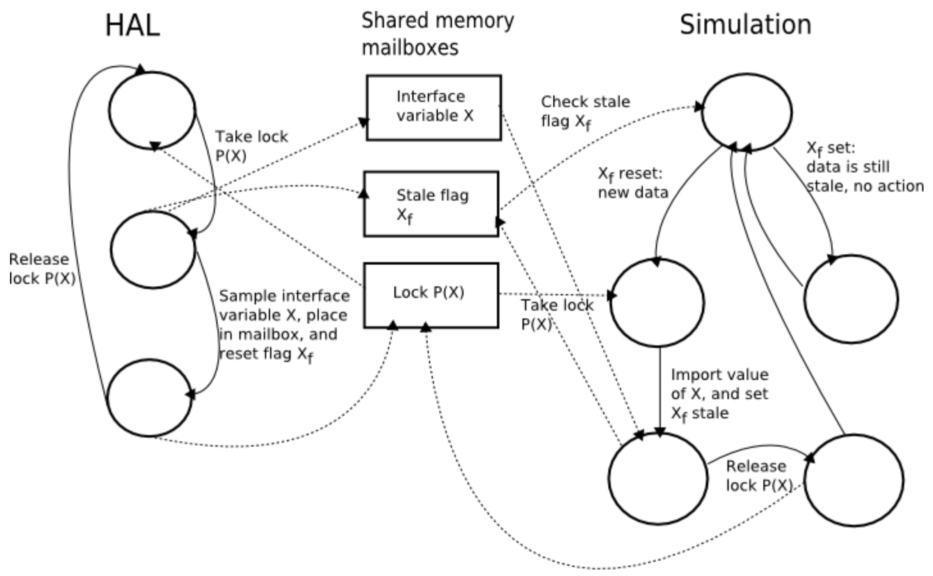


Figure 4.2: The locking scheme, with stale flags.

Once the semaphore operations were implemented, the semaphore set was placed in shared memory. It should be noted that this is a cooperative resource sharing scheme. It is possible for a rogue process to take and hold any lock indefinitely since there is no form of partitioning or protection of the shared memory region. It is also possible for a rogue process to ignore the locks completely when reading or writing. If locks are ignored when writing or reading, this will lead to the interested reader reading an inconsistent object. Thus, all access to the shared resource must be coordinated by semaphore. It should also be observed from basic principles that the stale flag scheme is by itself not a sufficient mutual exclusion mechanism. It is simply a way to avoid blocking the computational process unnecessarily to check for new data at times when the data in the mailbox has clearly not been updated since the last check.

In this thesis, individual slots in shared memory will be referred to as "mailboxes" (Figure 5.1).[2] The slot size should be defined as the largest atomic data type that the platform allows; in the case of the FIL hardware, it is a double-width floating point value with 64 bits of storage. Data items of non-floating point type can be written to the slot by obtaining a pointer to the slot's memory location and casting the pointer to the desired type before performing the write and any subsequent reads from that slot.

[2]Use of the term 'mailbox' in this implementation should not be confused with the more common IPC-related usage that refers to a message queue that is allowed to grow to any size.

5 DISCUSSION OF HIL SIMULATION FRAMEWORK

The simulation framework contains many components. Each component of the framework along with some details about its implementation will be described. Discussion about the implementation of the simulation itself will be limited to those aspects which affect its integration into the framework; the simulation's algorithmic implementation will remain a "black box" to assure the applicability of the framework to any simulation satisfying the requirements.

5.1 SYSTEM BLOCK DIAGRAM

In Figure 5.1, the framework is decomposed into several principal components. These include the Simulation Engine which is comprised of the PI (Proportional-Integral) controller that controls part of the HIL line, a Simulation Core that contains the numeric solver, a Load Bank Controller that controls the load of the HIL line, and a Visualization and Long Term Control (Max Flow) sender. These interact with the Hardware Abstraction Layer (HAL) through Shared Memory regions. Companion work has formally specified the relationship of each component to the system for the purposes of hardware/software co-design [32][33].

In this system, the HAL and the simulation process are considered critical processes, so at least three CPU cores are necessary to ensure that the soft real-time constraint is met.

5.2 SYSTEM HARDWARE/SOFTWARE

This testing was carried out on a 4-processor Concurrent Computer iHawk rack-mountable system. The CPUs in this system are Intel Xeon 2.2GHz with 512KB L2 cache and 2MB L3 cache. The CPUs are a P4 Netburst architecture and have hyperthreading enabled to enable two logical cores per CPU. Real-time scheduling is achieved by pinning processes to a particular CPU with the sched_setaffinity(2) Linux system call, and then routing hardware interrupts away from processors that are running critical processes using Concurrent's proprietary shield(1) command. The operating system is RedHawk Linux 2.2, which is essentially Red Hat Enterprise Linux 3.0 with a Linux 2.6.6 kernel and some Concurrent-authored realtime support patches (for use with their NightStar tools). The system has 4GB RAM and a 74GB SCSI hard disk. The D/A hardware is composed of two General Standards Corporation GS-16AO12 boards which are supported by the Comedi driver suite with the 'ni_pcimio' driver, and the A/D hardware is a General Standards Corporation 16AI64LL-2 with 64 input channels supported by the 'ai64ll' driver from GSC.

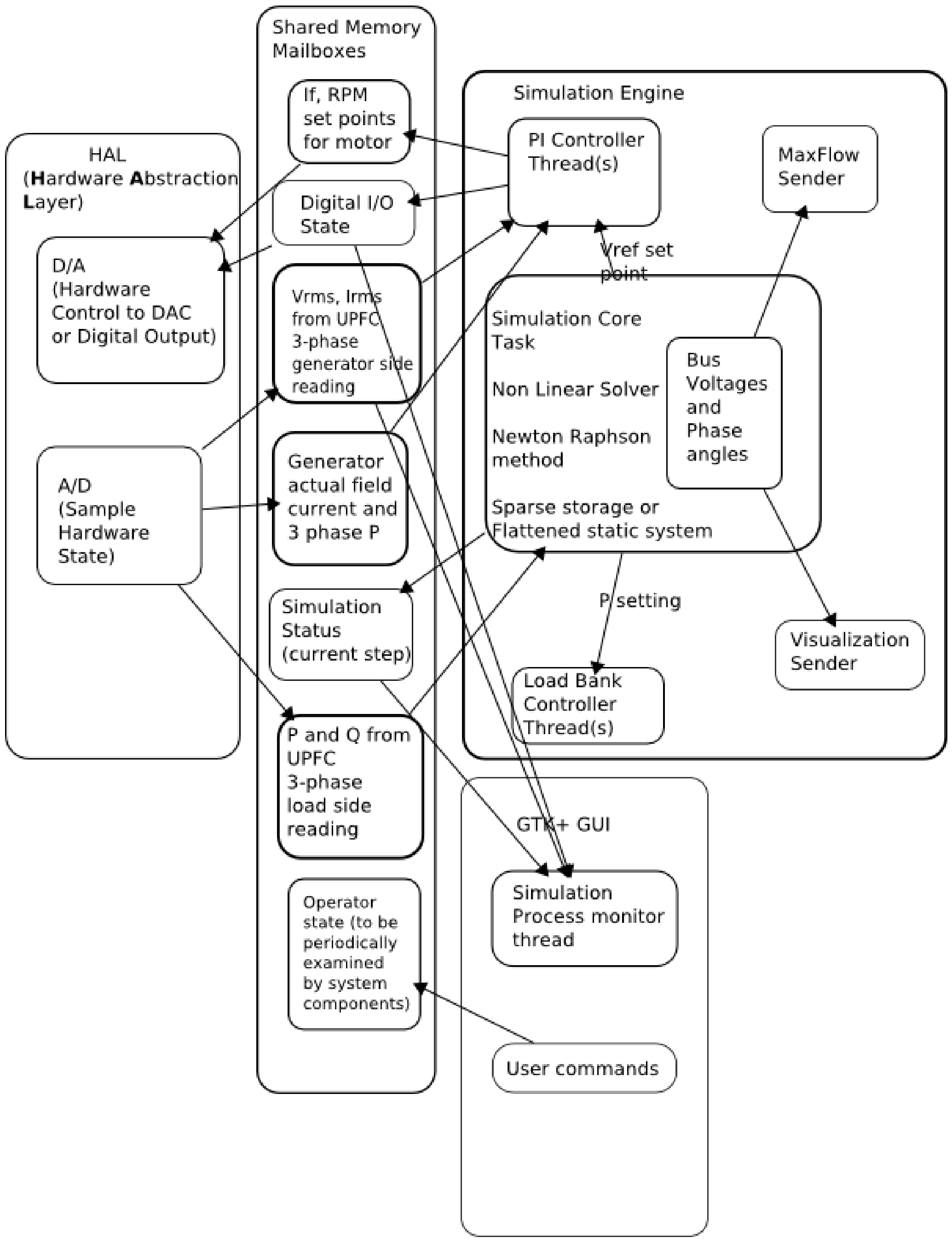

Shared Memory Mailboxes
If, RPM set points for motor
Digital I/O State
Vrms, Irms from UPFC 3-phase generator side reading
Generator actual field current and 3 phase P
Simulation Status (current step)
P and Q from UPFC 3-phase load side reading
Operator state (to be periodically examined by system components)
HAL (Hardware Abstraction Layer)
D/A (Hardware Control to DAC or Digital Output)
A/D (Sample Hardware State)
Simulation Engine
PI Controller Thread(s)
MaxFlow Sender
Vref set point
Simulation Core Task
Non Linear Solver
Newton Raphson method
Sparse storage or Flattened static system
Bus Voltages and Phase angles
P setting
Load Bank Controller Thread(s)
Visualization Sender
GTK+ GUI
Simulation Process monitor thread
User commands

Figure 5.1: High level diagram of the simulation system.

The D/A and A/D hardware is connected to the power generation and control hardware with a custom interface board.

The Flexible AC Transmission System (FACTS) consists of several devices, including power sensors, a Unified Power Flow Controller (UPFC), a DSP, and an embedded computer (Figure 5.2, Figure 5.3, Figure 5.4) [34].

The embedded computer implements a Long-Term Control algorithm that controls the state of the UPFC. The embedded computer communicates with the simulation computer via ethernet and receives operator input from the keyboard. The UPFC is implemented as a pair of compensators that together control real and reactive power flow on the transmission line.

5.3 SOFTWARE VERSIONS

The software versions used in the final experiments were:

Linux Kernel: version 2.6.6 of the RedHawk Linux kernel compiled with a custom configuration to add parallel port drivers (Section 5.8)

Linux Distribution: Concurrent's RedHawk Linux distribution, roughly equivalent to Update 8 of Red Hat Enterprise Linux Workstation version 3.0

GNU C Library: 2.3.2

GNU C/C++ Compiler: 4.1.1

Intel C/C++ Compiler: 9.1

Comedi: version 0.7.22 of libcomedi and the Comedi 'ni_pcimio' D/A driver

General Standards Corporation A/D Driver: version 1.30 of the 'ai64ll' driver

GTK+ Toolkit: 1.2.10

libieee1284: 0.2.10

shield: version 1.0.1 authored by Concurrent Computer

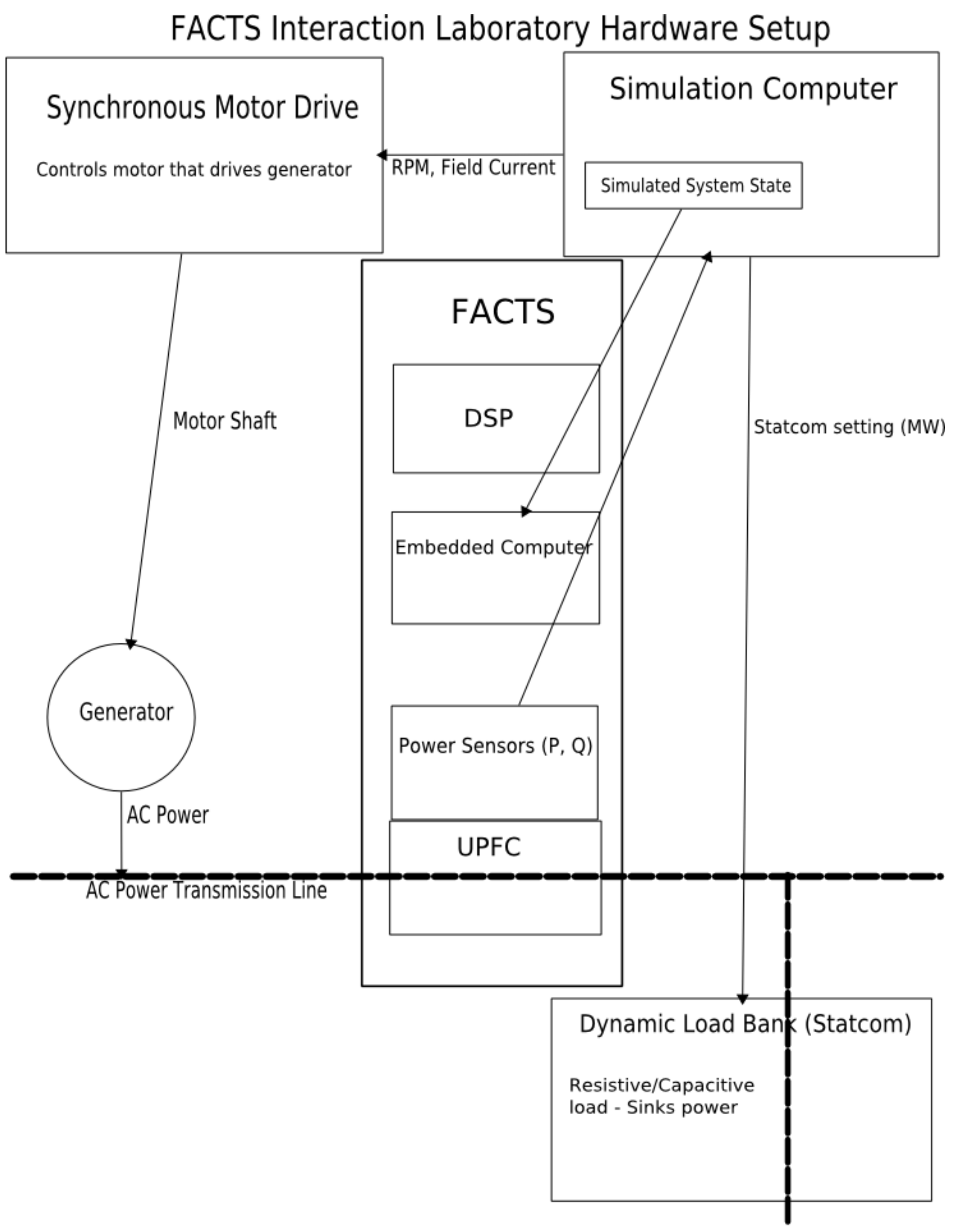


Figure 5.2: The FIL laboratory setup.

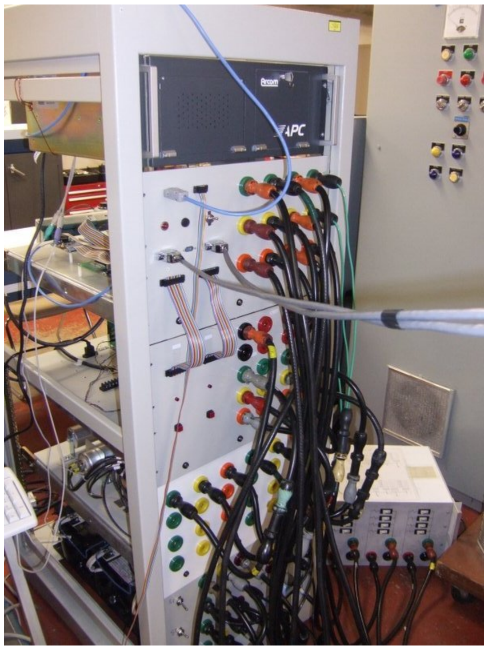


Figure 5.3: A FACTS device with embedded computer.

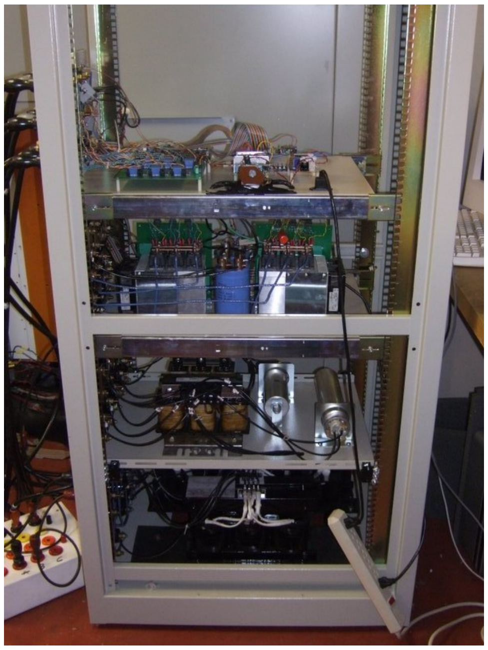


Figure 5.4: Internals of the FACTS device.

5.4 HAL (HARDWARE ABSTRACTION LAYER)

The HAL was developed in response to the requirement that the external hardware state be asynchronously updated, since the simulation engine needs to spend as much time as possible actually performing computations related to retiring a time step in order to reliably meet the real time deadline. HAL runs as a separate process, with automata as depicted in Figure 5.5, and acts as a concentrator for all physical I/O.

Its main loop runs as often as the application requires. On a SMP (Symmetric Multi-Processor) system such as in the FIL lab, one processor can be dedicated to running the HAL process. In this case, the main loop can run continuously and update the hardware state as quickly as the hardware allows because the CPU does not have to yield to other processes such as the simulation process or user interface.

The HAL main loop simply does the following in pseudocode:

```
SweepDtoA();
CollectAndConvertAD();
usleep(DELAY);
```

The function of SweepDtoA is to take the values that are in the mailboxes which correspond to what the current state of the DAC and Digital I/O outputs should be, and write out the entire state to the hardware. The state of the real hardware should thus consistently follow the state of these variables in the mailbox .

CollectAndConvertAD is a collection of steps taken to process the analog readings into an interface variable in the system. Several variables, such as voltage and current readings, are filtered through a first-order low-pass filter (LPF) to reduce sampling noise caused by sensor inaccuracy. The scale factors that are used to scale a voltage level to a real value to be placed in an interface variable are dependent both on the particular data item and the particular sensor that was used to gather its value, and thus are experimentally determined and then encoded as magic constants. The flow of data for one interface variable is depicted in Figure 5.6.

The sleep interval is not required unless the system cannot dedicate a CPU to the HAL process. In this case, sleeping for only 1 microsecond would not have much of a delay effect; its actual purpose is to ensure that the HAL process yields the CPU at this point so that another process is allowed to run.

5.5 SIMULATION CORE (FLATTEN4)

The simulation core is a library that implements the solution of the non-linear power system at a particular time step. The simulation core takes the physical connections, bus

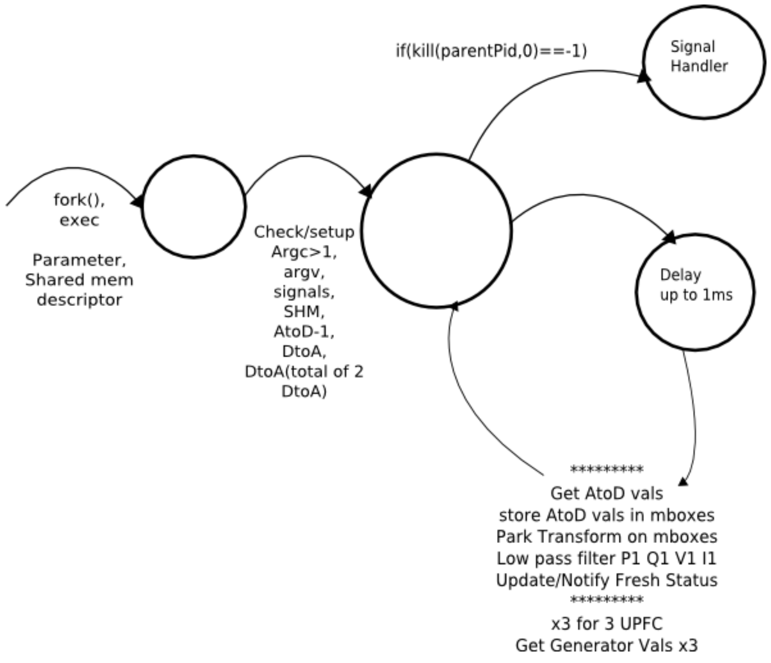

if(kill(parentPid,0)==-1)
Signal Handler
fork(), exec
Parameter, Shared mem descriptor
Check/setup Argc>1, argv, signals, SHM, AtoD-1, DtoA, DtoA(total of 2 DtoA)
Delay up to 1ms
****
Get AtoD vals
store AtoD vals in mboxes
Park Transform on mboxes
Low pass filter P1 Q1 V1 I1
Update/Notify Fresh Status
****
x3 for 3 UPFC
Get Generator Vals x3

Figure 5.5: Automata of the HAL.

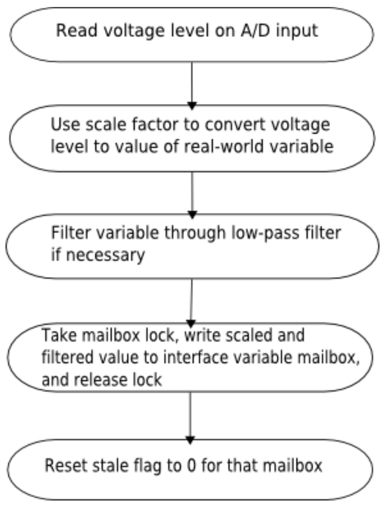

Read voltage level on A/D input
Use scale factor to convert voltage level to value of real-world variable
Filter variable through low-pass filter if necessary
Take mailbox lock, write scaled and filtered value to interface variable mailbox, and release lock
Reset stale flag to 0 for that mailbox

Figure 5.6: Data flow for an interface variable being sampled into the system.

voltages, phase angles, power generation, and admittances of the power system as inputs, and calculates the new bus voltages and phase angles using a non-linear system solver, specifically NR [35, pp 52-69]. The NR method involves repeatedly updating the Jacobian matrix that describes the partial derivatives of the state variables with respect to each other and directly solving the resulting linear system, until the "mismatch" variables (ΔP and ΔQ) are driven close enough to zero. This should take two to three iterations in the average case, but may take many more depending on how far away the system is from the steady state.

The first implementation of the simulation core was in C++. It was too slow to run in real time, even independent of the rest of the system. There were several aspects of this approach that hindered performance. The LU decomposition was performed dynamically each time the Jacobian was updated (several times within each NR solution) and access to the system matrices was performed dynamically. If the system matrices were stored as normal dense matrices, the only cost was wasted memory space and a larger cache utilization. On the other hand, if the system matrices were stored as sparse matrices, indexing became a much more complex operation. Using sparse matrices allows computational savings such as avoiding useless multiplications by zero in the LU factorization step. However, the number of matrix indexing operations alone made using sparse matrices infeasible if the program were to run in real time.

A new approach was developed to symbolically perform the LU decomposition at compile time, avoiding the expensive indexing and multiplications associated with the LU decomposition. This was accomplished by writing a MATLAB program that generated the C source code files comprising the solver [36]. Memory accesses performed by the C program were "flattened" so that memory was accessed through a single pointer for each matrix instead of through multiple levels of indirection. It was also found that using the Intel C compiler in place of the GNU C compiler improved performance of the "flattened" simulation core by roughly 20-30% in the experimental system. Experimentally, it was found that this solver performs the evaluation of the Jacobian and the NR solution of the system in real-time for all steady state time steps when the time step is defined as 1ms. When the system encounters a contingency such as the removal of a line, a change in generator voltage, or a change in power flow through the FACTS device under test, it initially violates the real time constraint, but then catches up within several time steps.

It is important to note that the simulation core library was written in a modular fashion that does not prevent it from being adapted to other uses beyond HIL experiments. Accordingly, it is only loosely coupled to the simulation framework described in this thesis.

5.6 SIMULATION CORE TASK (SIM_DRIVER)

This program wraps the simulation core and takes care of minimal locking to update the simulated system state from the sampled hardware state. Its automata is described in Figure 5.7.

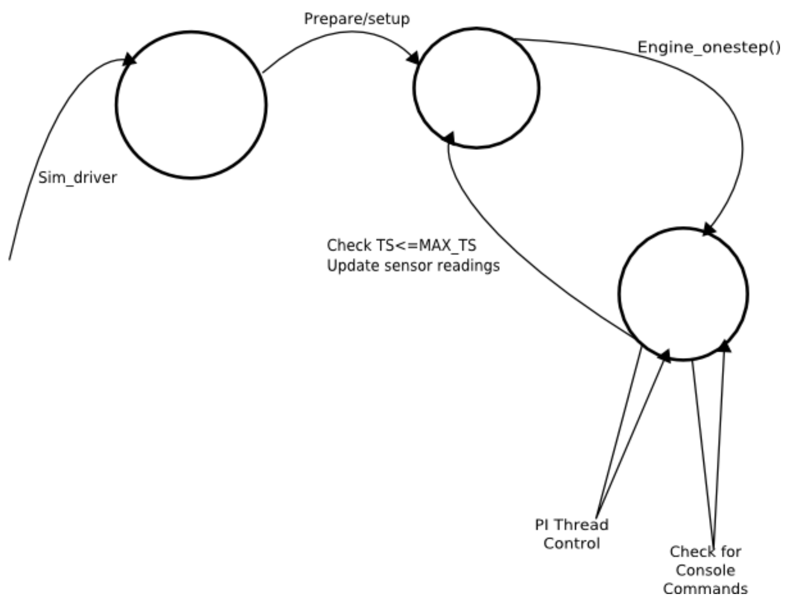


Figure 5.7: Simulation driver automata.

The simulation task also updates the state of the controller threads for the external hardware and processes such as the generator and loadbank controllers. It either takes the shared memory region passed to it by the controller process, or sets up its own shared memory region. It then spawns the HAL process and passes it the handle for the shared memory region.

5.7 PI (PROPORTIONAL-INTEGRAL) CONTROLLER(S)

The PI controller program is run as a thread within the simulation process. Up to three PI controller threads can be created, one for each of the three FACTS devices in the FIL power system. The PI controller thread is in charge of controlling the motor hardware and ensuring that it is left in a safe condition and that all physical constraints are met. It also controls the output voltage of the generator by varying the field current (I_f). [3]

The PI controller uses the mailbox system like any other simulation component. It sets the RPM and field current through the mailbox interface for the D/A card and reads back the output voltage and field current through the mailbox interface for the A/D card. Separate mailboxes are supplied for up to three PI controllers.

The PI controller's only input is a voltage set point for the bus to which the FACTS generator side is connected. This set point is determined by the output of the simulation for that time step. When the set point is updated, the PI algorithm steps the output field current to approach the new V set point asymptotically. The rate of stepping is controlled by the constants K_i and K_p. The further away the present V value is from the set point, the faster the new value will be approached.

If at any point the generator voltage, output current, or field current go out of range, the PI controller shuts down the motor.

5.8 LOAD BANK CONTROLLER

The variable load bank (Figure 5.8) is a programmable load bank that replaced the manually configured static load banks (Figure 5.9). It consists of an array of lead-acid wet cell batteries (Figure 5.10) that are connected in various numbers to the load side of the power system in order to draw power from the system as the batteries accumulate a charge.

The load bank is controlled by some power electronics that are interfaced to two Dalanco model 5000 DSP boards based on TI TMS320c5x DSP. These boards are installed in a standard PC and execute controller algorithms. A program on the PC is responsible for loading the DSP programs and graphing the output of the sensors as the loadbank is controlled[37].

A method of controlling the variable load bank from the simulation in response to the simulation state became necessary. The load bank processor takes an input in megawatts (MW). A modified PC parallel cable connection was implemented between the simulation computer and the load bank processor that can transfer 8 bits at a time and has an interrupt driven handshake. Since the simulation computer has a PCI-X system bus and no built-in parallel port, a PCI-X board was purchased; the board is based on an Oxford Semiconductor OX16PCI952 ASIC which implements a PC-compatible ECP parallel port and dual serial ports. The Pload setting for the FACTS load side bus is written to a mailbox that the

[3]In the FIL experiments, the RPM of the generator motor was fixed since at the time there was not a functional tachometer for feedback; one could also leave the field current constant and vary the speed of the motor to achieve the desired output voltage.

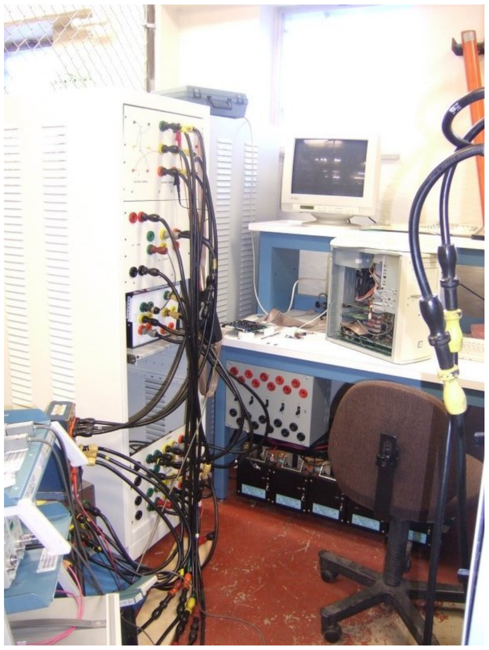


Figure 5.8: The programmable load bank console.

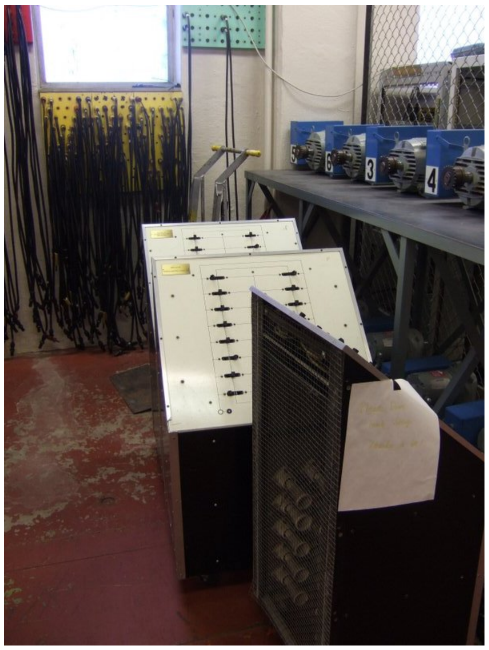


Figure 5.9: The static load banks.

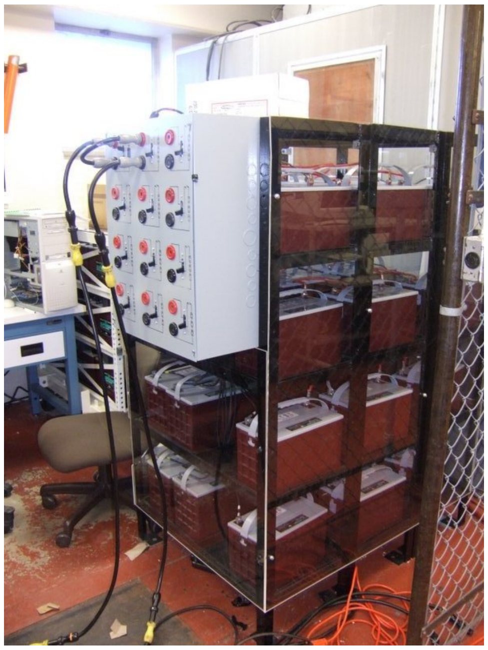


Figure 5.10: The load bank power storage array.

loadbank controller thread picks up. The load bank controller thread encodes the P value into 8 bits and places the encoded value on the parallel cable, then the interrupt line is asserted. On the load bank computer, the interrupt is handled by the MS-DOS-based driver program. The value on the parallel cable is read, and an acknowledgement interrupt is asserted back to the simulation computer, at which point the simulation computer restarts the communication by sending its most current P value. An open source library libieee1284 is utilized as a parallel port abstraction to simplify the code. The Linux kernel was modified to enable interrupt-driven operation for the Oxford port and the patch was submitted to the main kernel developers. A program was also written to program and to dump the Oxford card's onboard serial EEPROM, and to parse the contents of an EEPROM image. It was anticipated that the Oxford card could have its I/O addresses assigned to the standard PC parallel port I/O addresses in order to avoid the Linux kernel modification, but it turned out that such a configuration was not possible with the Oxford hardware. In the end the standard Linux kernel driver for PC parallel ports was used.

5.9 SIMULATION CONSOLE (SIMGUI)

It frequently becomes important for the operator at the simulation console (Figure 5.11) to exercise some sort of control over the simulation once it is running. The simulation architecture was designed so that the simulation could be run using any physical input method or "headless" with no console at all. All user interface communication is

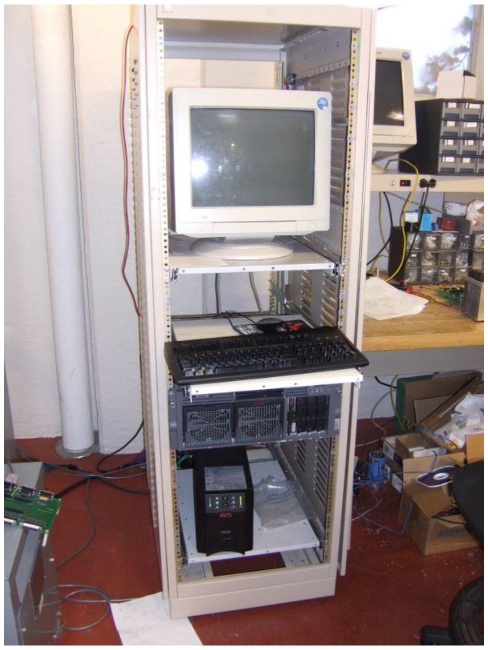
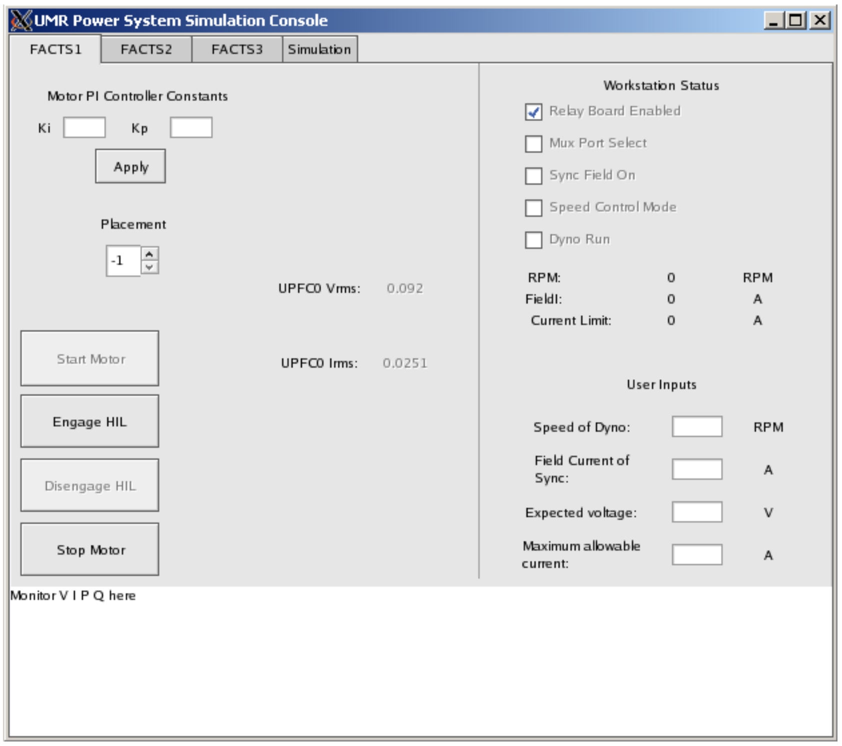


Figure 5.11: The simulation computer.

performed via state variables which are stored in mailboxes assigned for this purpose. The user interface was implemented using the GTK+ toolkit (Figure 5.12), but any toolkit could be utilized to construct any user interface as long as communication with the simulation is performed via shared memory. Since GTK+ has a callback-driven API with a C interface, it was necessary to employ caution when obtaining both GUI locks and shared memory locks to avoid the introduction of lock ordering errors that would cause a difficult-to-debug deadlock.


UMR Power System Simulation Console

FACTS1 | FACTS2 | FACTS3 | Simulation

Motor PI Controller Constants

Ki Kp

Apply

Placement

-1

UPFC0 Vrms: 0.092

UPFC0 Irms: 0.0251

Start Motor

Engage HIL

Disengage HIL

Stop Motor

Workstation Status

[✓] Relay Board Enabled

[ ] Mux Port Select

[ ] Sync Field On

[ ] Speed Control Mode

[ ] Dyno Run

RPM:	0	RPM
FieldI:	0	A
Current Limit:	0	A

User Inputs

Speed of Dyno:		RPM
Field Current of Sync:		A
Expected voltage:		V
Maximum allowable current:		A

Monitor V I P Q here

Figure 5.12: The power system simulation GUI.

5.10 SYSTEM STATE TRANSMITTER (SIMLISTENER)

The system state transmitter component transmits the current values of voltage and phase angles at each bus of the power system in response to incoming network requests. This state data is used by the algorithm in the Long Term Control (LTC) process of the FACTS and by the power system visualization console.

The transmitter is implemented as a thread which shares the system state with the simulation. The thread implements a minimal server which receives incoming connections and then enters a request loop. The request loop only supports a handful of commands, the most important of which is to initiate a state dump, at which point the system's "V" and "theta" variables are captured in a snapshot and then copied across the network. Since this is an infrequent operation, it might be possible for the LTC or the visualization to cause a slowdown in the simulation by repeatedly requesting the state variables, causing an increase in lock contention for the snapshot copy and a possible missed deadline for the simulation.

There are several approaches to providing the snapshot requests, depending on how the snapshot data is to be used. If the snapshot is required to be consistent with respect to the simulation time step, one approach is to limit the snapshot requests to no more than a certain number of requests per time period. The snapshot would have to be first copied into a local buffer before being sent to the network layer for transmission so that the lock is not held for an unreasonable amount of time. If the snapshot does not necessarily have to be consistent with respect to a simulation time step, another approach is possible. Since all of the simulation state data are stored in atomic data types according to the system design, the snapshot mechanism could be implemented so that the simulation simply writes the state data as usual and the state is read "as-is" from the buffer, possibly cutting into the middle of a time step, but requiring no lock at all.

6 RESULTS AND DISCUSSION

Experiments were performed with several different settings for the HAL update delay: 100 microseconds, 50 microseconds, 5 microseconds, and no delay. The time the simulation spent acquiring the lock, as well as the time spent in each critical section, were then examined.

The results in Figures 6.1 through 6.4 demonstrate that the worst case average latency is less than 20 microseconds (less than 5% of a simulation time step). The results show that, even when using a HAL update frequency far greater than that which is minimally required to capture the 300Hz system dynamics [7], the framework has no trouble at all keeping the lock acquisition time quite small.

The results in Figures 6.5 through 6.8 demonstrate that the time spent in the critical section for each interface variable, including releasing the lock, is on average less than 15 microseconds.

In the worst case the average delay caused by the lock acquisition is less than 20 microseconds and the average delay caused by updating each of four interface variables is less than 15 microseconds. Given these results, it is evident that for the power simulation implemented in this framework, as long as the simulation time step can be computed with at least 80 microseconds to spare, the additional locking imposed by the framework will not cause the real time constraint to be violated.

The experiments show that the average time to acquire a lock increases as the HAL update frequency increases. This is because the time the simulation spends acquiring the lock primarily depends on how frequently the HAL takes the lock to update the interface variables. On the other hand, the time spent inside the critical section should remain relatively constant. It is a balancing act between having too many locks, which adds constant overhead even when lock contention is low, and too few locks, which increases lock acquisition latency. When implementing a simulation engine within this framework, careful timing analysis should be conducted to determine how much lock contention is occurring, so that it can be appropriately addressed.

With a power system simulation implemented in this framework, the simulation is able to respond to changes in the power flow through the FACTS device in soft real time with update latency determined by the polling frequency of the HAL component. A system based on this framework has been in use since March 2007 in the FACTS Interaction Laboratory [38] at the University of Missouri-Rolla.

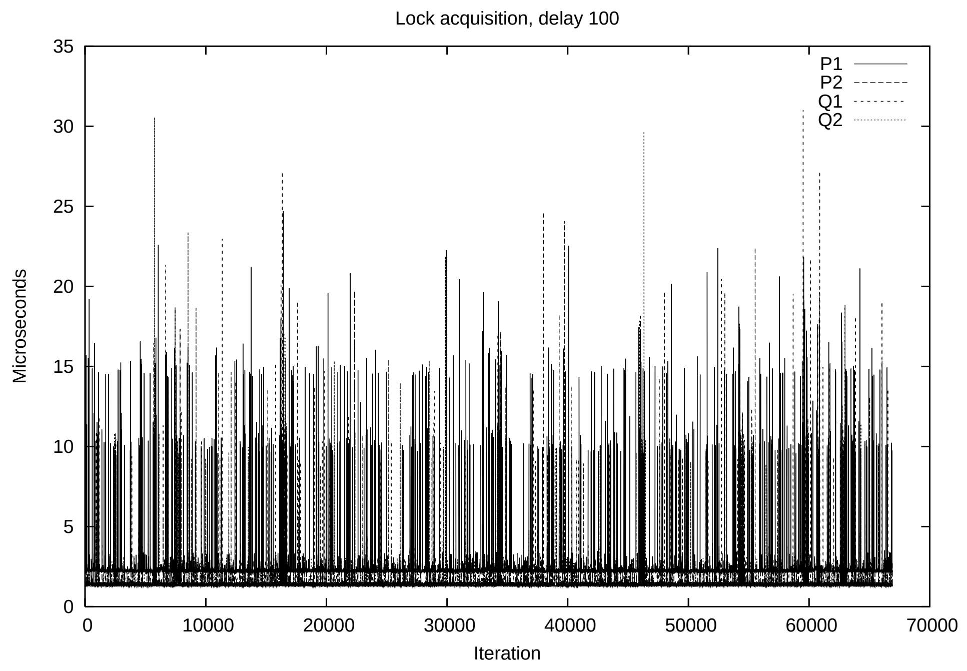


Figure 6.1: Lock acquisition latency, HAL delay 100 microseconds.

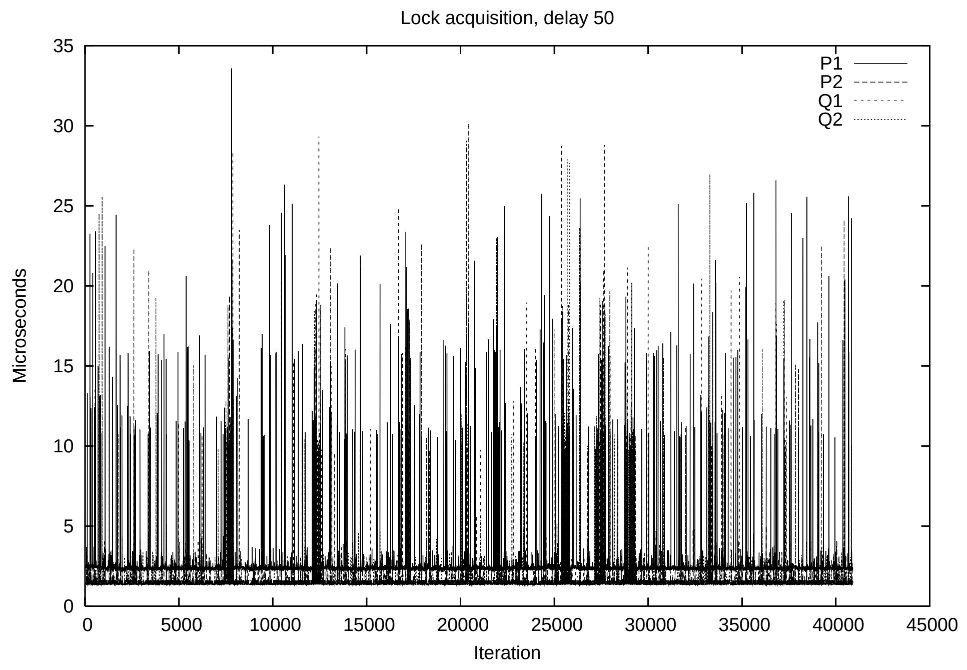


Figure 6.2: Lock acquisition latency, HAL delay 50 microseconds.

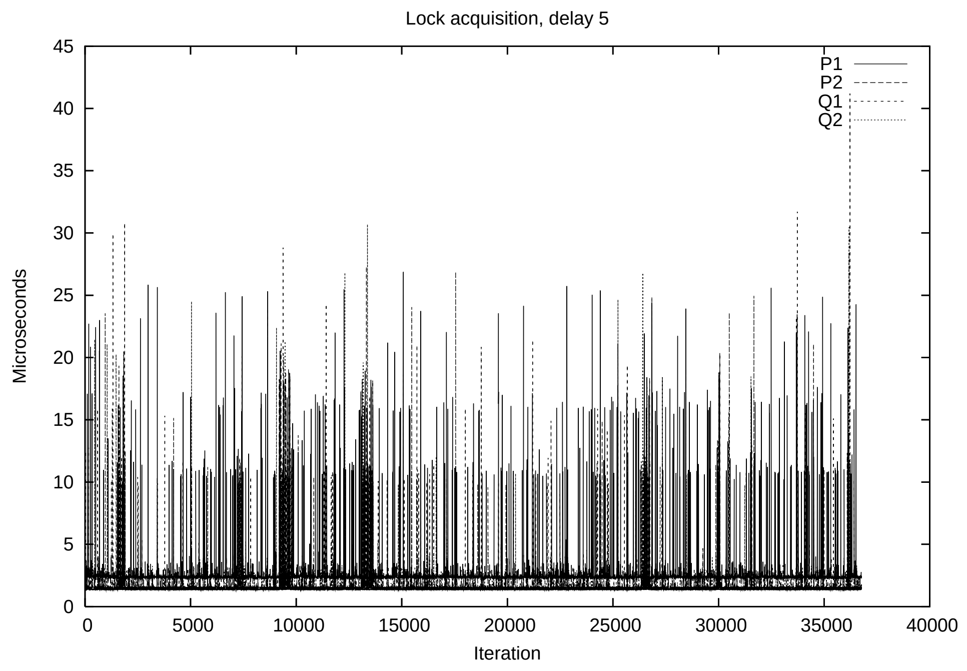


Figure 6.3: Lock acquisition latency, HAL delay 5 microseconds.

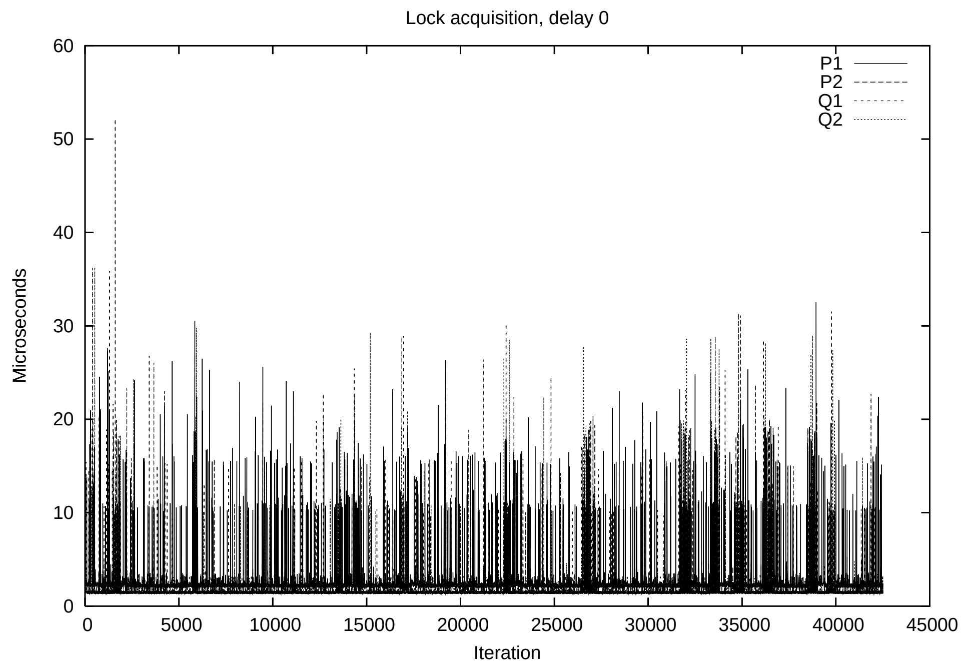


Figure 6.4: Lock acquisition latency, no HAL delay.

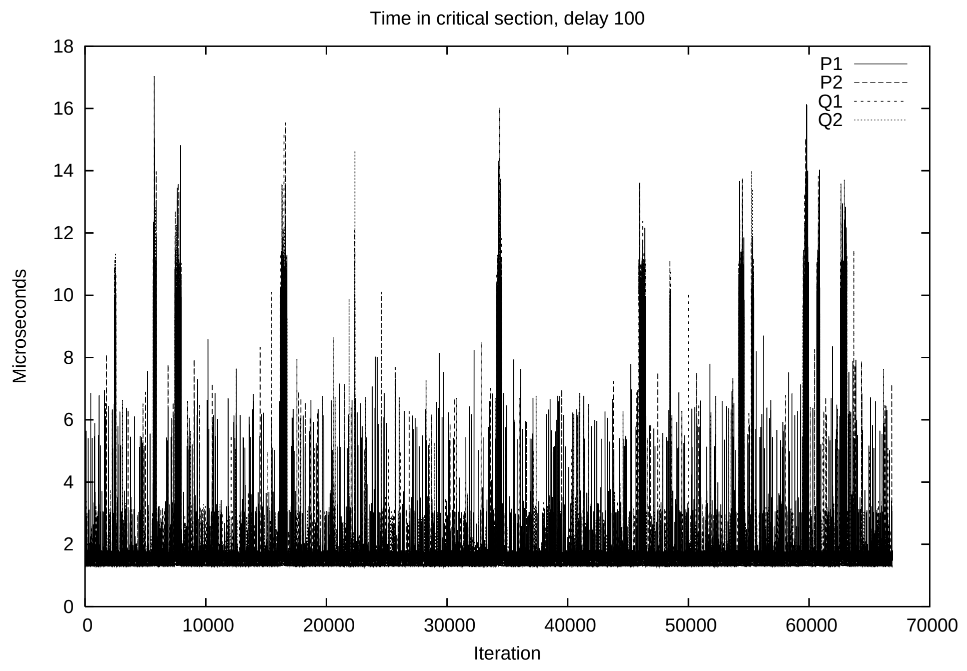


Figure 6.5: Critical section latency, HAL delay 100 microseconds.

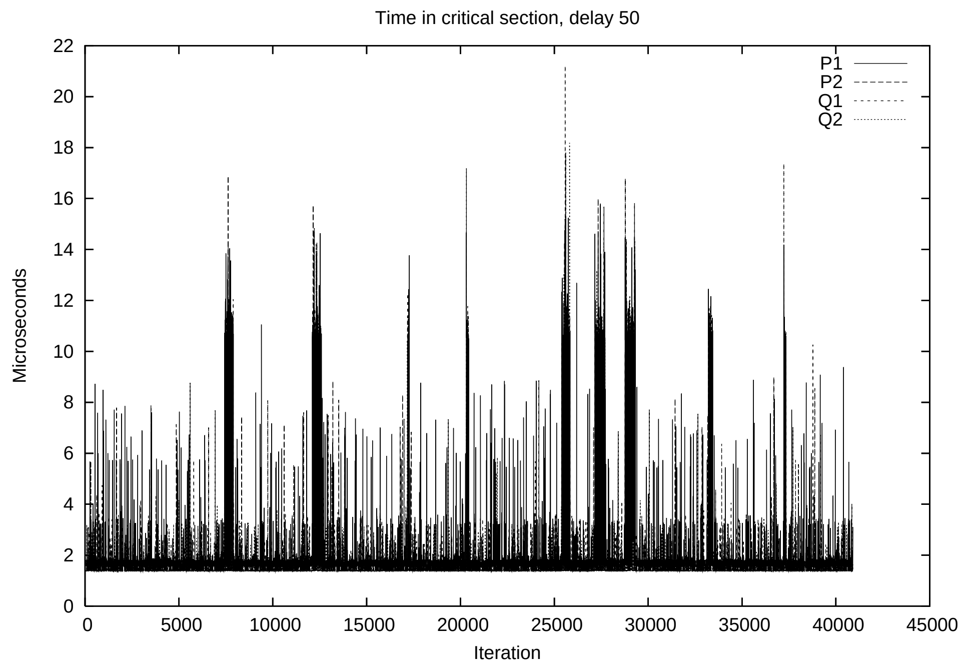


Figure 6.6: Critical section latency, HAL delay 50 microseconds.

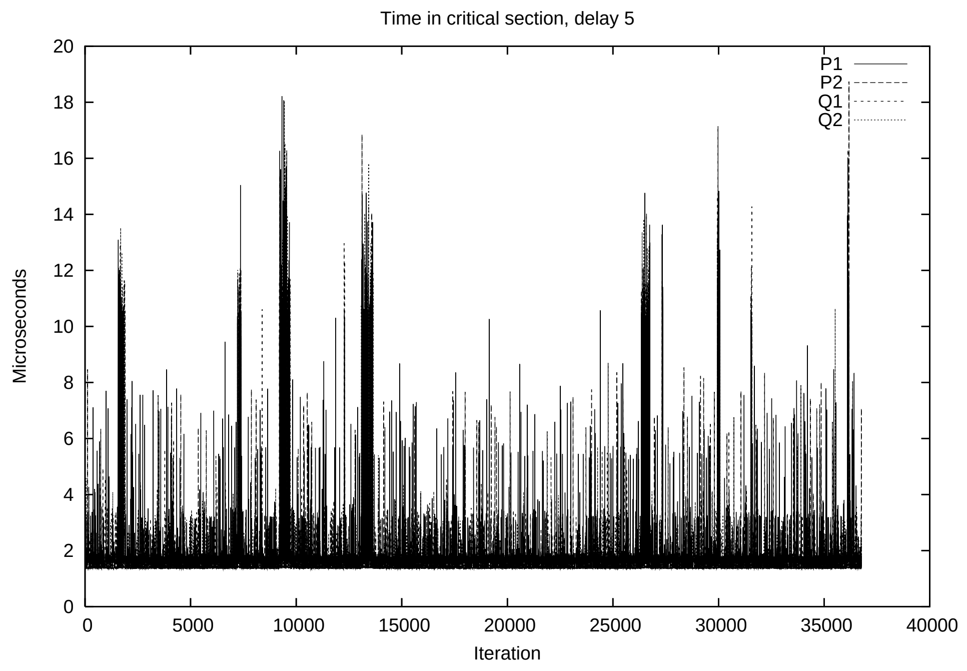


Figure 6.7: Critical section latency, HAL delay 5 microseconds.

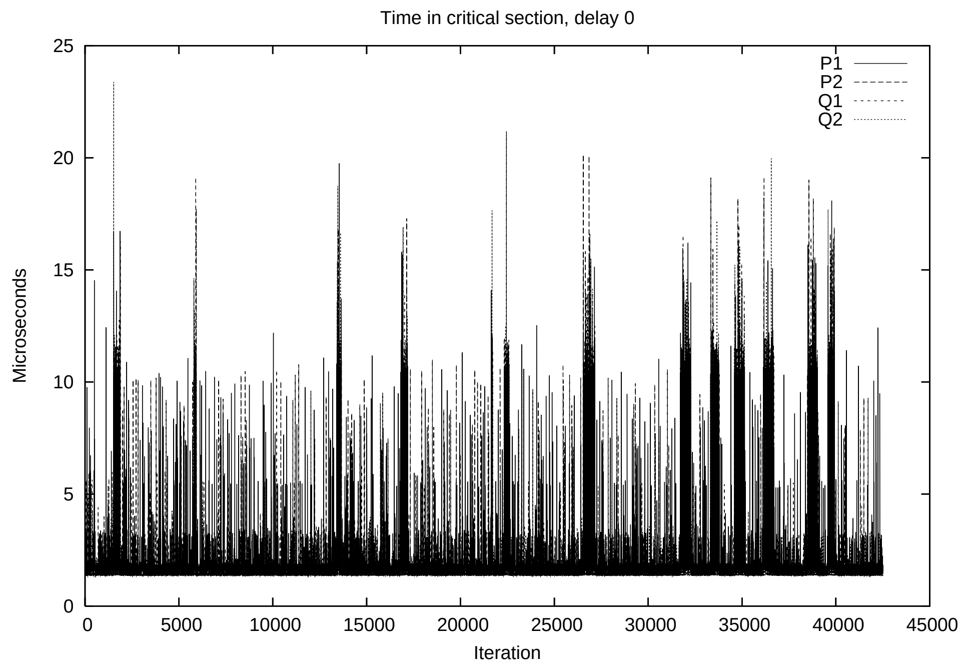


Figure 6.8: Critical section latency, no HAL delay.

7 CONCLUSIONS

In this thesis, the goal was to create a framework capable of soft real-time support for the FIL power system simulation. This goal has been accomplished by enabling soft real-time support for the FIL power system simulation at its 1 millisecond time step. This framework also accomplishes the goals of having an open architecture to be used with an arbitrary real-time simulation core, on any Unix-like operating system with the appropriate scheduling controls, and on any general purpose computer with a compatible operating system. As proof of its utility, the framework has been deployed and used to run experiments in the FACTS Interaction Laboratory.

8 FUTURE DIRECTIONS

One potential improvement to the framework described in this thesis would be to use a set of Unix signals to asynchronously notify the simulation process that data has been updated. In this way even the minimal shared memory polling that is performed by the simulation process could be eliminated. However, the number of signals available to Unix programs is quite small, so it is unclear whether a sufficient number of signals exist for the purposes of implementing a generic mechanism for asynchronous state update notification.

It should be noted that a potential error in the current implementation exists insofar as unused operating system signals are not masked, leaving a possibility for the operating system or other processes to interfere with the real time performance of the system. No evidence of such interference was observed in these experiments, but a more complex system may trigger this behavior, so if in doubt the signals should be masked.

It may be possible to integrate the entire simulation system into a single process with several POSIX threads. This would allow the removal of the external shared memory region and its associated management; a semaphore region will still be necessary unless the synchronization macros are ported to POSIX semaphores, however. A drawback to this approach, as noted above, is that a threaded system presents more problems with managing CPU affinity and scheduling priority, so this configuration may not be possible to implement on existing platforms.

Another avenue for exploration could be the implementation of the techniques described in [26], in which an interpolation technique is combined with a variable time step. This method attempts to account for events that happen during a time step. It would make the simulation more accurate, but at the cost of computational complexity, and it is unclear if the implementation of such a technique would be feasible in the presence of real-time constraints.

It may also be possible to develop this framework into a hard real-time framework on a hard RTOS if a hard real-time simulation core is utilized. It is unlikely that a power system simulation core can produce useful results under a hard real-time constraint, but other types of simulations not involving a non-linear solver may benefit from this approach.

BIBLIOGRAPHY

[1] M. Bacic, "On hardware-in-the-loop simulation," *Decision and Control, 2005 and 2005 European Control Conference. CDC-ECC'05. 44th IEEE Conference on*, pp. 3194–3198, December 2005.

[2] B. Lu, "The real-time extension of the virtual test bed: A multi-solver hard real-time hardware-in-the-loop simulation environment," Master's thesis, University of South Carolina, 2003.

[3] "RTDS Technologies, Inc. Homepage." Online. http://www.rtds.com/, fetched August 22, 2007.

[4] "The Open Source Definition." Online. http://www.opensource.org/docs/osd, fetched September 11, 2007.

[5] H. P. Figueroa, A. Monti, and X. Wu, "An interface for switching signals and a new real-time testing platform for accurate hardware-in-the-loop simulation," in *2004 IEEE International Symposium on Industrial Electronics*, vol. 2, pp. 883–887, May 2004.

[6] A. Monti, H. Figueroa, S. Lentijo, X. Wu, and R. Dougal, "Interface issues in hardware-in-the-loop simulation," in *2005 IEEE Electric Ship Technologies Symposium*, pp. 39–45, July 2005.

[7] Y. Sun, B. McMillin, X. F. Liu, and D. Cape, "Verifying noninterference in a cyber-physical system: The advanced electric power grid," in *Proceedings of the Seventh International Conference on Quality Software (QSIC)*, (Portland, OR), October 2007.

[8] W. Ren, L. Qian, M. Steurer, and D. Cartes, "Real time digital simulations augmenting the development of functional reconfiguration of PEBB and universal controller," in *Proceedings of the 2005 American Control Conference, 2005*, vol. 3, pp. 2005–2010, June 2005.

[9] S. Ayasun, S. Vallieu, R. Fischl, and T. Chmielewski, "Electric machinery diagnostic/testing system and power hardware-in-the-loop studies," in *4th IEEE International Symposium on Diagnostics for Electric Machines, Power Electronics and Drives, 2003 (SDEMPED 2003)*, pp. 361–366, August 2003.

[10] S. Ayasun, R. Fischl, T. Chmielewski, S. Vallieu, K. Miu, and C. Nwankpa, "Evaluation of the static performance of a simulation-stimulation interface for power hardware in the loop," in *2003 IEEE Bologna Power Tech Conference Proceedings*, vol. 3, p. 8, June 2003.

[11] X. Wu, S. Lentijo, and A. Monti, "A novel interface for power-hardware-in-the-loop simulation," in *2004 IEEE Workshop on Computers in Power Electronics, 2004*, pp. 178–182, August 2004.

[12] X. Wu, S. Lentijo, A. Deshmuk, A. Monti, and F. Ponci, "Design and implementation of a power-hardware-in-the-loop interface: a non-linear load case study," in *Twentieth Annual IEEE Applied Power Electronics Conference and Exposition, 2005 (APEC 2005)*, vol. 2, pp. 1332–1338, March 2005.

[13] Y. Liu, M. Steurer, S. Woodruff, and P. F. Ribeiro, "A novel power quality assessment method using real time hardware-in-the-loop simulation," in *11th International Conference on Harmonics and Quality of Power, 2004*, no. 11, pp. 690–695, September 2004.

[14] W. Zhu, S. Pekarek, J. Jatskevich, O. Wasynczuk, and D. Delisle, "A model-in-the-loop interface to emulate source dynamics in a zonal DC distribution system," *IEEE Transactions on Power Electronics*, vol. 20, pp. 438–445, March 2005.

[15] R. B. Wells, J. Fisher, Y. Zhou, B. K. Johnson, and M. Kyte, "Hardware and software considerations for implementing hardware-in-the-loop traffic simulation," in *The 27th Annual Conference of the IEEE Industrial Electronics Society, 2001 (IECON '01)*, vol. 3, pp. 1915–1919, Nov/Dec 2001.

[16] R. Friedland and B. Kulicke, "Digital simulation and hardware-in-the-loop test of controllers in electric power systems," in *First International Conference on Digital Power System Simulators, 1995 (ICDS '95)*, p. 277, April 1995.

[17] M. Steurer, S. Woodruff, N. Brooks, J. Giesbrecht, H. Li, and T. Baldwin, "Optimizing the transient response of voltage source converters used for mitigating voltage collapse problems by means of real time digital simulation," in *2003 IEEE Bologna Power Tech Conference Proceedings*, vol. 1, p. 6, June 2003.

[18] B. Lu, A. Monti, and R. A. Dougal, "Real-time hardware-in-the-loop testing during design of power electronics controls," in *The 29th Annual Conference of the IEEE Industrial Electronics Society, 2003 (IECON '03)*, vol. 2, pp. 1840–1845, November 2003.

[19] B. Lu, X. Wu, and A. Monti, "Implementation of a low-cost real-time virtue test bed for hardware-in-the-loop testing," in *32nd Annual Conference of IEEE Industrial Electronics Society, 2005 (IECON 2005)*, pp. 239–244, November 2005.

[20] B. Lu, X. Wu, H. Figueroa, and A. Monti, "A low cost real-time hardware-in-the-loop testing approach of power electronics controls," *IEEE Transactions on Industrial Electronics*, vol. 54, pp. 919–931, April 2007.

[21] "Real-Time Hardware-In-the-Loop (HIL) Simulation." Online. http://vtb.engr.sc.edu/research/reml/projects/rthil.asp, fetched August 22, 2007.

[22] L. U. Gökdere, C. W. Brice, and R. A. Dougal, "A virtual test bed for power electronic circuits and electric drive systems," in *The 7th Workshop on Computers in Power Electronics, 2000 (COMPEL 2000)*, (Blacksburg, VA), pp. 46–51, July 2000.

[23] S. M. Shah and M. Irfan, "Embedded hardware/software verification and validation using hardware-in-the-loop simulation," in *Proceedings of the IEEE Symposium on Emerging Technologies, 2005*, pp. 494–498, September 2005.

[24] P. Baracos, G. Murere, C. Rabbath, and W. Jin, "Enabling PC-based HIL simulation for automotive applications," in *IEEE International Electric Machines and Drives Conferences, 2001 (IEMDC 2001)*, pp. 721–729, 2001.

[25] V. R. Dinavahi, M. R. Iravani, and R. Bonert, "Real-time digital simulation of power electronic apparatus interfaced with digital controllers," *IEEE Transactions on Power Delivery*, vol. 16, pp. 775–781, October 2001.

[26] M. O. Faruque, V. Dinavahi, and W. Xu, "Algorithms for the accounting of multiple switching events in digital simualtion of power-electronic systems," *IEEE Transactions on Power Delivery*, vol. 20, pp. 1157–1167, April 2005.

[27] "Wind River: RTLinuxFree." Online. http://www.rtlinuxfree.com/, fetched November 3, 2007.

[28] "RTAI - the RealTime Application Interface for Linux." Online. https://www.rtai.org/, fetched November 3, 2007.

[29] "KURT: The KU Real-Time Linux." Online. http://www.ittc.ku.edu/kurt/, fetched November 3, 2007.

[30] E. W. Dijkstra, "Cooperating sequential processes," tech. rep., Technological University, Eindhoven, 1968. Second revision.

[31] D. C. Schmidt and T. Harrison, *Pattern Languages of Program Design 3*. Addison-Wesley, 1997.

[32] M. Ryan, S. Markose, X. Liu, B. McMillin, and Y. Cheng, "Structured object-oriented co-analysis/co-design of hardware/software for the facts power system," in *29th Annual International Computer Software and Applications Conference, 2005 (COMPSAC 2005)*, vol. 1, pp. 396–402, July 2005.

[33] M. Ryan, S. Simsek, X. F. Liu, B. M. McMillin, and Y. Cheng, "An instance-based structured object oriented method for co-analysis/co-design of concurrent embedded systems," in *30th Annual IEEE International Computers Software and Applications Conference*, (Chicago, IL), pp. 273–280, September 2006.

[34] L. Dong, M. L. Crow, Z. Yang, C. Shen, L. Zhang, and S. Atcitty, "A reconfigurable facts system for university laboratories," *IEEE Transactions on Power Systems*, vol. 19, pp. 120–128, February 2004.

[35] M. Crow, *Computational Methods for Electric Power Systems*. CRC Press, 2002.

[36] W. M. Siever, *Power Grid Flow Control Studies And High Speed Simulation*. PhD thesis, University of Missouri-Rolla, Rolla, MO, 2007.

[37] L. Zhang, Z. Yang, S. Chen, and M. L. Crow, “A PC-DSP-based unified control system design for FACTS devices,” in *IEEE Power Engineering Society Winter Meeting, 2001*, vol. 1, pp. 252–257, Jan/Feb 2001.

[38] “Fault-Tolerant and Secure Power Grid Systems using FACTS Devices; FIL Homepage.” Online. http://filpower.umr.edu/, fetched August 22, 2007.

VITA

Ryan C. Underwood was born in Fort Worth, Texas on September 17, 1980. He was raised in Saint Peters, Missouri where he attended primary, secondary, and high school. Following high school, he enrolled at the University of Missouri-Rolla where he attended until 2006. He was awarded a Bachelor of Science in Computer Science in 2004, after which he pursued a M.S. in Computer Science, also at UMR. His Master's studies were primarily focused on improving the FACTS Interaction Laboratory power system simulation and implementing a simulation framework for improved real-time performance. Following his thesis work, he received a M.S. in Computer Science from UMR in December 2007.