2004

NASA FACULTY FELLOWSHIP PROGRAM

MARSHALL SPACE FLIGHT CENTER

THE UNIVERSITY OF ALABAMA
THE UNIVERSITY OF ALABAMA IN HUNTSVILLE
ALABAMA A&M UNIVERSITY

An Open Source Simulation System

Prepared By:	Thomas Slack
Academic Rank:	Assistant Professor
Institution and Department:	The University of Memphis Department of Engineering Technology
NASA/MSFC Directorate:	Engineering, ED 19
MSFC Colleague:	Drew Hall

Doing Real Time, without the Real Royalties

An investigation into the current state of the art of open source real time programming practices. This document includes what technologies are available, how easy is it to obtain, configure, and use them, and some performance measures done on the different systems. A matrix of vendors and their products is included as part of this investigation, but this is not an exhaustive list, and represents only a snapshot of time in a field that is changing rapidly. Specifically, there are three approaches investigated:

1. Completely open source on generic hardware, downloaded from the net
2. Open source packaged by a vender and provided as free evaluation copy
3. Proprietary hardware with pre-loaded proprietary source available software provided by the vender as for our evaluation

The evaluation was one of two projects done in 10 weeks by two summer fellows of the NASA Faculty Fellowship Program (NFFP) at Marshal Space Flight Center (MSFC) and three undergraduate students all from different schools, two of whom were summer interns on the Visiting Researcher Exchange and Outreach (VREO) program. The third was a summer cooperative student. All of the work done by the group for ED 19 is stored in the Virtual Research Center (VRC) on line. The group will continue to collaborate with this mechanism over the next year and hopefully continue this effort next summer. The investigation was spit into two projects that were each headed by one of the faculty fellows. One was an investigation of collaborating networked computers the second is summarized in this paper.

The members of Division ED 19 had a goal to upgrade the capabilities of the Marshall Avionics System Test-bed (MAST) lab. A review of the capabilities of the lab is instructive. This is composed of hardware in the loop hard real time computer systems and robotic mechanism for simulating attitude adjustments. The robotic system consists of a table with roll pitch and yaw capabilities. Figure 1 shows a general view of how the systems interact to do hardware in the loop simulation. The system to be tested (T) is either strapped to the table, or in the case of some optical systems is positioned to look at an object strapped to the table. There may be additional simulation equipment and software (A) provided by the stakeholders involved in testing T. As much as possible T is placed in the environment and has the same signals that it would have in the real situation. Models of how the physical systems that are interacting with T may be in A, or in the MAST lab computers (M). M has the additional task of controlling the robotics in the lab to maintain whatever attitude that is needed for T to maintain the illusion of being on the mission.

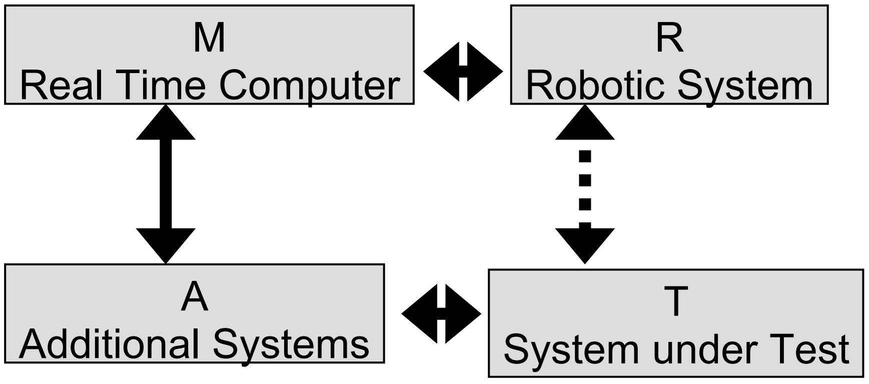

Figure 1: MAST Lab Equipment Configuration

This is a very simple view, but since the project was concerned with the upgrading and replacement of M, it is important to understand the capabilities required, but the details are less important.

The current computational capabilities of the computers, called M, are as follows: The systems, manufactured by Silicon Graphics Inc. (SGI), are called by them “Challenge” or “Origin” computers. They are an Irix OS on a multiple 200MHz MIPS processor shared memory platform. This platform has the capability of doing the functions described above, but should be upgraded as it is 7 years old. Since the upgrade was to be done, a review of the cost history of the MAST labs reveals that they are very expensive to maintain because of the software in the essentially proprietary systems build for NASA by SGI. There were also anecdotal incidences associated with the lack of software source that led personnel to look to systems that are open source.

During the course of the summer, it became apparent that the upgrade of the computer facility was timely because the new systems (particularly Integrated Vehicle Health Maintenance (IVHM)), that need to be simulated are orders of magnitude more complex than the ones simulated in the past. This fact points to using a different architecture for simulation than has been used in the past. More will be said about this in the Results and Conclusions section below.

It turns out that the real-time open source community on the Internet has made significant strides in the last few years, and so this upgrade is an opportunity to move away from more expensive proprietary towards an open source solution.

The goal then simply stated was to:
Define an architecture for open source based simulation systems to support large scale locally distributed simulations in real-time with hardware in the loop.

The part of the goal that this investigation covers is to discover what real time software is available in the open source community, how good is it, and how hard is it to use. We wished to examine and compare different systems from those available, so for the first two weeks of the summer we researched on the Internet to find what was available. The result of that search was collected in a matrix of currently available systems. This matrix, included on VRC, cannot be considered all encompassing.

The Test Systems

1. System RTAI

The first system is a completely open source system on generic hardware, downloaded from the net. This system is called RTAI, because RTAI was the open source real time kernel software used. The computer was placed on the desk of the author for use as a word processing and email tool during the summer. This turned out to be a Pentium 4 single processor system with 256 Mbytes of memory running at 2.523 GHz. This system was called debian after it was installed.

2. System RTLinux

The second system was an available Red Hat Linux system, called opus0, which was already in the MAST lab. This turned out to be a Dual processor Pentium 3 (Xeon) system with 1GByte of memory running at 2.795GHz. I turned hyper-threading off in an effort to be as time determined as possible. An evaluation copy of the RTLinuxPro software available from FSMLabs was installed on this system.

3. System Concurrent

The third system was provided as an evaluation by Concurrent Computer Corporation. This turned out to be a Dual processor Pentium 4 system with 2GBytes of memory running at 3.0 GHz. This system was provided with a real time clock card that is a key to the real time performance of the system. The other two systems achieved their real time performance by using a dual kernel model. This system used a modified Red Hat Linux 2.6 kernel that they call Red Hawk. Because the 2.6 kernel is fully preemptable, they did not use a dual kernel model to achieve the performance.

Installation of the Systems

The PC the RTAI system was to be installed on had two partitions already created on its 18 GB disk. It was therefore fairly straightforward to backup the contents of the second partition onto the extra space on the first, and repartition the disk to have four partitions instead of two. The contents were restored to one of the three new partitions on a FAT32 file system volume; the other two became the swap space and root partition of the new Linux system to be installed. This process took about two days. Under ideal conditions it would take about two hours.

The next step was to install the Debian linux system. Once the Linux was installed, it was by default a 2.2.20 kernel. While, on the Debian CDs was source for several kernels none of them was 2.2.20. After discovering which sources were available on the CDs, and which were needed by RTAI, we discovered a match between the 2.4.17 linux kernel which we had a source for and with the 24.1.13 RTAI version. The next step was to rebuild that kernel and reboot with it. Once the new kernel was installed it had to have the RTHAL patch installed. The RTAI then installed without a hitch in a few minutes.

The second system, RTLinux, was a download from FSMLabs. They provided us with the key to a download from their FTP site, and we downloaded and installed their 2.4.18 kernel on Opus0. This first version did not boot at all. Opus0 has SCUZZY disks, which I understand are more difficult to boot from. We consulted with FSMLabs and they provided us with a second download with new drivers on it. This 2.4.25 kernel download booted correctly, but was a beta version of their software and had some other problems that made it hard for us to use. They also had not understood that we had a multiple CPU system. When we mentioned this to FSMLabs they worked for several days and provided us with a third version.

The third system, Concurrent arrived from the vender, fully functional. This provides one end of the spectrum of installation of the systems. The other end is provided by the RTAI system. Between these extremes was the experience of installing the RTLinux system provided by FSMLabs.

The Test

FSMLabs provided a simple piece of code that we decided would be a good program to compare functionality on the different platforms. This code was written with POSIX threads. It creates a thread that has priority zero with the following steps:

1. Calculate a time approx. 1 ms in the future.
2. Ask the system to sleep till then.
3. Wake up and take the time.
4. Calculate how late you woke up.
5. Store the worst number you have found so far, and loop to 1.

This code was chosen and translated to the other two systems with minor changes. On the FSMLabs system, a second thread with very low priority was used to print the numbers each time a new worst case was found. The second thread shared the memory where the worst case was stored. The first thread would signal a semaphore each time it stored a new value in the worst case. The second thread would wake up and print the value sometime while the first thread was sleeping. Also on this system, a recording thread is generated for each CPU available. We added code that would make one CPU exclusively work with one thread. Since there were two CPUs on this system, after we turned hyper-threading off, we allowed the other CPU to handle all of the threads and processes left. In this way we hoped that the exclusive thread would be able to get the best results for this system.

On the RTAI system, the printing program was in a separate process. The programs communicated with FIFOs. Since this was a uniprocessor, the issue of making the CPU exclusively work on one thread did not arise.

On the Concurrent system, the translation was almost directly from the FSMLabs code, without the extra threads for each processor. This system has a special scheduler, which allows you to guard a CPU from interrupts and designate it as being exclusively being used by a single process. We used this scheduler to create two processes running the single thread. We guarded one CPU so that it would exclusively run the one thread. The other CPU was allowed to run the rest of the system. Since this system had hyper-threading turned on, we guarded both the other virtual CPUs so that they would be idle.

Results and Conclusions

The results of the sleep-jitter test program are shown in Figure 2.

	Test duration	CPU0	CPU1
RTAI	89 min	17.4	
RTLinux	25 min	92.85	158.24
Concurrent	60 min	133.82	

Figure 2: The Results of the Sleep-jitter test

As we look at these results we realize that there are many factors making the comparison weak. First the second two machines are symmetric multiprocessing machines, and the first is a uni-processor. Second, the RTLinuxPro system is a Beta release, and may therefore have some things not tested nor tuned. Third the second two machines are networked, and the first one, has only a loopback interface working. With all of these weaknesses, I believe that the order of the systems would not be changed significantly, only the distance between them.

During my discussions of these technologies with the venders, I got many numbers for the quickest interrupt latency that could be obtained. As they talked about the different systems that are out there they gave their opinion as to how the systems would fare in comparison. In general, this is the order that they gave. One thing can be said is that the completely open source solution is as at least as good any other available in pure performance.

As far as programming is concerned, when you are programming for RTAI it is certainly disconcerting at first to be continually adding modules to the kernel and removing them as you test software. It is also irritating to discover that under these conditions a normally rock solid Unix OS is as vulnerable to crashing as any Windows or Macintosh OS as you try out your software. After a while you get used to doing a sync before trying any new module just to be sure the disk is clean since you may have to power cycle the machine to get it back.

However, just because you do more programming in the user space, does not mean that you will be less likely to deadlock using RTLinux. It is the nature of real time programming on the edge of what a processor can perform to make an error that deadlocks the system regardless of the mechanism used to do it.

Of the three systems, the concurrent was the least likely to have a programming problem. This was because you could do the entire program in user space and in general test it. They use the fancy scheduler to play with placing it at a higher priority. You must pay a price in performance for this capability. First, it is using processes instead of threads to schedule. This is inherently less efficient. Second, the tools for tracing the operation must have some overhead however small.
Finally, we must consider the IVHM project when considering the upgrade. Such a project will lend itself to simulation by a large number of parallel CPUs working together. A benefit of using open source for real time simulation is that many more processors can be considered in the architecture because you are not paying for the software by the machine.

It is the opinion of the author that perhaps all of these systems be used in the upgrade of the computational facilities of the MAST lab. As a first step, an 8 processor CPU (perhaps with 4 now and upgradable to 8) with the concurrent technology be considered. This would be the development machine. A 16 or 32 node cluster of processors with minimal linux OS and either RTAI or RTLinux might be considered for the next addition. The ability to get the speed you need in the individual processors, at the same time control the whole with a overall system like that proposed by Gene Sheppard in his final report would give you the flexibility to expand to as large a system as you need.

The ability of the development machine to expand to multiple nodes as needed by tying the clock cards together is also a plus.