THE AMCAD REAL-TIME MULTIPROCESSOR OPERATING SYSTEM

Michael S. Rottman, 1Lt, USAF
Daniel B. Thompson

Wright Research and Development Center (WRDC/FIGLB)
Wright-Patterson Air Force Base, OH

Abstract

The Flight Control Division of the Air Force Flight Dynamics Laboratory has been researching the application of fault tolerant multiprocessor architectures and software to flight control and vehicle management systems. The Advanced Multiprocessor Control Architecture Development (AMCAD) in-house project has developed a Real-Time Multiprocessor Operating System (RTMOS) targeted for the AMCAD laboratory testbed. The RTMOS is similar in structure to commercial real-time operating systems, but has been expanded to support coarse grained multiprocessing, the reconfiguration of the workload in case of processor failure, and AMCAD's goal of hiding architecture specifics from the applications programmer. This paper will discuss the structure and capabilities of the RTMOS, as well as the intertask communications protocols and structures and the application programmer interface.

1 Introduction

The Flight Control Division of the Air Force Flight Dynamics Laboratory has been researching fault tolerant multiprocessor architectures since the early 1980s. This work began with the Continuously Reconfiguring Multi-Microprocessor Flight Control System (CRMMFCS) in-house project, which examined the potential benefits of pooled sparing, reconfiguration, and parallel processing as applied to flight/vehicle control. The Advanced Multiprocessor Control Architecture Development (AMCAD) program has continued the CRMMFCS research, further maturing and developing key technologies. Again performed in-house by Flight Control Division engineers, the AMCAD program is producing a testbed system to demonstrate the architectural concepts and provide a basis for further research. A key element of the AMCAD research has been the development of the Real-Time Multiprocessor Operating System (RTMOS). The target of the RTMOS is the AMCAD laboratory testbed. This paper begins with a brief overview of the host AMCAD architecure. The structure and features of the RTMOS as implemented for the AMCAD testbed system will be discussed. Finally, the current status and future plans for the RTMOS will be described.

2 AMCAD Overview

This section provides a brief overview of the AMCAD concepts. It is not meant to be a comprehensive examination of these concepts, much of which is outside the scope of this paper. More detailed descriptions of the overall system are presented elsewhere [1-5].

The primary objective of the AMCAD project is to provide the computational power and reliability needed for flight control and vehicle management systems while reducing the amount of required maintenance. To achieve these goals, AMCAD seeks to break away from conventional quad and triplex redundant architectures and use instead a fault tolerant multiprocessor configuration featuring nondedicated (software-mapped) redundancy, pooled sparing, and parallel processing. The architecture was developed to be expandable and adaptable to emerging military processor, bus, and high order programming language standards.

The AMCAD architecture is based upon modules consisting of multiple general purpose microprocessors, memory, and a bus interface unit (BIU). Each processor has its own backplane and BIU access port. Processors in a module are computationally independent, but can "shutdown" (isolate from the bus) faulty processors within that module. The modules are connected by a linear token passing bus network. This bus network is multi-level (hierarchical) for expandability and fault containment partitioning, with multiple busses per level (Figure 1).

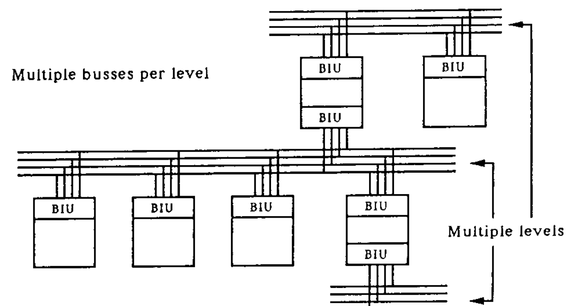

Figure 1. Multiple levels of bussing with multiple busses per level

U.S. Government work not protected by U.S. copyright.

None of the processors is physically dedicated to redundant computation; they serve instead as a pool of computational resources which may be used for parallel processing, multiple independent computations, or redundant computation if needed. Likewise, the multiple busses per level are not physically redundant. Data communications can be distributed across the multiple busses, allowing graceful degradation in case of a bus failure.

A critical concept in the AMCAD development is that of logical or nondedicated redundancy. Redundant "channels" of computation can be software mapped across the multiple processors to provide reliability in the presence of hardware faults. Faults are masked by voting the results of the redundant computations, allowing correct operation despite failures. If a processor performing part of a redundant computation fails, the total workload is redistributed to reestablish the redundant computation. A description of logical redundancy in the AMCAD architecture is given in [5].

Tasks communicate through the Virtual Common Memory (VCM), a concept which originated in the CRMMFCS work. An identical copy of "shared memory" is placed local to each processor. Broadcast bussing is used to transfer data items to each processing module. This allows communications to be done task-to-task through the VCM rather than processor-to-processor. All read operations are local, providing fast access to data and eliminating the shared memory bottleneck. Data in the VCM is protected from contamination by a segmentation scheme in which the proper key value is required for write-access to a segment.

One goal in the development of the AMCAD architecture was to make the physical configuration transparent to the applications programmer. Each processor has an identical copy of the operating system and applications software. All data is locally available to all processors (through the VCM). Tasks communicate by sending messages to and reading messages from the VCM. As a result, applications programs "see" the architecture as a set of homogeneous processing resources such that programs can be run on any number of processors; even one, if time constraints can be met. The programming model will be discussed at greater length in Section 3.3.

A major element in the AMCAD project has been the development of the AMCAD Real-Time Multiprocessor Operating System (RTMOS). The RTMOS combines aspects of distributed and real-time operating systems to provide support for real-time performance requirements, parallel tasking, task and data redundancy, and reconfiguration.

3 The AMCAD Testbed RTMOS Implementation

The development of the AMCAD RTMOS has focused on the application of real-time operating system techniques to multiprocessor systems. Single processor multi-tasking operating systems have been used extensively for real-time control and can provide efficient and timely scheduling of tasks. This model can be expanded to multiple processors by distributing the total workload of the system across the processors, giving each processor its own set of tasks to execute [3]. Each processor has an identical multi-tasking kernel to manage the concurrent execution of that processor's workload, as in the single processor model. A set of upper level system functions built on top of the kernel handles the interaction of the multiple processors. The set of local operating systems (kernel and upper level functions) collectively comprise the AMCAD RTMOS (Figure 2). This section will consider both aspects of the local operating systems: the single processor kernel model and the upper level multiprocessor functions.

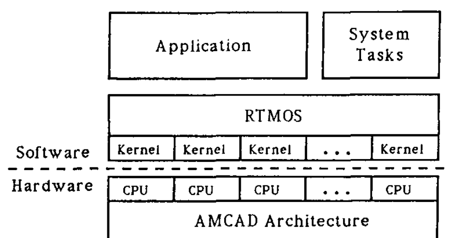

Figure 2. Hardware/Software Overview

3.1 The Multi-tasking Kernel

Each processor is allocated a portion of the total set of application tasks. The function of a processor's kernel is to manage the concurrent execution of tasks on that processor. This section first defines tasks and how they are implemented in AMCAD, followed by a discussion of how these tasks are scheduled to the CPU. Allocation of tasks to processors will be discussed in section 3.2.1.

3.1.1 Tasks in AMCAD

A **task** is a sequential unit of software which can be executed in parallel with or sequentially in coordination with other tasks, based upon precedence constraints between tasks. The application workload is partitioned into tasks in much the same manner as conventional software is organized into procedures or subroutines [3].

Each task is represented in the operating system by a data structure called a Task Control Block (TCB). The complete state of a task is reflected in its TCB, such as a pointer to the task, program counter, status register, stack pointer, register values, as well as the task priority, queue pointers, and alarm values. In addition, the stack area for each task is maintained in the TCB.

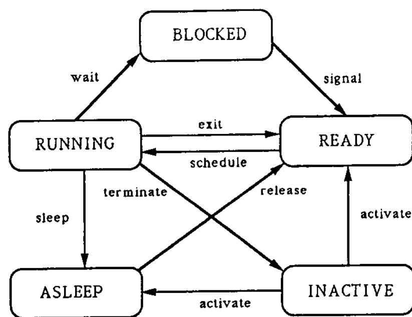

Figure 3. Process States

A task may be in one of five states at any given time, as shown in Figure 3:

- running - The task "owns" the CPU and will run until it blocks, completes, or is preempted.
- ready - The tasks is ready to run and is waiting for the CPU.
- blocked - The task is waiting for a system resource.
- suspended - The task is waiting until the next time it is to execute.
- inactive - The TCB has been released and the program is no longer executing.

To activate a task, a TCB is loaded with the appropriate initial information about the task, including its address, priority, execution period, and a pointer to its global data variable(s). Multiple instantiations of a task may be activated by creating multiple TCBs pointing to the same code, each with a pointer to different global data variables.

3.1.2 Task Scheduling

Task scheduling involves determining which task should be run at any given time. Tasks must be switching in and out of the processor in such a way that each task has a complete computing environment when it is running, in effect creating the impression that each task has its own virtual processor. This environment must be saved while the task is not running so that execution may resume at a later time. An excellent survey of scheduling algorithms can be found in [10].

Scheduling is performed in the AMCAD system by means of a prioritized ready queue and a timer queue. The ready queue is implemented as multiple first-come-first-served queues, one for each priority level. High priority tasks will always get the CPU before tasks with lower priority. Tasks waiting a ready queue are in the ready state.

The majority of tasks in real-time control applications are periodic in nature. Periodic tasks are invoked at fixed time intervals and are the normal computation of the controlled system. A nonperiodic task may be invoked at any time, typically for abnormal or critical situations. The problem with using a ready queue with periodic tasks is that these tasks spend valuable processor time waiting to start the next computation. To minimize this problem, a special timer queue is used. The kernel places periodic tasks in the timer queue when their computation is complete, suspending them for a specified amount of time (the period of the particular control computation). The queue is order by release time rather than by priority. In AMCAD, a local clock counter is incremented every one millisecond in response to an interrupt from a hardware timer. The interrupt handler updates the clock variable and "wakes up" all tasks in the timer queue whose start time has been reached.

3.2 Multiple Processor Issues

As stated above, the kernels are responsible for managing the operation of the local processors and the execution of the task sets allocated to those processors. The upper level system functions handle the interaction of the multiple processors. Reconfiguration, task allocation, intertask communications, and clock synchronization will be discussed in the following sections.

3.2.1 Reconfiguration

In AMCAD, reconfiguration refers to the redistribution of the application workload to account for changes in the state of the system. Because of the high reliability required for flight control and the high threat environment in which these systems operate, the capability to redistribute the task load when a processor fails is essential. The overall objectives of reconfiguration are to reestablish full redundancy levels for fault masking and to provide graceful degradation of application functions as resources are lost. One advantage in the use of pooled sparing in AMCAD is that all processors initially participate in the execution of the application workload, each at less than peak utilization. As resources are lost, the workload is redistributed among the healthy processors without loss of functionality. When the number of resources becomes insufficient to manage the entire workload and still meet timing requirements, the lowest priority jobs are lost first, providing graceful degradation.

Reconfiguration in AMCAD can be either periodic or on-demand. In the case of periodic reconfiguration, each processor examines the state of the system at fixed intervals. The system state is maintained in the VCM, with the status flag for each processor stored in a different segment so that no processor can modify another's flag. If the new system state differs from the previous state, reconfiguration is triggered. On-demand reconfiguration occurs when the system determines a processor is faulty and

triggers reconfiguration on all processors. Once reconfiguration has been triggered, each processor reinitializes its ready and timer queues and places all currently active TCBs back in the free TCB pool. The remaining healthy processors select a new task set using the task allocation process.

3.2.2 Allocating Tasks to Processors

The kernel discussed above dynamically schedules a processor's task set independent of the other processors. No mention is made of how the total workload is parceled out to the processors. This process is called task allocation or assignment and is performed during reconfiguration.

AMCAD uses a static allocation algorithm. Several factors affected the decision to use static rather than dynamic task assignment. Dynamic algorithms are more flexible and tend to balance the workload more evenly across the processing elements. However, static task allocation is used because it is faster, more deterministic, and supports the single processor model.

Allocation of tasks to processors is static in that the distribution of tasks only changes when the number of healthy processors changes. When reconfiguration is triggered, each processor selects its task set based upon the number of healthy processors and the processor identification number. Since the number of processors anticipated for each of the multiple levels of the AMCAD architecture is relatively small, a table driven algorithm is used. The task sets are predefined for each possible number of healthy processors.

Very few constraints are placed on task allocation. Since AMCAD uses a shared memory model for interprocessor communications, there is no need to place tasks which interact frequently on the same processor. Constraints added by the application of logical redundancy techniques are discussed in [5]. Clearly, however, performance will be affected by how effectively the parallelism of the system is exploited. Experimentation will be needed to tune the processor working sets to achieve load balancing.

3.2.3 Intertask Communications and Synchronization

A primary driver in development of the AMCAD RTMOS is to allow the application programmer to design applications without needing to know how many processors the system is using or which processors are executing which tasks. To meet this objective, an intertask communications protocol is needed which does not rely upon knowing the processor on which tasks are currently running. A pure message passing approach does not meet this criteria, but an indirect message passing algorithm using mailboxes [7] maps well to the AMCAD architecture.

In AMCAD this involves setting up a series of data structures in the VCM through which tasks can pass data or parameters. Every data item passed between tasks is assigned one of these data structures. These data "mailboxes" include two flags used to synchronize task interaction to protect data dependencies: the **producer** and the **consumer** flags. Routines are provided which interact with these flags to allow tasks to supply or acquire global data. These routines insure that data is not read from the global variable unless it is current and that data cannot be overwritten until it has been read. When a task produces a piece of data, it performs a **p.poll** operation; to consume a piece of data, a task performs a **c.poll**. The algorithms for these operations are shown below. The producer and consumer flags are implemented as counter values which are incremented each time a data item is successfully produced or consumed. This allows **p.poll** and **c.poll** to ensure they are referring to the same data item.

```
producer
if var(P) ≠ var(C) then
  report error to system
else
  var(P) = var(P) + 1
endif
put var(data)

consumer
save registers
while (var(P) = var(C)) and
     (not timeout) do wait
if var(P) = var(C) then
  report error to system
else
  var(C) = var (C) + 1
endif
get var(data)
restore registers
```

This method of intertask communication is a variation of a unilateral rendezvous. Only consumer tasks wait on data, with a timeout set so that wait time is bounded. If a producer task attempts to produce new data before previous data has been consumed, the error is reported to the operating system. The old data will be overwritten but the producer/consumer flags will not be updated. If a consumer task times out because the awaited data was never produced, the consumer will use the old data. This error is also reported to the operating system.

Ideally, some means of mutual exclusion would be provided to protect these data structures from access by several tasks at the same time. AMCAD's VCM and broadcast bussing approach, however, make mutual exclusion extremely difficult to implement. This problem is avoided by limiting the data structures to one producer task and one consumer task each. In this case, the protocol is sufficient for mutual exclusion.

3.2.4 Clock Synchronization

The efficiency of the dataflow-like operation supported by the RIMOS depends heavily on the synchronization of the local clocks. Since each processor maintains its own local clock variable independent of the other processors, there is potential for these local clocks to drift over time with respect to one another. If the local clocks drift far enough apart, waiting tasks will be released from the timer queues at the wrong times. Data will not be produced or consumed when expected. As a result, deadlines will be missed if the clocks are not synchronized periodically. It is not essential that the local clocks be identical, merely that they all be within a given "window" or range of one another.

RIMOS handles clock synchronization with two periodic system tasks, Synch_Report and Synch_Adjust, which are assigned to all processors. The Synch_Report task periodically broadcasts the local clock value. Though the period of the Synch_Report task is the same on all processors, the start times are skewed so that a different processor broadcasts each interval. For example, in a ten processor system, each processor might broadcast only once per second. CPU #1 would broadcast its clock at 100 msec, 1100 msec, 2100 msec, and so on. CPU #2 would broadcast at 200 msec, 1200 msec, 2200 msec.

These broadcast clock values are routed to special interrupt words at each processor's VCM. An interrupt handler stores the received clock value, along with the current local clock, in a table in local memory as his clock/my clock data pairs. Task Synch_Adjust at each processor periodically evaluates the local table and calculates how much to adjust the local clock based upon the average difference for the his-clock/my-clock pairs. Experimentation will be required to determine what periods are needed for Synch_Report and Synch_Adjust for the target AMCAD hardware.

3.3 AMCAD Application Interface

Several concepts have been emphasized in the development of the AMCAD RIMOS and described in this paper which affect the development of applications software for the AMCAD architecture testbed. To the software engineer, the architecture appears as a pool of homogeneous computing elements. Each processor has an identical copy of the operating system, application software, and data. Any software job can be assigned to any processor. The multitasking kernel allows concurrent execution of multiple tasks on each processor, allowing the total workload to be distributed across any number of processors. All intertask communications takes place through the VCM so that no task needs to know the location of any other.

These features form a basic programming model which allows the applications programmer to develop software for use with the RIMOS without needing to know or worry about any specifics of the underlying architecture. The programmer is freed from details of how tasks will be allocated or communicate. Routines are provided by RIMOS for intertask communication and task management. This "application interface" allows for the effective development of multiprocessor software, independent of the number and configuration of processors. Specifics of the underlying architecture are transparent to the application, thus easing software development.

4 Status and Conclusions

Development of the AMCAD RIMOS has been carried out in parallel with the construction of the laboratory testbed hardware. Most of the functions described above have been coded in assembly language and tested on a separate dual-processor board. Programming of the communications routines has begun, but these will require the testbed hardware for testing. Some code must also be revised to account for the differences between the dual-processor board and the AMCAD configuration. System software to manage failure detection and self-test must still be developed.

Much work remains to be done. The operating system software will be completed soon and testing will continue. Research is planned in the area of multiprocessor software development utilizing the programming model developed as part of the RIMOS. Additional research is underway to enhance the AMCAD Applications Interface to allow the transition of the multiprocessor software development methodology to other architectures. Additional details on the AMCAD architecture, RIMOS, and further research will be documented in a later, more comprehensive report.

Bibliography

[1] S. Larimer et al., "A Continuously Reconfiguring Multi-Microprocessor Flight Control System", AFWAL-TR-81-3070, AFWAL/FIGLB, May 1981.

[2] D. Thompson et al., "AF Multiprocessor Flight Control Architecture Developments: CRMMFCS and Beyond", NAECON 1986 Proceedings, May 1986

[3] D. Thompson, "A Multiprocessor Avionics System for an Unmanned Research Vehicle", AFWAL-TR-88-3003, AFWAL/FIGLB, March 1988

[4] D. Thompson, "Linear Token Passing Based Bus Interface for a Fault Tolerant Multiprocessor Testbed", Proceedings of the ACC 1989, August 1989

[5] D. Thompson et al., "The Use of Nondedicated Redundancy in a Fault Tolerant Multiprocessor", NAECON 1989 Proceedings, May 1989

[6] J. Stankovic et al.,"Introduction", Hard Real-Time Systems Tutorial, Computer Society Press of the IEEE, 1988

[7] J. Peterson et al., "Operating Systems Concepts", Addison-Wesley Publishing Company, 1985

[8] M. Maekawa et al., "Operating Systems: Advanced Concepts", Benjamin/Cummings Publishing Company, 1987

[9] A. Tanenbaum, "Operating Systems: Design and Implementation", Prentice-Hall, 1987

[10] S. Cheng et al., "Scheduling Algorithms for Hard Real-Time Systems - A Brief Survey", Hard Real-Time Systems Tutorial, Computer Press of the IEEE, 1988