Implementing Real-Time Robotic Systems Using CHIMERA II

David B. Stewart* Donald E. Schmitz Pradeep K. Khosla†

The Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract: *This paper describes the CHIMERA II programming environment and operating system, which was developed for implementing real-time robotic systems. Sensor-based robotic systems contain both general and special purpose hardware, and thus the development of applications tends to be a very tedious and time consuming task. The CHIMERA II environment is designed to reduce the development time by providing a convenient software interface between the hardware and the user. CHIMERA II supports flexible hardware configurations which are based on one or more VME-backplanes. All communication across multiple processors is transparent to the user through an extensive set of interprocessor communication primitives. CHIMERA II also provides a high-performance real-time kernel which supports both deadline and highest-priority-first scheduling. The flexibility of CHIMERA II allows hierarchical models for robot control, such as NASREM, to be implemented with minimal programming time and effort. CHIMERA II is currently being used with a variety of robotic systems, including the CMU Direct Drive Arm II and the CMU Reconfigurable Modular Manipulator System.*

Keywords: *Real-Time Operating System, Robot Control, Sensor Integration, Telerobotics, Standardization, NASREM.*

1 Introduction

Sensor-based control applications, such as robotics, process control, and manufacturing systems, present problems to conventional operating systems because of their need for several different hierarchical levels of control, which typically fall into three broad categories: *servo levels, supervisory levels,* and *planning levels.* The *servo levels* involve reading data from sensors, analyzing the data, and controlling electro-mechanical devices, such as robots and machines. The timing of these levels is critical, and often involves periodic processes ranging from 100 Hz to 1000 Hz. The *supervisory levels* are higher level actions, such as specifying a task, issuing commands like *turn on motor 3* or *move to position B,* and selecting different modes of control based on data received from sensors at the servo level. Time at these levels is a factor, but not as critical as for the servo levels. In the *planning levels* time is usually not a critical factor. Examples of processes in this level include generating accounting or performance logs of the real-time system, simulating a task, and programming new tasks for the system to take on.

In order to satisfy the needs of sensor-based control applications, a flexible real-time, multitasking and parallel programming environment is needed. For the servo levels, it must provide a high performance real-time kernel, low-overhead communication, fast context switching and interrupt latency times, and support for special purpose CPUs and I/O devices. For the supervisory levels, a message passing mechanism, access to a file system, and scheduling flexibility are desired. The real-time environment must be compatible with a software development workstation that provides tools for programming, debugging, and off-line analysis, which are required by the planning levels. A popular high level language must be available to minimize the learning time of the system. The details of the hardware should be isolated from the user, providing the user with a common software base regardless of hardware configuration. Finally, the real-time operating system should be designed so that programs running in simulation under a time-sharing environment can be incorporated into the real-time environment with minimal effort. CHIMERA II provides such an environment that is capable of supporting all levels of real-time sensor-based robot control applications on a multiprocessor computer system.

Although the motivation for developing CHIMERA II is similar to the ideology behind its predecessor CHIMERA[1], its implementation is radically different and includes significant improvements, mostly in the areas of support for multiple processors and VME backplanes, and a higher performance kernel with improved real-time scheduling. A previous paper [2] provides background into the motivation and issues considered in redesigning CHIMERA.

CHIMERA II is not the only system designed to address real-time control applications. Several real-time operating systems currently exist for such applications. These include the commercial operating systems such as VRTX, by Ready Systems [3]; iRMX II, by Intel [4]; and VxWorks, by Wind River Systems[5]; and research operating systems, such as CONDOR[6]; SAGE[7]; and Harmony[8]. In general, these systems suffer from one or more of the following: lack of multiprocessing capabilities, poor or insufficient interprocessor communication and synchronization mechanisms, no provisions for joining the simulation and real-time environment, both in terms of UNIX compatibility and communication primitives, or a lack of support for easily incorporating I/O devices and special purpose processors into the environment. The above systems require users to spend a large amount of their time developing features that should be in the operating system. CHIMERA II was designed especially for robotics systems; it provides the necessary features for reducing development time and increasing performance of real-time robotic applications.

One of the more elaborate hierarchical control architectures proposed is the NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM) [9], which heavily influenced many of the CHIMERA II design decisions. The NASREM architecture is described in more detail in the next section. Section 2 describes the hardware required to support CHIMERA II; while the software is described in Section 3. The performance of CHIMERA II is outlined in Section 4. CHIMERA II is then summarized in Section 5 as the recommended underlying system for developing real-time robotic systems.

1.1 Overview of the NASREM Model

Figure 1 shows a block diagram of NASREM. It consists of a hierarchy of several levels (nominally six levels). Each level is implemented as sets of three distinct modules, called *sensory processing, world modeling,* and *task decomposition.* The *sensory processing* modules are responsible for obtaining and integrating information from the system. At the lowest levels, this involves reading the data from different sensors in a system, while for the upper levels sensory information is more general and hardware independent. The *world modeling* modules control access to information within the global database, and provide the necessary synchronization for other modules which must update the world model. Models are application specific, and can be in the form of algorithms, data sets (typically for lower levels) and knowledge bases (for the upper levels). The *task decomposition* modules initiate actions based on user input and information within the world

*Dept. of Electrical and Computer Engineering

†Assistant Professor, Dept. of Electrical and Computer Engineering

CH2872-0/90/0000-0252 © $1.00 1990 IEEE

model. The upper-level commands would typically be of the form *move to point B*, while the servo-levels produce low-level commands such as *apply torque X to joint Y*.

A major goal of NASREM is to provide a standard architecture for incorporating multiple sensors and robots into a single application. Using a standard architectural model allows efforts in various research organizations to be leveraged and new technologies to be brought together quickly. Other advantages of standardization include portability and expandability of real-time code, ease of code development, and the provision of guidelines for system integration. CHIMERA II has the ability to hide the details of the hardware from the user, thus allowing the implementation of hardware independent code, which is a first and necessary step towards standardization.

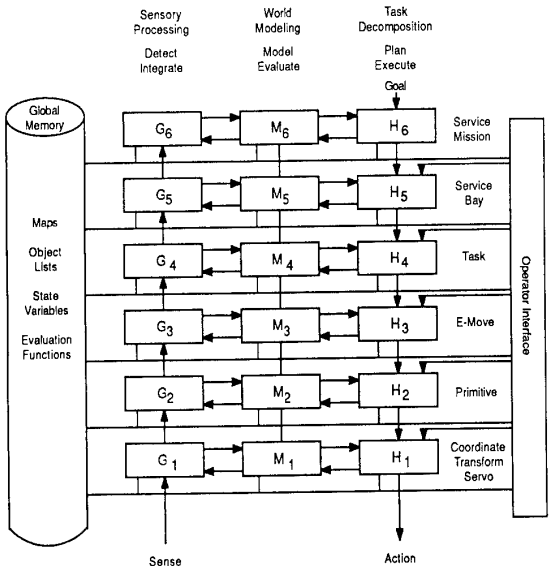

Figure 1: NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM) (from [9]).

2 Hardware Configurations

A typical hardware configuration for applications supporting hierarchical robot control would consist of multiple general purpose processors, possibly on multiple backplanes. The system may contain special processing units, such as floating point accelerators, digital signal processors, and image processing systems. The system will also contain several sensory and control devices, such as force sensors, cameras, tactile sensors, and range finders to gather data about the environment, and a variety of actuators, switches, and amplifiers to control robotic equipment.

Figure 2 shows one of many possible hardware configurations, and is based on the hardware architecture recommended by NASREM[9]. The hardware for each level of the hierarchy is on a separate VME bus, separated by gateways. In larger systems, there may be multiple VME buses dedicated to the same level, possibly connected in a star configuration. In a very simple system, there may be only one VME bus for the entire system.

Tasks at the same level will typically have high volume communication. A dedicated VME bus for that level reduces the memory bandwidth for intertask communication, since different levels need not compete for the same VME bus. In smaller applications where memory bandwidth is not a problem, multiple levels can share the same VME bus. In addition to the VME buses for each level, other high speed data buses can be used

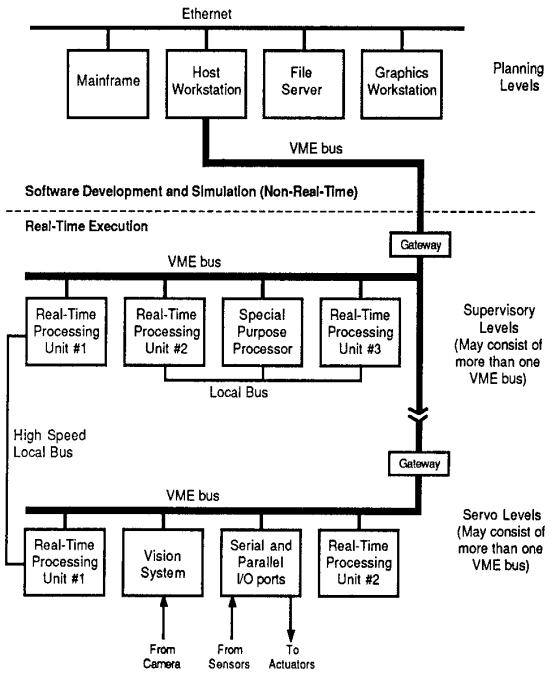

Figure 2: Generalized hardware configuration supported by CHIMERA II

either for fast local communication between neighboring processors, or as dedicated data paths for sensor and actuator signals.

The highest level of the hierarchical control consists of the non-real-time environment. One or more workstations can be connected via ethernet, with at least one of them connected via a gateway to the real-time environment, which we call the *host* workstation. Various utilities such as simulators, debuggers, graphical interfaces, and file systems are located at this level.

2.1 A Sample Application

CHIMERA II is currently implemented to run with a Sun host workstation, running SunOS 4.0.3. The effort required for porting the code to a different host should be minimal if the operating system on the new host supports System V interprocess communication and the enhanced *mmap()* facility that Sun provides. The Real-Time Processing Units (RTPUs) are VME-based single-board computers. Thus far, supported RTPUs are the Ironics IV32XX series with M68020 processors, and Heurikon M68030 processors. Porting to other M68020 or M68030 processors would require minimal effort. Porting to an entirely different processor family, such as the SPARC or 80X86 would require substantial effort. The VME to VME gateways are typically VME-VME adaptors, with the BIT3 Models 411 and 412 currently supported. These adaptors operate only in master/slave mode, where the address space of the slave VME bus is mapped into a part of the address space of the master VME bus through an *address window*. CHIMERA II has been designed to use these master/slave adaptors since they are the most popular type of adaptor on the market. Adaptors that use DMA to transfer data can be used, but the code required to support them has not yet been implemented.

The hardware currently supported is based on the needs of the Advanced Manipulators Laboratory at Carnegie Mellon University (CMU). Within our own labs, three very different systems are currently using CHIMERA II. These are the CMU Direct Drive Arm II (DDArm II)[10], The CMU Reconfigurable Modular Manipulator System[11], and the Flexible Arm

Manipulator[12]. The diverse requirements of these systems have proven the flexibility of CHIMERA II to support a wide variety of actuators and sensors.

The CMU DDArm II is an example of a multi-sensor robotic system. Its current hardware configuration is shown in Figure 3. The system consists of three VME backplanes, separated by BIT3 adaptors, and a Multibus backplane separated by a VME-Multibus adaptor. No RTPUs reside on the Multibus; all processors on that bus are treated as devices. A Sun 3/260 is used as the host workstation, with one Heurikon M68030 and two Ironics M68020 processors as the RTPUs. Also included in the system is a Mercury 3200 floating point accelerator with 20 Mflop peak performance, an Imaging Technology vision system, and six Texas Instrument TMS320 digital signal processors. Several serial and parallel I/O ports are used to connect to position, force, and tactile sensors, while an analog input connects to a camera on the end effector. A six-degree-of-freedom joystick, a terminal, and the Sun keyboard and mouse provide the interfaces for human interaction with the system.

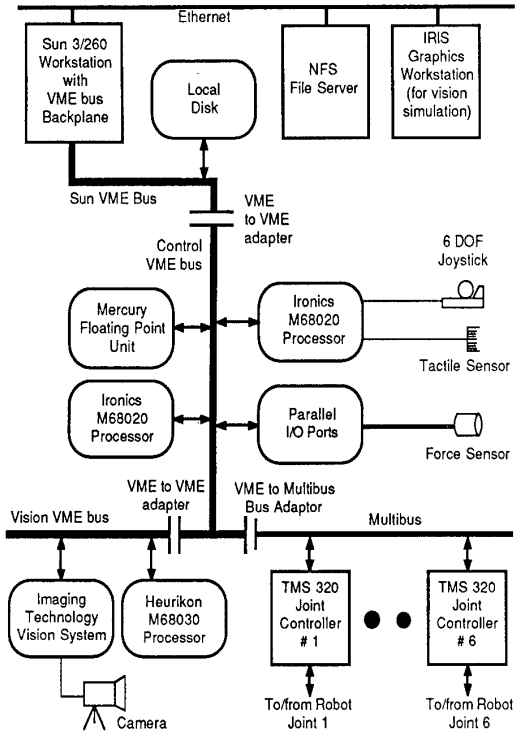

Figure 3: CMU Direct Drive Arm II hardware configuration

3 Software Architecture

The concept of a layered hardware platform eases system planning and integration, and provides convenient guidelines for developing large systems. Such layered hardware with multiple processors, however, presents nightmares to system programmers who must worry about the multitude of different addresses, address spaces, specialized devices, and hardware dependencies. It becomes extremely difficult to write standardized software when the hardware is very diverse.

CHIMERA II was designed to remove these problems from the user. First it provides a communications layer that makes all the interprocessor communication complexities transparent to the user. Second, it provides a kernel with device drivers that makes the very different hardware look similar.

3.1 Overview of CHIMERA II Software

Control processes running at the *servo levels* involve reading data from sensors, analyzing the data, and controlling electro-mechanical devices, such as robots or machines. The timing of these levels is critical, also known as *hard real-time*, and often involves periodic processes ranging from 100 Hz to 1000 Hz. CHIMERA II provides both *minimum-laxity-first* and *highest-priority-first* schedulers for programming in real-time. The *minimum-laxity-first* scheduler allows the user to specify tasks with execution deadlines. A failure handler is called whenever a task fails to meet its deadline as a result of a system overload.

In most robotic systems, the CPU bandwidth provided by a single processor is insufficient, thus creating the need for multiple processors. If the cost or complexity of communication between multiple processors is too high, however, it may nullify the effect of parallelism. CHIMERA II addresses this problem by exploiting the fact that all processors are on the VME bus, and uses direct block copies for all communication, instead of using time-consuming network protocols such as TCP/IP. The interface to these communication primitives is transparent across multiple processors, thus simplifying the programming of multiprocessor applications.

The planning levels do not necessarily require a real-time scheduler. In many cases, the non-real-time environment offered by a typical UNIX workstation is desirable to provide both a graphics terminal and a development environment. Traditional methods of linking non-real-time to real-time systems via ethernet and using network protocols is both difficult to program and requires a huge amount of overhead. Ideally, from the user's point of view, there is no difference between the real-time and non-real-time systems. Our design of CHIMERA II allows processes on the host UNIX workstation to communicate directly with the processors in the real-time environment, using the same synchronization primitives. There is no need for any complex network protocol to establish such communication.

Applications for CHIMERA II are programmed using the C language. Using languages other than C should be possible; however no consideration was given to this aspect during the development of CHIMERA II. Assembly language can also be used for specialized applications, but it is usually not necessary, since CHIMERA II provides the necessary features for writing exception handlers and device drivers in C.

Many standard C library calls are available, such as the *stdio, strings,* and *math* libraries, and also various utilities such as *quicksort, binary search,* and an optimized *block copy*. With the exception of the *stdio* library, these libraries provide identical functionality to their UNIX counterparts. The *stdio* library has been enhanced to run within a shared memory environment. Files can be shared among multiple tasks on the same processor. The *stdio* library incorporates the appropriate mutual exclusion to ensure the integrity of the file buffers. In addition to the standard library routines, CHIMERA II emulates many of the UNIX system calls, such as *open(), read(), write(),* and *mmap()*. These routines all generate remote procedure calls when the needed resource belongs to a different processor within the system.

3.2 Real-Time Kernel

A copy of the CHIMERA II real-time kernel executes on each RTPU. The kernel provides all the scheduling and task control primitives through a C library. The low-level details such as the model or manufacturer of the RTPU are built-in to CHIMERA II and are hidden from the user.

One of the major goals when developing the kernel was to provide the required functionality at the highest performance possible, by sacrificing traditional operating system features which are hardly used. Our basis for measuring performance is the amount of CPU time during normal execution which must be dedicated to the operating system functions, such as scheduling, task switching, and communication overhead. Some of the major design decisions made in developing a kernel especially for real-time sensory control are described below. For the sake of brevity, we have left out many details of the software. These are available in [15].

Tasks: A task in CHIMERA II is also known as a thread or lightweight

process in other operating systems. A user program which is downloaded into an RTPU consists of a single executable file. The kernel is supplied as a C library and is linked into the executable image. When a program is downloaded to an RTPU and executed, some kernel initialization is performed, following which the user's *main()* routine is *spawned* as a task. Any other task can then be started from *main()*.

Task communication: There is no parent/child relationship among tasks. Any task can communicate with any other task either through local shared memory or local semaphores. Within a single executable file, all global variables are automatically shared among all tasks. Local semaphores are available and can be used either to provide mutual exclusion during critical sections (binary semaphores), or to provide synchronization among tasks (general or counting semaphores). Global shared memory, remote semaphores, and message passing are also available to tasks when communication across multiple processors is required. The global communication is described in Section 3.4.

Intertask security: In general, all of the tasks running on a given RTPU (or set of RTPUs) are written and invoked by a single user. It is reasonable to assume that these tasks are designed to cooperate. We have thus sacrificed the intertask protection, allowing one task to access the address space of any other task. This has resulted in the elimination of a lot of overhead incurred in performing system calls or their equivalents.

Memory management: The total address space used by all tasks on one system is limited by the physical memory available on the RTPU. CHIMERA II does not provide any virtual memory, as the memory management and swapping overhead not only decreases the performance of a system drastically, but it also causes the system to become unpredictable, thus violating one of the major rules of real-time systems. CHIMERA II provides its own version of the *malloc()* family of routines to allocate real-memory.

Special purpose processors and devices: The CHIMERA II kernel uses a UNIX-like approach of using device drivers to isolate the user from the details of special hardware. The kernel supports the *open, close, read, write, ioctl,* and *mmap* drivers. These drivers are usually much simpler to write than their UNIX counterparts because of the lack of intertask security, as described above. Special purpose processors usually have memory which can be memory mapped, thus making the entire address space of the processor available to the tasks using it. Accessing special purpose processors is strictly via master/slave relationships, where tasks on the RTPUs control the execution of code on the special processor.

Exception and interrupt handlers: User-defined exception and interrupt handlers can be defined either on a per-task or per-RTPU basis. This allows users to alter the default action of various exceptions, which is to halt the task. For example, a *bus error* exception can be trapped so that a task that tries to access unavailable memory on the VME bus does not die. Similarly, a task can catch a *division-by-zero* exception and modify its computation accordingly, instead of having to check for zero on every calculation. CHIMERA II provides the facilities to write these handlers in C. The C routine is declared with a special header, and installed dynamically by a subroutine call. Routines are also provided to enable or disable interrupts, or to lock the CPU for short periods.

3.2.1 Real-Time Task Scheduler

One of the main components of any real-time kernel is its scheduler. CHIMERA II provides a policy/mechanism separation scheme[13] which allows the user to replace the standard schedulers with application specific algorithms. The CHIMERA II default scheduler algorithm was developed by incorporating various standard algorithms with extensive experimentation and tuning to obtain the best performance for typical job mixes. The scheduler is based on a combination of *minimum-laxity-first*[14], *highest-priority-first*, and *round-robin* scheduler techniques.

The heart of the scheduler is the *minimum-laxity-first* deadline scheduler. A task can specify its timing requirements in terms of a deadline time, and an estimate of the execution time as a range from minimum execution time required to maximum execution time required. The scheduler will then choose tasks to run based on which task has the smallest laxity. Laxity is calculated as follows:

$$laxity = deadline\_time - present\_time - cpu\_time\_still\_needed$$

When more CPU power is requested from all tasks than is available, one or more tasks may fail to meet their deadline, in which case the CHIMERA II kernel automatically calls a failure handler. The failure handler is specified by the user on a per-task basis. Typical uses of the failure handler include the following: aborting the task and preparing it to restart the next period; sending a message to some other part of the system to handler the error; performing emergency handling, such as a graceful shutdown of the system or sounding an alarm; maintaining statistics on failure frequency to aid in tuning the system; or in the case of iterative algorithms, returning the current approximate value regardless of precision.

One important criterion with a deadline scheduler in an overload situation is selecting which tasks to run and which to let fail. The user not only specifies the failure handler, but also a failure handler priority. The failure handler can thus run at higher or lower priority than all other tasks in the system.

The CHIMERA II scheduler will use a *highest-priority-first* to distinguish between multiple tasks with equal laxity, and when there are no ready-to-run tasks with deadlines. Equal priority tasks are scheduled in a *round-robin* manner. There may be times when a non-deadline high priority task is extremely important and must execute ahead of any other task, even if the other tasks have deadline. The kernel allows tasks to set their *criticalness* to a value higher than the default value, which will then allow that task to run ahead of any task with a deadline.

Use of the deadline scheduler is optional. Some applications are better programmed just using a highest-priority-first scheduler as is available in commercial real-time operating systems. By not specifying any task deadlines within the system, the CHIMERA II scheduler behaves as a highest-priority-first preemptive scheduler. In such cases, however, there is no way to specify a failure handler since the tasks do not have deadlines.

3.3 System-Level Communication

One of the major strengths of CHIMERA II is that all the details of the hardware environment are transparent to the user. For example, to perform work with a file system, such as opening the file, reading it, then closing it, the user uses the standard UNIX syntax of *open(), read(),* and *close()*, or optionally using the C standard I/O *(stdio)* library. The CHIMERA II system determines which processor owns the file, sends the appropriate message, performs the operation, then returns the result. The user is thus unaware that any file operation is remote.

Every processor, including the host workstation, has a server task and an *express mail* mailbox which are used to provide transparent access to the remote devices and the host file system from any other RTPU. The server is capable of translating symbolic names into pointers, performing any necessary address calculations to account for the various address spaces and offsets within the multiple-VME-bus system, and performing system calls on behalf of remote tasks.

A low-overhead message passing mechanism, which we call *express mail*, is used to send system messages to the servers. System messages are messages sent only by built-in kernel routines, as opposed to user messages as described in Section 3.4. Any task can (indirectly) send a system message; however, only the servers can read these messages.

Many UNIX system calls, including *open(), close(), read(), write(), mmap(),* and *ioctl()* have been emulated as C procedures, with the ability to send messages when the required resource belongs to a remote processor. Whenever these calls have to access a remote processor, a message is sent to the remote processor's *express mailbox*. Each RTPU has at least one mailbox, which is in a part of memory known to all other processors. A server task on the remote process handles all incoming messages by performing the system call on behalf of the originator's task. Pointers to the data blocks to be read or written are passed as part of the message, as opposed to including the entire data in the message. This guarantees short messages and no additional buffering, which is especially important in calls such as read

and write, where the data can be sizeable. After the server completes the system call and all data to be returned has been placed into the originator's memory, an express mail message is returned to the originator, with the return value of the system call included within the message. Upon failure of a system call, the standard UNIX *errno* is also returned.

3.4 User-level Interprocessor Communication

CHIMERA II's express mail facility can only be used by system utilities. However, it provides the underlying facilities required for setting up shared memory segments, remote semaphores, and message passing among different processors in the real-time environment.

Any task can create or attach to a shared memory segment using a single procedure call. When any other task wants to attach to that segment, a procedure call is made which returns a pointer, with all address conversions performed, so that the user can use the pointer as though the memory was local. Even tasks running on the host workstation can access the shared memory segment transparently. This method not only makes accessing the global shared memory transparent to the user, but also provides a speed limited only by the VME bus, since there is no operating system overhead involved in accessing the shared memory.

Semaphores are created in the same manner as a shared memory segments, except that instead of a shared memory pointer being returned, a pointer to a semaphore structure is returned. That pointer is used in subsequent semaphore operations, which are functionally equivalent to their local counterparts, but can be used remotely across processors. These semaphores can be used both to control access to critical sections and to synchronize multiple tasks on different RTPUs.

There are times when interprocessor communication is better performed via messages than by using shared memory and semaphores. CHIMERA II also provides a user-level message passing where the user has the ability to specify the size of the queues and messages, the location of the queues, the type of the messages, and the priority of the messages.

All of these interprocessor communication primitives provide a level of transparency to the user. The user can send a message, perform a semaphore operation, or access some part of global shared memory without having to specify its physical location within the system. All of these facilities also offer the ability to communicate over a secondary local bus (e.g. the VSB bus) if one exists. The result of CHIMERA II's interprocessor communication features is much faster development time of applications requiring multiple processors and possibly multiple VME buses.

3.5 Host Interface

The entire hardware configuration requires at least one workstation to operate as host. The host provides access to the tools needed to compile and download programs to the real-time environment, and acts as a file server to those tasks requiring file access.

On the host workstation, three categories of processes are executing: the server process, console processes, and user processes. These processes communicate via shared memory and semaphores using the same interface as described in section 3.4. These facilities are a front end to the interprocessor communication supplied by the host UNIX operating system. The server process provides similar functionality as the servers on the RTPUs, except that it can access the host file system directly, and it includes special primitives to support the console processes. The *console process* provides the user interface which serves to download and execute programs on the RTPUs. The console process also provides the *stdin, stdout,* and *stderr* files for tasks executing on the RTPUs. Only one console process is needed to control an entire CHIMERA II system. However, if multiple RTPUs use *stdin*, only one of them can have it active at any given time. Other tasks attempting to read from *stdin* would block, and send a *Waiting for TTY Input* message to the user's terminal, similar to what UNIX does with background processes. If multiple RTPUs require *stdin* simultaneously, then multiple instances of the console process can be created, each on a separate terminal or within a separate window on the host workstation.

User processes are just like any other UNIX process running on the host, except that an additional CHIMERA II library is linked in, allowing the process to use the interprocessor communication package. These processes can then read from and write into the memory of any real-time processor, send or receive messages, or synchronize using remote semaphores. This feature allows users to create their own custom user interfaces, which possibly include the graphics or windowing facilities offered by the host workstation.

4 Performance

The success of a real-time operating system in developing robotic applications is based both on the ease of use and on its performance. Writing code to run under the CHIMERA II environment is as simple as writing a C language program to run under UNIX. As for the performance, several critical operating system features were timed to provide a general idea of the capabilities of CHIMERA II. The timings shown in Table 4 were performed on Ironics model IV3220 RTPUs, each with a 20 MHz M68020 CPU and a 20MHz M68882 floating point coprocessor (FPU).

Table 1: CHIMERA II Performance

Context Switching:	
Timer-driven reschedule, no task swapping	28 μsec
Timer-driven reschedule, with context switch	94 μsec
Resource contention context switch	85 μsec
Interrupt Latency (for handlers written in C:)	
VME Interrupt	28 μsec
Mailbox Interrupt	33 μsec
Express Mail:	
32-byte message	87 μsec
Local Semaphores:	
P() operation, no blocking	7 μsec
P() operation, blocking	92 μsec
V() operation, no waking up	7 μsec
V() operation, waking up a task	30 μsec
Remote Semaphores:	
semP() operation, no blocking, semaphore onboard	20 μsec
semP() operation, no blocking, semaphore offboard	22 μsec
semP() operation, blocking	add 85 μsec
semV() operation, no waking up, semaphore onboard	23 μsec
semV() operation, no waking up, semaphore offboard	25 μsec
semV() operation, waking up onboard task	add 30 μsec
semV() operation, waking up offboard task	add 63 μsec
Shared Memory:	
Overhead for reading from shared memory	0 μsec
Overhead for writing into shared memory	0 μsec

Possibly the most important performance criterion of a multitasking kernel is the reschedule and context switch time. Upon the expiration of a time quantum, a timer interrupt is generated, forcing a *time-driven* reschedule. If the currently running process has the minimum laxity or is highest priority, it remains running, and the timer interrupt results in no task swapping. The scheduling time in this case is 28 μsec, resulting in a peak CPU utilization of 97 percent with a one millisecond time quantum. At the other extreme, a full context switch is needed, which performs in 94 μsec, providing minimum CPU utilization of 91 percent for CPU-bound jobs. A full context switch involves suspending the current task by saving its entire context, including the FPU registers, selecting the next task to run using the minimum-laxity-first deadline scheduler, and resuming that task by restoring its entire context. Note that saving and restoring the FPU registers accounts for over half the context switch time alone. Although it is possible to improve the context switch time for tasks which do not use the FPU, it was decided that most control tasks use at least some floating point operations. Instead of keeping track of when floating point operations were used, the full FPU context is saved and restored at each context switch. A context switch arising as a

result of the running process blocking due to resource contention does not have to recalculate any dynamic priorities; it takes 85 μsec. Worst case CPU utilization can be increased to over 99 percent by increasing the time quantum to 10 milliseconds, which can be done for all but the servo level tasks which must run at frequencies typically above 100Hz.

The interrupt latency for the highest priority VME interrupt is 28 μsec for routines written in C. Shorter latencies are possible for exception handlers written in assembly language, but no performance timings are available. The savings are estimated at 8 μsec. *Mailbox* interrupts have a latency of 33 μsec.

For the express mail facility, sending a 32-byte message (a *typical* system level message) from one RTPU to another (or to the host) takes 87 μsec. No timings are available yet for user-level message passing, although it is expected to be only a few microseconds slower than the express mail.

The local *P()* (wait) and *V()* (signal) semaphores each execute in 7 μsec when there is no blocking or waking-up occurring. A blocking process adds the time of the resource contention context switch before the next process begins executing. The *V()* operation takes an additional 23 μsec to wakeup a blocked process.

Inevitably, remote semaphores take longer than local semaphores. A non-blocking *semP()* remote semaphore operation takes 20 μsec if the semaphore is physically stored on the same board as the executing task performing the operation. The same operation takes only 2 μsec more if the semaphore is off-board. If the *semP()* operation results in blocking the running task, then the time of the resource contention context switch is also needed before the next task can execute. The *semV()* remote semaphore wakeup operation takes 23 μsec for an onboard semaphore and 25 μsec for an offboard semaphore, if no tasks are to be waken up. Waking up a blocked task with the remote *V()* operation which is on the same RTPU takes an additional 23 μsec, while waking up an offboard task takes another 33 μsec to send a mailbox interrupt to the proper RTPU.

There is no software overhead involved in accessing shared memory. The speed of shared memory transfers across the VME bus are limited only be the hardware. As a result, interprocessor shared memory is by far the fastest means of communication among tasks within a multiprocessor system.

Note that the times above are subject to small variations as the code becomes more developed. For example the ready queue is currently not sorted. It is expected that speed increases of 5 to 10 percent per reschedule can be achieved with such modification. On the other hand, adding new features may increase the length of some commands by a small percentage. Nevertheless, the timing measurements provide a fairly good representation of the performance of CHIMERA II. The high performance of CHIMERA II allows it to maintain over 90 percent CPU utilization for control tasks running up to 1000 Hz, thus allowing it to be used even with the most computational and time demanding servo levels of a hierarchical architecture.

5 Summary

When implementing a real-time robotics system, too much time is typically spent with low-level details to get the hardware to work, as opposed to higher level applications which allow the system to do something useful. CHIMERA II is an operating system and programming environment that adds a layer of transparency between the user and the hardware by providing a high-performance real-time kernel and a variety of communication features. The hardware platform required to run CHIMERA II consists entirely of commercially available hardware, and the design allows it to be used with almost any type of VME-based processors and devices. It allows radically differing hardware to be programmed using a common system, thus providing a first and necessary step towards the standardization of robotic systems. This results in a reduction of development time and an increase in productivity.

6 Acknowledgements

The research reported in this paper is supported, in part, by U.S. Army AMCOM and DARPA under contract DAAA-2189-C-0001, by NASA under contract NAG5-1091, by the Department of Electrical and Computer Engineering, and by The Robotics Institute at Carnegie Mellon University. Partial support for David B. Stewart is provided by the Natural Sciences and Engineering Research Council of Canada (NSERC) through a Graduate Scholarship.

References

[1] D. E. Schmitz, P. Khosla, R. Hoffman, and T. Kanade, "CHIMERA: A real-time programming environment for manipulator control," in *1989 IEEE International Conference on Robotics and Automation*, (Phoeniz, Arizona), pp. 846–852, May 1989.

[2] D. B. Stewart, D. E. Schmitz, and P. K. Khosla, "CHIMERA II: a real-time multiprocessing environment for sensor-based robot control," in *In Proc. of IEEE International Symposium on Intelligent Control 1989*, (Albany, New York), pp. 265–271, September 1989.

[3] J. F. Ready, "VRTX: a real-time operating system for embedded microprocessor applications," *IEEE Micro*, vol. 6, pp. 8–17, August 1986.

[4] *Extended iRMX II.3*. Intel Corporation, Santa Clara, CA 95051, 1988.

[5] L. Kirby, "Real-time software controls mars rover robot," *Computer Design*, vol. 27, pp. 60–62, November 1 1988.

[6] S. Narasimhan, D. Siegel, and J. Hollerbach, "CONDOR: a revised architecture for controlling the Utah-MIT hand," in *Proceedings of the IEEE Conference on Robotics and Automation*, (Philadelphia, Pennsylvania), pp. 446–449, April 1988.

[7] L. Salkind, "The SAGE operating system," in *1989 IEEE International Conference on Robotics and Automation*, (Phoeniz, Arizona), May 1989.

[8] D. Green, R. Liscano, and M. Wein, "Real-time control of an autonomous operating system," in *Proceedings of the 1989 IEEE International Symposium on Intelligent Control*, (Albany, New York), pp. 374–378, September 1989.

[9] J. S. Albus, H. G. McCain, and R. Lumia, "NASA/NBS standard reference model for telerobot control system architecture (NASREM)," NIST Technical Note 1235, 1989 Edition, National Bureau of Standards, Gaithersburg, MD 20899, April 1989.

[10] T. Kanade, P. K. Khosla, and N. Tanaka, "Real-time control of the CMU Direct Drive Arm II using customized inverse dynamics," in *Proceedings of the 23rd IEEE Conference on Decision and Control*, (M. P. Polis, ed.), (Las Vegas, NV), pp. 1345–1352, December 12-14, 1984.

[11] D. E. Schmitz, P. K. Khosla, and T. Kanade, "The CMU Reconfigurable Modular Manipulator System," in *Proceedings of 18-th ISIR*, (Australia), ISIR, 1988.

[12] V. Feliu, K. Rattan, and H. B. Brown, "Adaptive control of a single-link flexible manipulator in the presence of joint and friction and load changes," in *1989 IEEE International Conference on Robotics and Automation*, (Phoeniz, Arizona), May 1989.

[13] H. Tokuda, J. W. Wendorf, and H. Wang, "Implementation of a time-driven scheduler for real-time operating systems," in *In Proc. 8th IEEE Real-Time Systems Symposium*, pp. 271–280, December 1987.

[14] W. Zhao, K. Ramamritham, and J. A. Stankovic, "Preemptive scheduling under time and resource constraints," *IEEE Transactions on Computers*, vol. C-36, pp. 949–960, August 1987.

[15] D. B. Stewart, *CHIMERA II: A Real-Time UNIX-Compatible Multiprocessor Environment for Sensor-Based Robot Control*. Master's thesis, Dept. of Electrical and Computer Engineering, Carnegie Mellon University, December 1989.