Chimera 3.0

The Real-Time Operating System for Reconfigurable Sensor-Based Control Systems

David B. Stewart

Pradeep K. Khosla

Advanced Manipulators Laboratory

The Robotics Institute

Carnegie Mellon University

Supported by U.S. Army AMCOM and DARPA under contract DAAA-2189-C-0001

Carnegie Mellon

Primary Authors of Chimera 3.0:

David B. Stewart, Ph.D Student, Department of Electrical and Computer Engineering, Carnegie Mellon University. MS-ECE Dec. 1989 from Carnegie Mellon University, Pittsburgh, PA; B.Eng Jun 1988 Concordia University, Montreal, Canada. David has been with the project since January 1989,

Pradeep K. Khosla, Associate Professor, Department of Electrical and Computer Engineering, and Director of Advanced Manipulators Laboratory, The Robotics Institute, Carnegie Mellon University, Pittsburgh, PA.

Other Contributors:

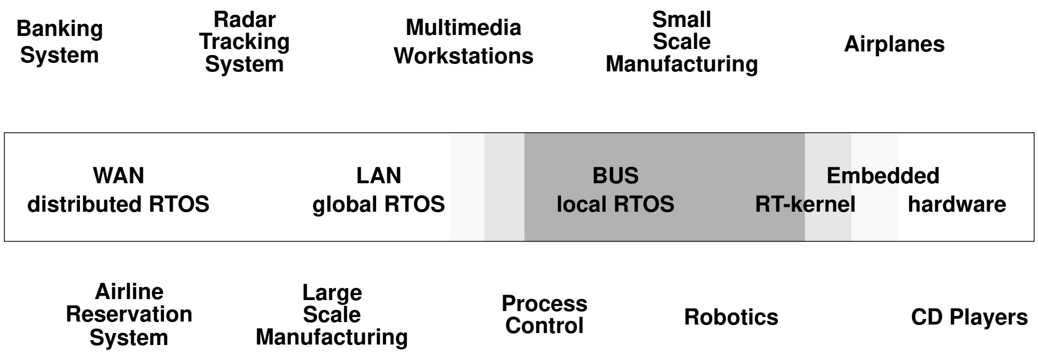
Don Schmitz, was a co-author of both Chimera I (1986-1989) and Chimera II (1989-1990). Since February 1990 he has been with Transarc Corp. in Pittsburgh, PA. Pradeep has been principal investigator of Chimera since its conception in 1986.

Matthew W. Gertz, a Ph.D. student at Carnegie Mellon University, joined the Chimera project in 1991, and is the primary developer of *Onika*, a generic graphical user interface for reconfigurable systems.

Kurt Schaeffer is an undergrad research programmer who has helped in the development of Chimera 3.0 since June 1992.

Richard Volpe has been a consultant throughout the history of Chimera (1986-present) by using it extensively, and establishing many of the requirements for a real-time operating system ideal for robotics. Richard received his Ph.D. in Physics from Carnegie Mellon University in 1990. He is now a research scientist at the Jet Propulsion Laboratory, California Institute of Technology, Pasadena CA.

Real-Time Spectrum



Carnegie Mellon

Shaded area shows target application areas of our work

There has not been any single definition agreed upon to define what is a real-time system; and probably there never will be. It is because different applications may have a different view as to what the words “real-time” really mean. For purposes of defining the scope of our work, we assume that a real-time system is one in which the time to execute a program is of concern, as it affects the outcome, either in terms of functionality or minimum performance requirements.

A real-time system may require one or more of the following characteristics: hard deadlines, soft deadlines, fast response time, and/or deterministic response. The large-scale real-time systems, which are distributed over wide-area-networks (WANs) tend to require fast response time for selected operations. The smaller scale real-time systems, such as those that execute in local area networks (LANs) or within an open-bus hardware architecture environment, may require fast response time, but also tend to have more periodic tasks with soft deadlines. These deadlines should usually be met, but missing them occasionally does not seriously impact the system. The smaller more dedicated real-time systems often have hard deadlines for critical tasks, and missing a deadline can have a major impact on the performance of a system. The most critical tasks are often implemented strictly in hardware, as it provides the most reliable system. However, as systems become more complex, designers look towards software to take on the additional burden, hence much of today’s research in real-time systems is towards meeting hard deadlines with software systems.

In our research at the Advanced Manipulators Laboratory, we are not concerned as much with hard real-time systems, but with intelligent sensor-based control systems, which may have a few hard real-time tasks, but most of the system can be considered soft real-time. Our target applications are robotics, process control, and manufacturing, as shown by the shaded area of the real-time spectrum. Instead of concentrating on all tasks meeting deadlines at all times, we concentrate on obtaining the complex functionality required by the target applications, by designing a software architecture that is robust to the occasional missed deadline, and provides a direct mapping of that design into the target open-architecture hardware setup. A result of that work is the Chimera 3.0 Real-Time Operating System for Reconfigurable Sensor-Based Control Systems.

In this talk, the concept of a reconfigurable system is first described, followed by details of Chimera 3.0.

Goal

- Desired Programming Interface
  - user-friendly easy-to-use interface
  - sequence jobs to create an application
  - reusable software modules configured to create jobs
- Before Execution: Automatic Integration
  - automatic interfacing of modules
  - global off-line scheduling to map software to hardware
  - configuration verification
- During Execution
  - fault detection and handling
  - local real-time scheduling
  - dynamic reconfiguration

Carnegie Mellon

The difficulty in programming intelligent sensor-based control systems, such as advanced robotic systems, has prevented their widespread use in applications such as manufacturing, space exploration, hazardous material handling, and everyday maintenance chores. Currently, a high level of education is required to use these systems, as the user must have combined experience in computers, sensors, device drivers, real-time systems theory, mechanics of manipulators, control systems, and several other specialty areas. In order for these systems to be used more effectively, the complexity of the systems must be hidden from the user, with a straightforward way of programming them with very high level commands.

An example scenario for using a robotic system follows:

> I approach a graphics workstation and login. A user-friendly interface starts up, and I can select to run any existing application, or build a new one. Suppose I select to build a new application. A library of jobs that can be performed by the system is displayed in a window. These jobs include picking up and putting down objects, moving to different locations, searching for objects, finding flaws in objects, sorting components from a box, and many other useful tasks. Suppose I want to build an application to clean my room. I select the sequence of operations *search for object, pick up object, if object is garbage then move to trash can, otherwise move to closet, put down object, repeat*. I then save this program, and tell the system to *go*.The system first checks out the application I have programmed, and identifies any problems, such as missing hardware or software modules, potential and probable timing errors of real-time tasks, possible conflicts when sharing resources, and improper interfacing of software modules. As an example, I have a *pick-up* operation, which means my robot must have a gripper. If it doesn't, then there is a hardware problem that needs to be fixed, and the system tells me about the missing hardware components. If everything checks out, then the system begins to execute the application, and I have confidence that the robotic system executes the application as I programmed it.

The work described in this talk represents steps in achieving a system as described above.

Approach

- Design underlying software as a *Reconfigurable System*
  - Static Reconfigurability
  - Dynamic Reconfigurability
  - Modular and Reusable Software
    - must be developed independent of target application
    - must be developed independent of target system configuration
    - software modelling
    - specifications for module decomposition and interfacing
  - Automatic Integration

Carnegie
Mellon

A reconfigurable system must support multiple configurations. The ability to change from one configuration to another falls into two broad categories, both of which are supported by our new architecture. **Static reconfigurability** involves changes to a system off-line. It calls for the interchangeability of both hardware and software, without the need to reprogram the system. **Dynamic reconfigurability** involves run-time changes to a system without halting the system. Such systems may have many sensors, actuators, and controllers, only a subset of which are used at any time.

The following issues then deal with the static and dynamic reconfigurability of the system:

For *static reconfigurability*, we are concerned with the correctness of each configuration, based on the input and output connections, timing constraints and resource requirements.

For *dynamic reconfigurability,* we are concerned with how we can predictably perform the transition from *configuration i* to *configuration j*, while retaining the integrity of the system at all times.

Modular and reusable software*:* A consistent interface must be designed which ensures that modules can be developed and implemented *independent of the target application*, since neither the final application nor software configurations are known *apriori*. Modules must also be designed and written *independent of the target hardware*, since the final hardware configuration is not known *apriori*. All interprocessor communication must be transparent to the processors, and hardware independent interfaces must be designed for all special purpose processors, sensors, actuators, I/O devices, and other sub-systems. To achieve these requirements, software models of each module are required, and generic specifications are needed for the integration of these modules.

Automatic integration: Given an application, automatically determine the software modules and configurations required, and map into the available hardware. If for any reason the integration cannot be performed (such as insufficient CPU power or unavailable devices) then an error is reported to the user so that the problem can quickly be rectified. These errors are reported *before power is turned on to the system*. If the system can be successfully integrated, then all necessary steps to download and execute code are performed automatically, and the application will execute correctly. Intermediate levels of integration are possible, where most of the system can work, but there may be some problem areas; the system will alert the user with appropriate warnings, and the user can then decide whether to proceed with or abort the application.

Motivation for Designing Reconfigurable Systems

- **Primary Motivations:**
  - **required to support reconfigurable hardware**
  - **allows fixed configuration hardware to support multiple applications**
- **Other Motivations**
  - **forces development of reusable software**
  - **allows for rapid development of applications**
  - **produces a flexible and expandable system**
  - **enables systematic development of complex systems**
  - **reduces development costs**
  - **simplifies technology transfer**

Carnegie Mellon

In addition to achieving the main goal described in a previous slide, there are many other reasons for developing a reconfigurable software architecture, and a real-time operating system to implement such an architecture.

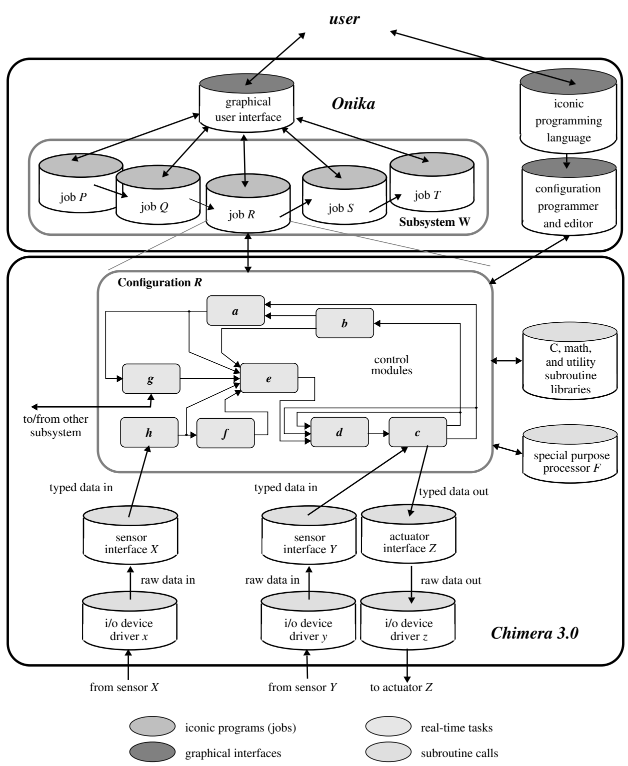
The primary motivations for our work include the following:

- Reconfigurable hardware, such as open architecture computing environments and the CMU Reconfigurable Modular Manipulator System (RMMS), require reconfigurable software in order to take full advantage of all the capabilities of the hardware.
- Reconfigurable software is useful for supporting multiple applications on any given fixed hardware configuration.

There are also many other advantages to designing reconfigurable software, even for systems which do not have to be reconfigurable. These other motivations include the following:

- In order for software to be reconfigurable, it must be reusable. Therefore any software designed using a reconfigurable software framework can be quickly and easily reused in other projects.
- Because software can be reused from other applications, it increases the speed in which new applications can be developed.
- A requirement for reconfigurable systems is that a new module can be added to an existing system, without changing any code within the existing modules. This requirement means that a reconfigurable system can be expanded very easily just by adding new modules; there is no need to modify existing code.
- Reconfigurable software is inherently modular and can easily be decomposed into one or more subsystems. This eases the development of complex systems.
- Because of the software reuse, modularity, and ease of expandability, new applications can be developed rapidly, and hence development costs are reduced, as compared to developing a new application from scratch.
- Since modules are developed independent of the target application and independent of the target hardware, a module (and hence the technology implemented within that module) can easily be transferred to another institution which is also using the reconfigurable software framework. Technology transfer is thus straightforward, even if different institutions are working on very different applications or have very different hardware setups.

Software Framework for Reconfigurable Control Subsystems



A *reconfigurable control subsystem* is defined as a sequence of *jobs*, which can be programmed by a user. Each job corresponds to a predefined *configuration*, and has a set of pre-conditions and post-conditions. If both the post-conditions of the current job and the pre-conditions of the next job in the sequence are met, then a dynamic reconfiguration can be performed within the subsystem.

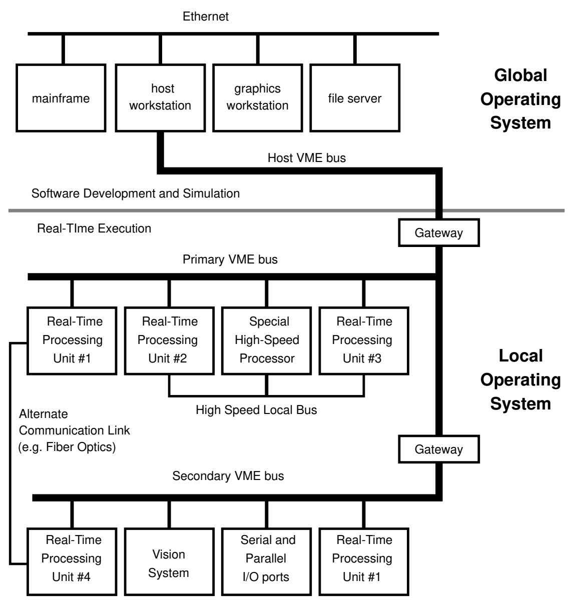
An *application* is defined as one or more subsystems executing in parallel. These subsystems include both reconfigurable control subsystems and other subsystems, such as dedicated vision systems, path planners, and expert systems.

We define a *control module* as a reusable software module within one or more configurations. A reconfigurable system consists of many control modules, each of which can be modelled using a simplified version of a port automaton. Each module has zero or more *input ports*, and zero or more *output ports*. Each port corresponds to a data item required or generated by the control module. A control module can also interface with other subsystems, such as vision systems, path-planners, or expert systems. (For more details, see Stewart, Volpe, and Khosla, "Integration of Real-Time Software Modules for Reconfigurable Sensor-Based Control Systems," in *Proc. of 1992 IEEE/RSJ Int'l Conf. on Intelligent Robots and Systems (IROS '92)*, Raleigh NC, July 1992.)

Control modules can communicate to sensors and actuators through a generic sensor/actuator interface, which in turn communicate to the hardware via device drivers; they can communicate to floating point accelerators, image processors, transputers, or other special hardware through a generic special purpose processor interface; or they can communicate to other subsystems using any of the communication mechanisms available in the real-time operating system.

In this talk we will concentrate on the real-time operating system (RTOS) required to support such a software framework for reconfigurable control subsystems (herein referred to simply as "reconfigurable systems").

Typical Open-Architecture Hardware Configuration



Carnegie Mellon

We concentrate on **open architecture hardware configurations**. These systems are designed for easy expandability. They generally have some form of common bus, such as a VMEbus, Multibus, etc. Processor and input/output (I/O) hardware can be added or removed from the system by adding or removing commercial or custom hardware boards. The real-time operating system (RTOS) executes on the real-time processing units (RTPUs), which are generally single-board-computers (SBCs) with a general purpose CPU.

Open architecture systems are contrasted by *embedded systems*, such as found in automatic teller machines, airplanes, CD players, etc. Embedded systems have a finite set of functions, and are not easily expandable. They are generally packaged up, with a limited number of inputs. Our research *does not* focus on these fixed-configuration embedded systems.

An open-architecture system also differs from a *distributed system*, which generally implies some form of local or wide area network, in which execution of programs may execute at any node, and storage of data may be at any node. An open-architecture system is generally a single node within a distributed system. Our work concentrates on the software and hardware architectures of that single node, when that node is used for real-time sensor-based control.

A **global operating system** executes on the host workstation. It provides a development environment for the local node, and communication with other nodes within a network.

A **local operating system** executes on the RTPUs which are accessed through the host workstation. It provides the utilities required to manage the local hardware, and the communication with the global operating system. When hardware must be accessed in real-time, then the local operating system must be real-time. Although it may be desirable, it is *not* necessary that the global operating system also be real time.

Chimera 3.0 is a local RTOS, and is currently designed to work in conjunction with the SunOS (UNIX) global operating system.

Note that some commercial RTOS, such as OS-9, LynxOS, and iRMX, combine the functionality of both the global and local operating systems into one. As a result, these systems generally have both inferior development environments, since they don't have the software base of most UNIX systems, and an inferior real-time functionality, since much of their effort goes into developing the environment. Several commercial RTOS which do operate as local operating systems, such as VRTX and VxWorks, have generally been more successful in gaining a wide user-base because of their better programming environment and enhanced real-time functionality..

Chimera: (ki mir' e) [Gr. chimaira, she-goat] 1. Gr. Mythology: A fire breathing monster with lion's head, goat's body, and serpent's tail. 2. An impossible or foolish fancy.

History of Chimera:

During the development of the Reconfigurable Modular Manipulator System (RMMS) in 1986, real-time software support was required to rapidly develop and test code for the RMMS. At that time, commercial real-time operating systems did not exist. Experience with the CMU Direct Drive Arm II (DDArmII) system from the previous years indicated that without a programming environment hours and hours of development time led to only a few seconds or minutes of actual test code execution.

In an attempt to create some kind of usable environment, Don Schmitz and Regis Hoffman developed the CHIMERA real-time kernel. It supported only a single processor on a VME backplane. It had a very primitive user interface. Nevertheless, it was a significant improvement over not having any environment at all. The name "CHIMERA" was selected not so much because Don and Regis wanted to name their "monster" project after a monster, but in jest of the second definition of *chimera*, which is "An impossible or foolish fancy." Although the real pronunciation is *ki mir' e*, in time the pronunciation of our system has become *kim-e ra'*.

Early in 1989, the DDArmII system was upgraded to be VMEbus-based, and the CHIMERA real-time kernel installed. The DDArmII was to be the testbed for research in multi-sensor integration. Although CHIMERA improved the environment of the DDArmII, many problems still existed. The most significant was the fact that it did not support multiple processors, and communication mechanisms were virtually non-existent. Commercial operating systems such as VxWorks, VRTX, and pSOS were gaining popularity and were evaluated at that time, but they represented only an incremental improvement over our already existing CHIMERA system. We felt that building the multiprocessing and communication features on top of an existing commercial RTOS would be just as much effort as doing so based on our experience with CHIMERA; however, in the latter case we would have the freedom to incorporate any changes we want, instead of being left at the mercy of commercial object code. The completely new system was named CHIMERA II.

In designing the requirements for CHIMERA II, much emphasis was placed on the attempts in our lab to design reconfigurable systems. With the RMMS we were addressing reconfigurability at the hardware level. With CHIMERA II, we addressed the software needs of dynamically reconfigurable systems. We released a first version of CHIMERA II in June 1989.

As our research evolved, so did our real-time operating system, designed especially to support a software architecture for reconfigurable systems. Many of the features required to support reconfigurable systems are not available in any commercial RTOS, but have been incorporated into Chimera 3.0. Two projects at CMU began to use Chimera 3.0 in June 1992. Chimera 3.0 will be officially released in November 1992. The remainder of this talk concentrates on those features.

Thanks to Matthew W. Gertz for the Chimera logo.

Notes

Chimera 3.0

- Based on VMEbus, MC680X0 RTPUs, and Sun Host
- GNU C and C++ Cross Compiler
- Provides Tools for Building Reconfigurable Systems
  - real-time kernel
  - complete set of C, C++, and math libraries
  - multiprocessor communication support
  - host workstation interface
  - interface to low-level mechanisms
  - I/O support

Carnegie Mellon

Chimera 3.0 is based on VMEbus hardware, and is currently supported on Motorola MC680X0-based processors, and requires a Sun 3, Sun 4, or Sun SPARCstation host.

The GNU C and C++ compilers are used to generate the code to execute in the real-time environment.

Chimera 3.0 provides all of the traditional RTOS utilities, which are found in most commercial RTOS, and many extensions, especially for multiprocessor support and for the implementation of reconfigurable systems. Each of these items is described separately in the remainder of this talk.

Carnegie Mellon

Real-Time Kernel

- Multitasking preemptive kernel
- Local communication:
  - local semaphores
  - local shared memory
  - user signals
- Static and dynamic real-time scheduling
  - rate monotonic algorithm
  - earliest deadline first
  - maximum-urgency-first
  - user definable
- Supports any number of periodic and aperiodic tasks
  - no need to program timers
- Fault detection and handling
  - global error handling
  - deadline failure detection

The **real-time kernel** provides the basic run-time support for a real-time multitasking application, including the following primitives: *spawn(), kill(), exit(), abort(), qclock(), clock(), get_time_quantum(), pause(), set_deadline(), , clear_deadline(), sleep(), usleep(), delay(), set_failhandler(), kprintf(),* various setting/getting task priorities, and **local communication** features, including shared memory among all tasks on the RTPU, very fast local *P()* and *V()* semaphores (even faster than the VxWorks "damn fast semaphores"), and user definable signals allowing one task to interrupt the execution of another task on the same RTPU.

Most commercial real-time operating systems only support static highest-priority-first scheduling, which allows the implementation of a basic rate monotonic algorithm. For many dynamically changing applications, this is not sufficient. Chimera also offers **dynamic priority scheduling** in addition to the static priority scheduling available in other systems, allowing the implementation of either *earliest-deadline-first* or *maximum-urgency-first* scheduling (for more details, refer to Stewart and Khosla, "Real-Time Scheduling of Dynamically Reconfigurable Systems," in *Proc. of Intl. Conf. on Systems Engineering*, Dayton, Ohio, pp. 139-142, August 1991.) Due to the design of the Chimera kernel, any part of the kernel can be overriden by user code. This includes the scheduler, which allows users to implement their own real-time scheduling algorithm.

Unlike most commercial RTOS, where a separate timer is required for each periodic task, Chimera provides a **hardware independent layer to the system timers**, thus allowing real-time applications to remain hardware-independent. The interface makes it appear as though every task in the system has its own timer. The user only needs to specify its next wakeup time, and all handling of the timer is taken care of transparently by the kernel. In contrast, commercial RTOS require that a task with special timing constraints program the timer(s) itself, and go through complicated interrupt or signal handling schemes to handle the timer interrupt.

Chimera has a novel error detection and handling feature built-in to the kernel, which we call **global error handling**. It has been estimated that in some systems 80% of the code is for error checking and handling. The global error handling mechanism in Chimera removes the need for much of that code, as there is no need to constantly check the return values from system calls or handle processor exceptions. Instead, whenever an error occurs, an error signal is generated, which is handled by an error handler. Default error handling is to print a detailed error message, then abort the task. User-defineable handlers can be installed to override the default handling, either for a specific error or for a set of errors. No other RTOS has comparable functionality. This powerful feature can save hours of programming, and results in more robust code, as errors within the system are constantly monitored and handled, even if the user does not write explicit code to do so. This feature also provides separation of main body code and error handling code, which is a popular software engineering technique for developing modular systems, but usually only available in selective languages like *Ada* and *Exceptional C*.

Details of the **deadline failure detection** is in the next slide.

Possible Actions for Deadline Failure Handler

- User definable. examples include:
  - aborting task and preparing it to restart next period
  - emergency handling, such as
    - graceful shutting down of system
    - sending message to another part of system
  - modifying criticality or user priority of task
  - maintaining statistics on failure frequency
  - in case of iterative algorithms, return current approximate value

Carnegie Mellon

The Chimera kernel provides **deadline failure detection**, which calls a user-defined handler to be called whenever a task misses its deadline. Other RTOS strictly provide highest-priority-first scheduling which allows you to implement the rate-monotonic algorithm, but do not provide any support to handle tasks based on their execution and deadline times. Therefore, if a task misses a deadline in another RTOS, there is no way of handling that error, and the system continues as though nothing was wrong. Such lack of error handling is unacceptable for any real-time control system.

As an example of the possible actions for error processing, this slide gives various possible actions as a result of a deadline failure. The programmer has complete ability to define any actions based on the application.

Performance of Chimera Kernel vs. Commercial RT-Kernels

Test name	pSOS+	VRTX32	LynxOS	VxWorks	Chimera
Create/Delete Task	591	371	—	1423	420
Ping Suspend/Resume	128	142	—	177	190
Ping Semaphore	219	239	397	232	220
Get/Release Semaphore	63	55	74	33	12
Queue Fill	46	26	140	20*	66†
Queue Drain	43	29	132	22*	61†
Queue Fill/Drain	91	59	278	44*	120†
Queue Fill, Urgent	47	27	170	72*	67†
Alternate Queue F/D	238	252	867	371*	570†
Allocate Memory	40	27	57	68	42
Deallocate Memory	38	33	20	83	36

* VxWorks does not support message passing. These times are from simulating message passing through the use of shared memory.

† Chimera's kernel does not have built-in message passing. The timings are for local messages when using the multiprocessor message passing mechanism in the Chimera communication library.

The benchmark and times for commercial RTOS are taken from A. Topper, *A computing architecture for a multiple robot controller*, Masters Thesis, McGill University, Montreal, Canada, June 1991.

Create/Delete Task: This test measures the time it takes to create and delete a task. A task deletes itself as soon as it is created. The created task has a higher priority than its creator, so the time quoted actually includes a create, start, delete, and two context switches.

Ping Suspend/Resume Task. A low priority task resumes a suspended high priority task. The high priority task immediately suspends itself. This measurement includes two task context switches and the time it takes to suspend and resume a task. There is no facility to suspend/resume a task in LynxOS. In Chimera, local semaphores are used to suspend and resume tasks.

Ping Semaphore: Two tasks communicate with each other through semaphores. The time measured includes the following: task A gets a semaphore, signals task B, then blocks on another semaphore, task B wakes up, signals task A, then blocks, and then task A wakes up.

Getting/Releaseing Semaphore: the timer reportes includes the time it takes to get an available semaphore and immediately release it, within the same task context.

Queue Fill is the time to place a message into a message queue. *Queue Drain* is to remove it. *Queue Fill/Drain* is the combined time for a task to place a message in the queue then immediately retrieve it. The *Queue Fill Urgent* is a priority message insert. *Alternate Queue F/D* is the combined time for task A to send a message to task B, then block, task B receives the message, sends a message to task A, and A receives that message. Note that the times given may not be directly comparable, as the functionality of the message passing system and the size of messages for each system differs, and those differences are not documented in the referred thesis.

Allocate Memory is the time to allocate a block of memory using malloc(). *Deallocate Memory* is the time to release that block of memory using free().

Chimera is not necessarily the fastest of all RTOS, but in all cases its speed is comparable. The real strength of Chimera is in features not found in other RTOS, especially dynamic scheduling, global and deadline failure handling, interprocessor communication, and reconfigurable systems support.

Complete Set of C, C++, and Math Libraries

- concurrent stdio
- string manipulation and type conversion
- memory and bit field operations
- random number, searching, and sorting
- memory allocation
- time of day utilities
- command interpreter
- standard math library
- matrix math library
- configuration file reading utilities
- GNU libg++ standard library

Carnegie Mellon

Following is a list of library routines available with Chimera 3.0. Most routines have identical functionality to UNIX, although some have enhanced functionality to support the multiprogramming model of Chimera 3.0. The extensive set of libraries allows programs developed on a UNIX system to be integrated into the real-time system with relative ease.

Concurrent standard I/O: *fopen(), fclose(), fgetc(), getc(), getchar(), fgets(), gets(), fputc(), putc(), putchar(), fputs(), puts(), fread(), fwrite(), printf(), fprintf(), sprintf(), scanf(), fscanf(), sscanf(), fflush(), setbuf(), setbuffer(), setlinebuf(), setvbuf().* Because multiple tasks can share the same file descriptor, these routines all implement the necessary mutual exclusion to ensure data integrity.

String manipulation and type conversion: *strcat(), strncat(), strcmp(), strncmp(), strcasecmp(), strcpy(), strncpy(), strlen(), index(), rindex(), strchr(), strrchr(), strpbrk(), strspn(), strcspn(), abs(), strtol(), atoi(), strtod(), stof(), etc..*

Memory and bit field operations: *bcopy() bcmp(), bzero(), bfill(), ffs(), memcpy(), memccpy(), memcmp(), memset(), memchr(), bclr(), bclrb(), bset(), bsetb(), bchg(), bchgb(), btst(), btstb(), bfffo(), ror(), rol().*

Random number and sorting: *srand(), rand(), random(), srandom(), initstate(), setstate(), qsort().*

Memory allocation: *malloc(), calloc(), realloc(), free(), cfree(), mavail(), maxavail(), malloc_verify()* plus similar allocation functions for global shared memory segments.

Time of day utilities: *gettimeofday(), ctime(), asctime(), localtime(), gmtime(), timezone(), dtime().*

The **command interpreter** library makes it easy for a programmer to design a custom command-line interface. These utilities are not available in any other RTOS.

Standard math library: *sin(), cos(), sincos(), tan(), asin(), acos(), atan(), atan2(), exp(), log(), log1p(), pow(), sinh(), cosh(), tanh(), sqrt(), fabs(), isnan(), isinf(), ceil(), floor(), rint(), e(), pi().*

Matrix math library: this library provides a full set of matrix math operations, including such operations as *addition, multiplication, inverse, transpose, gaussian elimination,* and *copy*. This library is not available with any other RTOS.

The **configuration file reading utilities** make it easy for a programmer to design code to read in custom configuration files. These utilities are not available in any other RTOS.

The GNU libg++ standard library has been ported to Chimera 3.0, so that code written in C++ can use any of those libraries, including the standard streams library.

Multiprocessor Communication and Synchronization (slide 1 of 2)

- Chimera is a true multiprocessor RTOS
  - most commercial RTOS are single processor, and support multiple processors through inefficient networking only
- Summary of Features:
  - automatic address resolution
  - express mail for system communication
  - global shared memory
  - spin-locks
  - remote semaphores
  - priority-based message passing
  - user-definable VME and mailbox interrupts

continued …

Carnegie Mellon

The multiprocessor communication and synchronization facilities are some of the primary features which set Chimera 3.0 apart from commercial RTOS. We expect that the next generation of commercial RTOS will include many of the features already provided in Chimera 3.0, as they are required for developing reconfigurable software, and provide standard interfaces for often used communication facilities.

Most commercial RTOS are designed for single processors, and they obtain multiprocessing by connecting multiple processors through a backplane or network. The communication between processors is very slow, as they typically use sockets and the TCP/IP network protocol, even if communication is over the backplane. This communication is non-real-time and very limited, thus limiting the effectiveness of a multiprocessor setup for real-time control. Chimera 3.0 differs from these systems in that it is a true multiprocessor RTOS. It offers a multitude of features for various kinds of interprocessor communication and synchronization.

Automatic Address Resolution: every RTPU, memory board, special purpose processor, and I/O device card in a VMEbus has a unique address, in one of multiple address spaces. Each RTPU, in turn, addresses local memory and VMEbus memory differently. The result is that pointers that access anything other than local memory must continually add or subtract base offsets to access the same memory locations. Typically code must explicitly add and remove offsets when required, thus making code non-portable and/or forcing it to be recompiled, if it is moved from one processor to another, which is not acceptable for reconfigurable software. Chimera 3.0 provides automatic address resolution for all addressing in the system, allowing users to program multiple RTPUs totally independent of the VMEbus and local RTPU addressing.

Express mail is the system-level communication used by the Chimera kernel on one RTPU to communicate with the kernel on a different RTPU. The user does not have direct access to it, but all other multiprocessor communication is based on it. The express mail also allows a host workstation to take full control of the real-time environment, and provides the high speed communication between the host and real-time environment. No high-overhead TCP/IP connection is required between the non-real-time development environment and the real-time execution environment.

Global shared memory allows tasks on different RTPUs to share memory segments with *no* operating system overhead. The operating system performs all address offset calculations automatically during the initialization of the segment. As a result, software never needs to hard-code the physical addresses of memory boards or RTPUs, and hence software remains hardware independent, thus fulfilling one of the primary requirements of reconfigurable software.

continued …

Multiprocessor Communication and Synchronization (slide 2 of 2)

- Summary of Features (continued)
  - extended file system (host procedure calls)
  - remote procedure calls
  - special purpose processor interface
  - triple-buffered one-to-one communication
  - global state variable tables
  - multiprocessor subsystem task control

Carnegie Mellon

Notes for *Multiprocessor Communication and Synchronization*, continued:

The **spin-locks** provide a C-level interface to the *read-modify-write* instructions, which are the most basic form of interprocessor synchronization available. The Chimera utilities allow the polling time to be set for the spin-locks, and the user has control on whether or not a task swaps out if the lock is not obtained.

Remote semaphores are like the local semaphores, except that they can be shared by tasks executing on different RTPUs. Like the global shared memory, the interface to the remote semaphores is hardware independent.

The **priority-based message passing** facility allows any task in a system to send a message to any other task in the system, using a very convenient *send* and *receive* interface. The messages may be typed, thus allows arbitrary sized messages with arbitrary functions to be sent via the same message queue. In addition, messages are prioritized, either as *first-in-first-out, last-in-first-out*, or *highest-priority-first out*. The priority mode is user selectable.

Although interrupts are usually reserved for device drivers, the user has the ability to send and receive both **VMEbus IRQ interrupts** and **mailbox interrupts**. The Chimera interface is hardware independent, and removes the need for the programmer to setup any hardware-specific registers to send or handle these interrupts.

The **extended file system** allows a task on any RTPU to access the file system and environment of the host workstation. When necessary, a **host procedure call** is performed, in which the execution of the specified command is executed remotely on the host, even though it appears to be a local call on the RTPU. This allows the RTPU and host to share the same file systems, and hence the same files. The RTPU also has access to environment information, such as the host's date and time, environment variables, and current working directory. Conversely, the host workstation has access to the real-time environment. More details on the host workstation interface are given in a later slide.

The **remote procedure call** mechanism has two purposes. The first is to provide SPPs and subsystems not executing the Chimera kernel to execute system calls on the RTPUs in the same way an RTPU executes these commands on the host. The net result is that a command like *printf()* can be executed on an SPP, which in turn is executed on an RTPU, and in turn executed on the host workstation, so as to appear on the user's screen. In such a case a two-step communication is used to print from the SPP to the user's screen, but both steps occur transparently to the user. The second is to provide a user-definable mechanism in which the user can specify a set of subroutines which will transparently execute on a remote processor. The remote processor could be another RTPU, an SPP, or another subsystem. Note that for an SPP, the SPP interface can be used instead, as described next.

continued …

Notes for *Multiprocessor Communication and Synchronization*, continued:

Chimera 3.0 provides a **generic special purpose processor interface**, which allows a user to quickly and easily incorporate supported special purpose processors (SPPs) into their applications. SPPs include floating point accelerators, digital signal processors, image processing and vision systems, intelligent peripheral controllers, LISP machines, and transputers. The SPPs are treated as slaves in the system. They do not run the Chimera kernel, and hence have little operating system overhead. Instead, a small Chimera executive runs on the SPP, which handles incoming messages, and calls an appropriate subroutine to execute the user's code on the SPP. The generic SPP interface presents a standard for interfacing with SPPs. Traditionally every SPP would have its own unique interface, as designed by the manufacturers of that SPP. Interfaces from different manufacturers are usually very different and not compatible with each other, making it difficult for a user to develop code independent of the target hardware architecture, thus violating one of the major criterion for developing reconfigurable software. With this generic interface, a user can write code which calls an algorithm on a remote SPP, and will retrieve the results of that computation from the SPP, without any dependencies on the specifics of the SPP. The interface thus makes it possible to use SPPs in a reconfigurable system.

Triple-buffered one-to-one communication: situations often arise where one-to-one communication of a fixed amount of data is required. One task is the *producer*, and writes data into a shared memory location periodically, while the other task is the *consumer*, and reads from that shared memory location periodically. Such communication has often been implemented using a double-buffer technique. While the producer is writing the data into one buffer, the consumer can read the data from the other buffer. When both are finished, they swap buffer pointers, and can continue with the next buffer. This method insures that the producer and consumer are never accessing the same data at the same time. There are problems with this scheme, however. It requires that both tasks operate at the same frequency and be properly synchronized. Alternately, the tasks may be at different frequencies, but if those frequencies are not multiples of each other, then one task ends up constantly blocking while the other task tries to finish using the buffer. Another problem exists if there is a clock skew between the clocks of the two tasks, which can cause undesirable blocking between the tasks. A solution to this problem is to use three buffers instead of two. At any given time, there is always one buffer not in use. When one of the two tasks finishes with its buffer, it can immediately use the free buffer. This allows both tasks to execute independently, and not block. The Chimera triple-buffered one-to-one communication mechanism implements this triple buffering scheme, handling all the communication and synchronization through a simple *read* and *write* interface. The mechanism has also been designed to automatically handle the byte-swapping problems when big and little endian architectures must communicate. This allows special purpose processors, intelligent I/O devices, and other subsystems to use this mechanism in conjunction with a task running under Chimera on an RTPU.

Notes for *Multiprocessor Communication and Synchronization*, continued:

Global state variable tables: A state variable table mechanism has been developed to allow tasks on multiple RTPUs to share state information, and to update the state information quickly and correctly. This mechanism automatically creates one global table of state variables, and a local copy of the table for each task which requires access to the state variables. Tasks always make use of the local copy, with periodic updates (typically at the beginning and end of each periodic cycle) to and from the global table. Chimera 3.0 automatically handles all the communication and synchronization through a simple *read* and *write* interface. The global state variable table communication is a predictable mechanism, as it allows the exact calculation of the required VME bus bandwidth required when integrating multiple real-time control tasks, as required for analyzing static configurations in a reconfigurable system. (For more details, see Stewart, Volpe, and Khosla, "Integration of Real-Time Software Modules for Reconfigurable Sensor-Based Control Systems," in *Proc. of 1992 IEEE/RSJ Int'l Conf. on Intelligent Robots and Systems (IROS '92)*, Raleigh NC, July 1992.)

Chimera 3.0's most powerful feature for implementing reconfigurable systems is its **multiprocessor subsystem task control mechanism** (we call this mechanism *SBS* for short). A subsystem can contain one or more tasks, which must execute on one or more RTPUs. The SBS automatically handles the task creation, initialization, timing and error handling for each module and the synchronization and communication between multiple tasks within the subsystem. The programmer only has to write code specific to the module, without having to worry about interfacing the module with the rest of the system. This satisfies the reconfigurable systems criteria that *modules are designed independent of the target application.* The SBS also allows a module to execute on any RTPU in the system. Moving a module from one RTPU to another is trivial, as the subsystem task control mechanism *allows modules to be designed independent of the underlying target hardware architecture*, thus satisfying another major criterion for reconfigurable systems software design. The net result of the SBS is that new modules can be written in a matter of minutes or hours, instead of the days or weeks it would take to obtain the same functionality in most commercial RTOS. Chimera 3.0 also allows multiple subsystems to coexist, with each subsystem being able to either gain exclusive usage of the RTPU, or share an RTPU with another subsystem.

Host Workstation Interface

- Command-line Interface to All RTPUs
- Communication to RTPUs over VMEbus
- Supports user-defined graphical interfaces
- Appears as an RTPU for Interprocessor Communication
- Isolation of RT and non-RT with VME-VME adaptor
- Extended File System
- Real-Time Environment == Simulation Environment
- Access to RTPU Built-in Monitor plus Enhancements

Carnegie
Mellon

The **host workstation** is an integral part the Chimera 3.0 Real-Time Programming Environment. Unlike many commercial real-time operating systems which attempt to duplicate the UNIX development environment, Chimera 3.0 takes advantage of all the tools available, and makes use of them. This includes the editors, compilers, debuggers, window managers, file systems, and networks.

All RTPUs can be controlled by a single **command-line interface**, thus eliminating the need to have a separate window or VT100 terminal open for every RTPU. However, each RTPU may have its own window if that proves to be more convenient for a given application. The command-line interface provides the ability to download, execute, and monitor programs on the RTPUs. The interface is also used as the *standard input (stdin), standard output (stdout)* and *standard error (stderr)* for programs running on the RTPUs.

The **host communicates over the VMEbus to the real-time environment**, thus providing a high-speed parallel real-time link between the host and RTPUs. This setup also allows the host to become an integral part of the real-time programming environment, by appearing to the RTPUs as another RTPU. Thus **processes on the host can communicate with any tasks on the RTPUs** using the global shared memory, remote semaphore, and message passing interprocessor communication mechanisms previously mentioned. These processes can also make use of the host's graphical environment, thus allowing programmers to develop application-specific graphical interfaces.

A **VMEbus-to-VMEbus adaptor** can be used between the host workstation and real-time environment in order to isolate non-real-time traffic (e.g. from disk reads/writes) from the real-time traffic.

The RTPUs do not have their own file system. Instead, **all file system operations are performed remotely** (and transparently) by the host workstation. Thus the RTPUs use the same file system as the host. There is no need to copy or FTP executable or data files between the host and real-time environments.

Real-time code development is done on the host workstation, and software modules are moved to the real-time environment only after they have been debugged, leaving only the interaction between modules to be debugged in real-time. This is possible because Chimera 3.0 provides the same libraries and communication primitives to both the real-time environment and the host workstation. This **allows the simulation and real-time environments to be nearly identical**.

Chimera 3.0 provides an **interface to the built-in monitor** of the RTPU, thus allowing the use of such monitors directly from the workstation, instead of requiring a separate terminal for every RTPU. In addition, the monitor provides additional features over an RTPU's built-in monitor, such as the automatic conversion of symbols to addresses and enhanced help menus.

Interface to Low-Level Mechanisms

- reentrant code
- setjmp and longjmp
- setting time resolution
- changing vector base register
- accessing CPU registers from C
- testing existence of memory
- locking out interrupts

Carnegie
Mellon

The low-level mechanisms are generally functions that will only be used by more experienced programmers when developing such things as exception handlers and device drivers. In order for these low-level routines to be completely written in C, Chimera 3.0 provides a C-language interface to a variety of utilities which normally would only be accessible by using assembly language.

Reentrant code is similar to subroutine calls, except that it is usually executed in response to an interrupt or signal, and thus must save the current processor state before executing, and restore the state afterwards. Chimera 3.0 provides a utility which allows the user to convert any subroutine into a reentrant subroutine, thus allowing interrupt and exception handlers to be written in C.

The **setjmp** and **longjmp** “global goto” statements are very useful for recovering from errors, exceptions, and deadline failures.

Chimera 3.0 uses a default 1 msec timer resolution. In order to reduce operating system overhead or increase timing resolution, the user can **change the timer resolution**. Possible values of this resolution are limited by the capabilities of the hardware timers available. If a particular resolution is not possible, the timer will be set to the closest supported resolution.

Changing vector base register: normally all tasks will share the same exception table. However, Chimera allows any task to make its own copy of the table. That task can then install its own exception handlers without affecting the default handlers for all other tasks in the system.

At times it may be desirable to **directly access the CPU registers** from C. A C-language interface is provided to read and write any CPU register. This feature should be used with caution, as improper changes to registers may conflict with compiled C code.

Chimera 3.0 provides an interface for **testing the existence of memory** on the VMEbus. This allows a task to ensure that hardware is properly responding without causing the task to terminate due to a bus error.

When executing time-critical or atomic code, it is possible for the user to **lock out interrupts.**

I/O Devices

- Any VMEbus-based I/O device can be used; e.g.:
  - serial ports
  - parallel ports
  - analog-to-digital and digital-to-analog
  - frame grabbers
  - memory mapped devices
  - serial buses
- UNIX-like device driver support
- Second level interface to sensors and actuators
- Transparent remote access to host workstation's devices:
  - file system
  - monitor (for *stdout)*
  - keyboard (for *stdin*)
  - network

Carnegie Mellon

As done with UNIX, device drivers are used to provide a hardware independent layer to I/O devices. All I/O devices can be read and written using the *read()* and *write()* [emulated] system calls. The hardware details, such as I/O registers, interrupt handling, and device addressing, are all taken care of by the device drivers and kernel. The library of supported device drivers is continually growing, and programmers have the ability to add their own device drivers.

Currently Chimera 3.0 provides drivers for the following I/O devices: Ironics IV3220/30 serial ports, Ironics IV3204 local VMX bus, XYCOM Model 240 Parallel I/O board, XYCOM Model 500/590 Analog Input Module, XYCOM Model 505/595 Analog Output Module, Ironics IV1645 A/D Converter, Ironics IV1640 D/A converter, Ironics IV1623 Parallel I/O, Imaging Techonology Series 150 Vision System. Device drivers for several custom-built I/O boards are also provided as examples.

Sensors and actuators are generally connected to these I/O devices. The second-level device driver support can be used to provide a hardware independent interface to the sensors and actuators in a system. A generic interface to the sensors and actuators is being designed as part of the reconfigurable software architecture methodology on which Chimera 3.0 is based.

The transparent access to the host workstation's devices allow the real-time environment to make use of the host workstations's file system, monitor (or any window within the monitor) for *stdout*, the keyboard for *stdin*, or the network. (Note: currently access to the network must be done explicitly, it is not transparent in the real-time environment. Transparent access should be available in Chimera 3.1).

Chimera 3.0 Documentation

- Complete documentation in seven manuals:
    - Beginner's Tutorial
    - Installation Manual
    - User's Guide
    - Programmer's Manual
    - System's Manual
    - Reconfigurable Systems User's Manual
    - Reconfigurable Systems Programmer's Manual

Carnegie
Mellon

Chimera 3.0, The Real-Time Operating System for Reconfigurable Sensor-Based Control Systems, comes with a full set of documentation:

The **Beginner's Tutorial** provides three simple examples to allow users to become familiar with the Chimera environment, and to whet their appetite for the capabilities of Chimera.

The **Installation Manual** should be used by the system administrator when installing the system. This manual also describes the hardware and software support for the current version of Chimera, and possible configurations for multiprocessor systems.

The **User's Manual** describes the *chim* user interface. It provides details on downloading and executing programs within the real-time environment, and using the interface as a remote console to one or more processors. It also provides suggestions on building custom programs on the Sun which communicate with the real-time environment.

The **Programmer's Manual** provides a reference manual for programming applications running under Chimera. Included are the task control primitives, interprocessor communication routines, standard C library calls, including *stdio*, emulated system calls such as *read()* and *write()*, and many other useful C library calls available in Chimera.

The **System's Manual** is intended for advanced users. It provides details for adding new devices and processors to the system, and for modifying parts of the kernel, such as the scheduler, interprocessor communication, or real-time synchronization primitives. The hardware dependent code is also described in this manual, along with hints on porting code to new hardware platforms. This manual also provides a section of issues to consider when setting up a multiprocessor system.

The **Reconfigurable Systems User's Manual** describes our software architecture for reconfigurable systems, and the user's interface for running a system using only existing software modules. The modules can be reconfigured based on the application, and configuration files are used to change default parameters for each of the modules. The interfaces to the subsystem task control mechanism are also described.

The **Reconfigurable Systems Programmer's Manual** describes the modular specifications for a reconfigurable software module. Programmer's who are writing their own modules should refer to this manual.

Systems Using Chimera

- **At Carnegie Mellon University:**
  - **Troikabot System for Rapid 3D Assembly**
  - **Direct Drive Arm II**
  - **Reconfigurable Modular Manipulator System**
  - **Self Mobile Space Manipulator**
  - **Amada Steel Sheet Bending System**
- **Elsewhere**
  - **Jet Propulsion Laboratory**
  - **Air Force Institute of Technology**
  - **University of Alberta**
  - **Wright State University**
  - **PosTech University**
  - **etc.**

Carnegie Mellon

Several installations of Chimera currently exist. At Carnegie Mellon University (CMU), the following projects are using Chimera:

The **Troikabot System for Rapid 3D Assembly** is a system of 3 PUMA 560 robots. It combines design and robotics with the goal of developing a system that is capable of generating physical assemblies directly from 3D solid modeler descriptions.

The **CMU Direct Drive Arm II** is used as a testbed for developing new control algorithms, and for multi-sensor integration research. The hardware setup of this system is shown in the next slide.

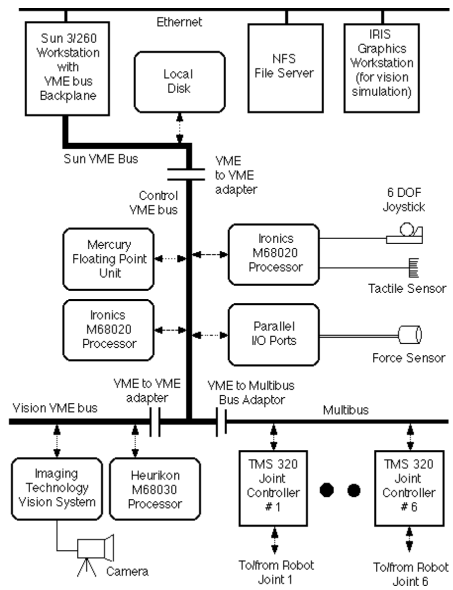
The **CMU Reconfigurable Modular Manipulator System (RMMS)** is an advanced manipulator design that is reconfigurable and can be adapted to specific applications.

The **Self-Mobile Space Manipulator** is a robot that has locomotion and manipulation capabilities for inspection and repair of space station freedom.

The **Amada Steel Sheet Bending System** uses a robotic manipulator to feed a previously manually-operated steel sheet bending system.

Outside CMU, several university, industry, and government labs are also using Chimera. This includes (but is not limited to) projects at NASA's Jet Propulsion Laboratory, California; the Air Force Institute of Technology, Ohio; University of Alberta, Canada; Wright State University, Ohio; and PosTech University, South Korea.

CMU Direct Drive Arm II



The CMU Direct Drive Arm II (DDArmII) hardware architecture is an example of the type of system that can be set up using Chimera.

The host workstation is a Sun 3/260 with VME backplane, and BIT3 VMEbus-to-VMEbus adaptor. The host is networked with a variety of other systems, including and NFS file server and an IRIS graphics workstation.

The system has two general purpose M68030 RTPUs, on which the Chimera real-time kernel is operating. A Mercury 3200 floating point accelerator is incorporated into the system as a special purpose processor. The DDArmII is interfaced to software through a series of six TMS320 digital signal processors, which are on custom-built boards in a Multibus backplane. A VMEbus-to-Multibus adaptor is used to link the system with the VMEbus real-time system.

The system also contains position sensors for the robot joints, a force sensor, a tactile sensor, and a 6-DOF joystick. The position sensors are connected to the TMS320s which act as intelligent I/O devices. The force sensor is connected to a parallel I/O port, while the tactile sensor and joystick are connected through serial ports.

A stand-alone Imaging Technology vision system has been incorporated into the system with communication through a BIT3 VMEbus-to-VMEbus adaptor. The Heurikon M68030 processor is executing the OS-9 operating system, which controls the special purpose processors in the vision system (Chimera is not used on the Heurikon because all the libraries for the Imaging Technology are provided as OS-9 object code, and source code is not available). This connection demonstrates Chimera’s ability to interact and operate in conjunction with other complete “black-box” systems. The vision system has been used for visual servoing at a rate of up to 15 Hz.

The setup allows the DDArmII system to run with a 1200 Hz servo rate when using PD or PID control, an 800 Hz rate when using cartesian control without dynamics compensation, and a 400 Hz rate when using cartesian control with dynamics compensation.

Summary

- Chimera 3.0
  - The real-time operating system for reconfigurable sensor-base control systems
- Further Research:
  - software architectures
  - dynamic reconfiguration
  - real-time task profiling
  - automatic integration
  - generic graphical user interface

Carnegie
Mellon

Chimera 3.0 has been developed to address the needs of designing and implementing reconfigurable and reusable sensor-based control systems. It provides many features allowing the development of hardware independent code which is required for such a system. The project is ongoing, and current efforts include the following:

Software Architectures: we are continuing our work in software modelling of control modules, sensors and actuators, special purpose processors, and I/O devices. Using these models, we can obtain more precise static configuration analysis and verification.

Dynamic Reconfiguration: we are looking into the question of how we can dynamically change from one configuration to another while maintaining the integrity of the system. Some of the issues involved are real-time issues, while others deal with control system stability. We are looking for developing sets of pre-conditions and post-conditions which can be used to determine when a system is in a state at which time dynamic reconfiguration can be performed.

Real-time task profiling: in order to implement most real-time scheduling algorithms, a very good execution model of the real-time task is required. However, such execution models are often not available, and may change depending on the system configuration or target hardware being used. We are investigating methods for automatically profiling real-time tasks to determine accurate execution models, and to use that information for automatic integration and real-time scheduling.

Automatic Integration: several issues deal with automatic integration. The most obvious is the global scheduling which involves allocation of tasks to processors. We are also investigating how to handle situations where there is insufficient resources to do the requested application. Modifying task frequencies or sharing resources can sometimes still allow a system to run, albeit perhaps with a lower level of performance. We are also analyzing the interaction of modules which operate at different frequencies, to try and automatically determine the best choice of frequencies for tasks which may run at any range of frequencies, and for sensors in which not every sample of data is required.

Generic Graphical User Interface: we are developing a user interface and visual programming language that can be used to quickly program and configure reconfigurable systems, which is called *Onika*. Matthew Gertz is working on this project as part of his Ph.D. research at CMU. Onika will be integrated into Chimera 3.1.