IMPLEMENTATION OF NON-DEDICATED REDUNDANCY
IN A FAULT TOLERANT MULTIPROCESSOR TESTBED

Capt Michael S. Rottman
Daniel B. Thompson
Wright Laboratory (WLF/FG) Wright-Patterson Air Force Base, Ohio

Abstract
The Flight Control Division of the Air Force Wright Laboratory (WL/FiGL) has been researching fault tolerant multiprocessor architectures for flight control and vehicle management systems. A critical aspect of this research is to break from the dedicated (static) triplex or quadruplex redundancy primarily used in flight control, utilizing instead a pool of processing resources with non-dedicated (software-mapped) redundancy. This paper describes the implementation of non-dedicated redundancy in the Advanced Multiprocessor Control Architecture Development (AMCAD) conceptual system and testbed.

Introduction
As military aircraft mission requirements have expanded (stealthy airframes, greater aerodynamic capabilities, increased pilot aiding, etc.), demands on embedded computing have become much greater. In flight control, trends are toward higher degrees of integration and increased functionality, while maintaining high levels of reliability.

The Flight Control Division of the Air Force Wright Laboratory (WL/FiGL) has been researching fault tolerant multiprocessor architectures for flight control and vehicle management systems 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 to flight/vehicle control. CRMMFCS marked the beginning of research into the concept of non-dedicated redundancy, a break from the static (dedicated) architectures commonly used in flight control. The Advanced Multiprocessor Control Architecture Development (AMCAD) program has continued the CRMMFCS research, focusing on the application of non-dedicated redundancy to flight control and vehicle management systems.

This paper will examine aspects of the AMCAD architecture and operating system concepts supporting non-dedicated redundancy. It will describe the concept and implementation of non-dedicated redundancy in the AMCAD testbed, and the lower-level operating system mechanisms for providing redundant communications and voting on redundant data. Finally, it addresses future research under consideration for this testbed.

AMCAD Overview
This section provides a brief overview of the AMCAD concepts. It is not meant to be a comprehensive examination of these concepts, many of which are outside the scope of this paper. More detailed discussions of the system architecture and operating system are presented elsewhere [1-7].

Objectives
The AMCAD in-house project is producing a testbed system to demonstrate and evaluate hardware and software architecture concepts and provide a basis for future research. The primary objective of AMCAD is to derive and demonstrate architectural concepts providing greater computational power and functional integration while meeting reliability requirements. A further goal is to enable reliable operation, even after multiple failures, providing for longer operation without required maintenance. To achieve these goals, AMCAD breaks from conventional quadruplex and triplex redundant architectures and instead uses a fault tolerant multiprocessor configuration featuring non-dedicated (software-mapped) redundancy, pooled sparing, and parallel processing. The architecture was developed to be expandable and adaptable to emerging military processor, data bus, and high order language standards.

Architecture Concepts
The AMCAD hardware architecture is based on processing modules consisting of multiple general purpose microprocessors, local memory, and a bus interface unit (BIU). Each processor has its own "backplane and bus access port. Figure 1 shows a module from the AMCAD testbed. Processors in a module are computational 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), with multiple busses per level (Figure 2).

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 as 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.

U.S. Government work not protected by U.S. copyright.
The multi-level bus configuration has two benefits. First, the multiple levels partition system data into "local" and "global" communications. The functionality of processors on a level will have common data needs with frequent exchanges. Inter-level communications is less frequent and of a cross-functional or between subsystems nature. Second, the bus structure provides a capability for fault containment partitioning of the system into "reconfiguration zones." Reconfiguration and redundancy can be handled within levels for fault detection and correction.

Tasks communicate through the Virtual Common Memory (VCM). An identical copy of "shared memory" is replicated at 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. Redundant data variables can be spread over multiple segments to prevent critical data contamination from a single source.

RTMOS

The software architecture is based on the Real-Time Multiprocessor Operating System (RTMOS), which combines aspects of conventional real-time and distributed operating systems to support real-time performance requirements, parallel tasking, task and data redundancy, and reconfiguration. The RTMOS is comprised of an identical multi-tasking kernel on each processor and a set of system management functions built on top of the kernels. Each processor is allocated a portion of the total set of application tasks, and executes its copy of the kernel to manage those tasks. The system management functions perform such operations as clock synchronization, fault detection and isolation, and reconfiguration. The RTMOS architecture-independent programming model allows the workload to be distributed onto any number of processors; even one, if timing constraints can be met.

Non-Dedicated Redundancy

This section describes the concept of non-dedicated redundancy and its implementation in the AMCAD fault tolerant multiprocessor testbed. It presents a brief overview of the concept of non-dedicated redundancy, extracted from [7], followed by a detailed discussion of the implementation for the testbed architecture. This discussion covers the RTMOS requirements generated by non-dedicated redundancy, the data structures used by the RTMOS to manage the task and data redundancy, the mapping of tasks to processors, communications primitives, the voting algorithm used, and task reconfiguration. The discussion concludes by examining some of the benefits of non-dedicated redundancy as seen in the AMCAD implementation.

Concepts

The primary component of fault tolerance in the AMCAD architecture and testbed implementation is non-dedicated redundancy, which breaks from the dedicated (static) triplex or quadruplex redundancy primarily found in flight control. With non-dedicated redundancy, redundant "channels" of computation can be software mapped across the multiple processors to provide computational reliability in the presence of hardware faults. Faults are immediately masked by voting the results of the redundant computations, allowing continued correct operation despite failures. If a processor performing part of a redundant computation fails, the total workload can be redistributed to reestablish the full redundancy of the computation.

In non-dedicated redundancy, the redundant task sets interact at the task level rather than the channel level, with voting occurring at the task input stage. Figure 3 demonstrates this concept for triplex tasks. Each redundant producer task creates a separate output, each of which is stored in a different segment of the VCM. Each consumer task reads all three of the data items, votes them, and selects a value to use. This process occurs independently for each consumer task. The consumer triplex task may then produce data for another triplex task. Figure 3 also illustrates that the consumer and producer triplex tasks may be performed on different sets of processors.

One important benefit of the VCM concept for non-dedicated redundancy is that a copy of the global data is local to each processor. This allows the time-critical element of redundancy, voting, to be performed on local data without the overhead of accessing the bus. Writes to the VCM, however, are global.
operations requiring bus access. The segmentation and limited task write-access privileges protect the redundant data from corruption.

**Implementation**

**Requirements** The implementation of non-dedicated redundancy in the AMCAD testbed derives several additional requirements which the RTMOS must satisfy. Since individual tasks or sets of tasks can be replicated to the desired level of redundancy, the RTMOS must provide mechanisms for the flexible, efficient management of redundant tasks, communications, and voting.

Redundant tasks can be either identical or dissimilar. In many critical applications, redundant tasks are dissimilar to provide protection from generic software faults. Each redundant task performs the same function and has the same input and output characteristics, but may be developed by different people, with different algorithms, or even in different languages. In the simpler case, all redundant copies of a task can be replicated from the same code segment, creating a need for either copies of the task code or task code that is "reentrant," with the associated ability to manage multiple tasks sharing the same code.

Since multiple tasks may use the same code segment but will use different local and global data variables, a flexible means is needed for notifying each task which variables to use. Though each task within a redundant task has consistent numbers of input and output variables, different redundant tasks may have differing I/O requirements. The variable information passed to tasks must be adaptable to allow for the different requirements of different redundant task sets.

The RTMOS is responsible for determining and controlling task access to the segmented VCM to prevent tasks from producing or consuming global variables for which they are unauthorized. This management of VCM access must be transparent to the application tasks, and can be accomplished by operating system control of segment keys and protection of the operating system data structures from the application.

Finally, the RTMOS must provide a flexible and efficient mechanism for voting the redundant computations. This mechanism should be transparent to the application programmer.

**RTMOS Data Structures** The RTMOS utilizes two key data structures to provide the necessary global variable and redundancy management capability. The Variable Information Table (VIT) defines the configuration and level of redundancy of the global data variables in the VCM. The VIT is a static list of all global data variables. For each variable, the VIT defines the VCM address of the variable, the number of the VCM segment containing the variable, the segment "key" value, and the task identification numbers (IDs) for the variable's producer task and consumer task. This table is defined by the application designer, with an identical copy stored in ROM on each processor to prevent corruption of the variable configuration data. Since each variable has only one producer and one consumer task, the VIT structure is a powerful tool for controlling task write access to the VCM.

If the variable is redundant, the VIT also cross references the addresses of the other variables in the redundant data set. All data redundancies are therefore defined in the VIT; the mailbox variables themselves are inherently simplex. They contain only the producer and consumer flags, the variable data size, and the data itself.

The second data structure is the Task Variable Table (TVT), which is used to communicate to a task what global variables it produces and consumes and what local variables it uses. Each task has its own static TVT defined during application design. When a task is switched into the processor for execution, it is passed a pointer to its TVT. In this manner, each task (even those based on the same code segment) is provided the mailbox numbers of the variables it produces and consumes, as well as pointers to its own private workspace variables.

The designer must ensure that a task's TVT is consistent with what the task code expects. If the task produces two global data items, the first two items in the task's TVT must be the mailbox numbers of those two items, in the expected sequence. Consistency checks on the task code and TVT information could be easily automated, particularly if the code were written in a high order language such as Ada, which separately specifies the input and output parameters of each task.

**Mapping** Since AMCAD tasks communicate through shared mailbox variables in the VCM, application tasks are effectively independent of one another and the specific hardware architecture. A task consumes certain global data items, it performs some function or collection of functions, and it produces others global data items. Under this task model, a given task does not need to know which other tasks provide it with data or consume its data. Likewise, since communications occurs through "shared memory" rather than directly task-to-task, the mapping of tasks to processors is completely transparent to the tasks themselves. This approach allows the application to be designed without concern for the number of processors in the system or which tasks will run on which processors. Tasks can be mapped onto any number of processors; even one, if application time constraints can still be met.
Because of the AMCAD shared memory communications model, there is no need to place tasks which interact frequently on the same processor. Tasks can be distributed onto the multiple processors according to data precedence relations in order to exploit the parallelism of the application. That is, if three tasks can execute in parallel they could be mapped to different processors to reduce overall execution time. Since task assignment is statically defined at design time, experimentation is needed to "tune" the processor mappings for a given application to achieve load balancing.

Task redundancy creates one critical constraint on the mapping of tasks to processors: no two redundant tasks should be performed on the same processor. This constraint is only common sense. If two copies of a triplex task are mapped to a faulty processor, the fault may not be masked by voting and may propagate. For triplex redundant tasking to be effective, therefore, at least three processors are needed. In the AMCAD architecture, a pool of processors is utilized such that the maximum level of redundancy (for the most critical tasks) is always provided for.

Communications A flexible, transparent communications mechanism is required to efficiently implement non-dedicated redundancy. Tasks must be able to produce data to and consume data from the appropriate VCM mailboxes without contending with low-level details of data redundancy management.

AMCAD intertask communications is a form of unilateral rendezvous. Only consumer tasks wait on data, with a timeout set so that wait time is bounded. Producer tasks always produce new data to their mailbox variables, regardless of whether the previous frame's data has been consumed. If a producer task overwrites unconsumed ("old") data, the old data is overwritten and the error is reported to the RTMOS, but the variable's producer/consumer flags are not updated. If a consumer times out because new data was never produced, the consumer uses the "old" data and reports the error to the RTMOS. This produces a producer/consumer protocol described in more detail in [2].

Data production is a strictly simplex activity. As noted above, the global mailboxes themselves are singular data items. Each task of a redundant set produces its own data item, which is transmitted to its appropriate global variable. The SEND operation used by the tasks to transmit data to the VCM does not have the overhead of dealing with redundant data.

The SEND algorithm accepts a pointer to a local variable to send and the destination VCM mailbox number. The mailbox number is used as an index into the VIT, where information about that mailbox is stored. If the sending task is listed as the mailbox's producer task in the VIT, a buffer is allocated for the data. The VCM segment number and key value are obtained from the VIT and loaded in the buffer. Each longword of data is then stored in the buffer with an offset into the VCM segment where the data should be located. When the buffer is full, it is added to the Transmit_Message queue for transmission by the bus interface unit.

Regardless of whether a task or data item is redundant, a task performing a receive (RECV) operation only obtains one data item. Any collection of redundant data items and voting on the item is done by the RTMOS transparent to the calling task. Only the resulting data item is passed to the task. Ideally, the task would simply issue a RECV call. If the RTMOS determined that the data item was redundant, based on the mailbox's VIT entry, a redundant receive (with voting) would be performed. A simplex item would be consumed directly. While this approach provides the most transparency to the application tasks, it drastically limits flexibility. For the AMCAD testbed, the application can specify the level of redundancy to the RECV, allowing control over the number and location of voting as needed. This provides a flexible means of examining the overhead of AMCAD data redundancy and amount of voting necessary.

For the AMCAD testbed, the levels of redundancy were limited to simplex and triplex for simplicity. Tasks can therefore request simplex (RECV) or triplex (TRECV) receives to consume data. A RECV call provides the data from the indicated mailbox to the task; a triplex TRECV returns a voted copy of the triplex data variable to the task.

The RECV system call is passed the mailbox number of the source variable and a pointer to a local variable to accept the data. As with SEND, the mailbox number provides access to the appropriate VIT entry. If the calling task is the valid consumer, the variable's producer/consumer flags are queried. RECV will loop until either the flags indicate that new data has been produced or a timeout expires. Upon exiting the loop, the data is copied from the VCM to the local destination variable. If the loop had timed out, an error is signaled to the RTMOS; otherwise the mailbox consumer flag is updated and sent to the VCM.

TRECV is similar to the RECV call, but with modifications for gathering and voting the triplex data items. The voted result data item is returned to the calling task. Because the AMCAD producer/consumer protocol only permits one producer and one consumer task per mailbox, TRECV cannot simply do a RECV on each of the three mailboxes. Rather, each task in a triplex set does a RECV on its associated mailbox, then simply copies the data from the other two mailboxes to special voting buffers. The TRECV routine can be programmed to delay for the producer/consumer flags of the other two variables if necessary. A triplex voting routine produces a result data item, which is stored in the task's local destination variable. Figure 4 shows the flow of data between triplex tasks.

Voting Algorithm The voting algorithm used in the AMCAD testbed is limited to triplex redundant data. If other levels of redundancy were desired, separate voting routines would be needed for each level.

The voting algorithm is implemented in a subroutine called by the TRECV system call. By using a separate routine to perform the triplex voting, the voting algorithm can be tailored or replaced as appropriate for the specific application. Each application may have its own particular demands for the voter which require unique routines. Some applications may require that the three values agree exactly, while other may settle for approximate agreement. Consistency checks on the data may be required to ensure that the voted value is reasonable.

The voting algorithm implemented for the AMCAD testbed, shown in Figure 4, is simply to demonstrate the concept. It employs simple reasonableness checks against the previous frame's data and amongst the triplex values themselves to
The zone is assigned a subset of the total application task load for the processor from affecting the rest of the system. Once the zone's task load must be reconfigured to redistribute the tasks to the remaining healthy processors after a failure.

After processor failure and removal, the zone's task load must be reconfigured to redistribute the tasks to the remaining healthy processors. Reconfiguration is similar to power-up task assignment, with the entire task load of the zone being reassigned to the new number of healthy processors. The assumption in AMCAD is that each processor has a local copy of the zone's entire software load, removing the need to transfer code between processors upon reconfiguration. Since the system data is shared by all processors through the VCM, reconfiguration results in minimal transient time. Fault masking through voting of redundant computations allows operation to continue reliably without restarting from the failure point.

The ability to reconfigure the task load upon removal of a faulty processor greatly enhances the power and flexibility of non-dedicated redundancy. By reconfiguring the tasks onto the remaining healthy processors, any redundant tasks disrupted by the failure are reestablished to maintain the desired level of redundancy despite the failure. No redundancy is lost until the number of processors falls below the desired redundancy. With static, dedicated redundancy, each failure reduces the available redundancy.

Benefits

Several important benefits may be realized through application of the non-dedicated redundancy concepts described here. First, application software development is simplified because the design is independent of the number or configuration of processors. Also, because the operating system performs the reading and voting of redundant data from the VCM transparent to the application, the software designer is freed from worrying about fault tolerance techniques. Verification and validation of the application is simplified because of the functional partitioning of the application into smaller subsystems.

Another benefit of non-dedicated redundancy in AMCAD is that the task-level redundancy and voting allows faults to be masked immediately rather than after an entire sequence of computation. Data can be voted entering each triplex task, thus detecting errors as they occur. An added benefit is the additional fault isolation capability provided by the task-level voting. Since errors can be caught when they occur rather than after the entire computation, there is a better opportunity to isolate the cause of the miscompare.

Non-dedicated redundancy and reconfiguration provide a flexible, powerful mechanism for reestablishment of triplexes after failures. Reassignment of the task load to the remaining healthy processors in the reconfiguration zone allows redundancy levels to remain intact, providing continued fault masking in the presence of faults.

By initially providing additional processing capability greater than needed by the application, all processors may be assigned a portion of the total task load and run at less than maximum capacity. As processors within the reconfiguration zone fail or are damaged, the remaining processors become more highly utilized. When maximum processor loading is reached, graceful degradation can occur. The lowest priority tasks can be left unassigned when this stage is reached.

The most important benefit is the higher number of arbitrary sequential failures that can be tolerated with non-dedicated redundancy. Static redundancy techniques can be combined with sparing, but the spare resources are dedicated to particular tasks or channels. With non-dedicated redundancy, all processing...
capability can be applied to all computing requirements within that zone. As a result, greater processor utilization and reconfiguration flexibility is possible. In effect, sparing occurs at the processing task level rather than the processor level.

Future Testbed Research

The basic concepts of non-dedicated redundancy have been implemented and tested in the AMCAD fault tolerant multiprocessor testbed using a simple application program. The application, a control surface reconfiguration algorithm called KI, was coded and replicated to examine the performance and correctness of the non-dedicated redundancy implementation. Preliminary testing indicates an overhead of roughly 20% for voting and redundancy management on the existing hardware. Also, a number of failure conditions were introduced to verify correct operation of the voting algorithm.

The AMCAD testbed provides a powerful testbed capability with which to further examine reconfiguration and non-dedicated redundancy techniques and their application to flight control and integrated vehicle management systems. Further study of the performance and efficiency aspects of reconfiguration is planned for the AMCAD testbed hardware. The testbed also enables further investigation of several key issues for applying multiprocessor architectures to flight critical systems. Of particular interest are: the functional partitioning of flight critical systems; the development, test, and maintenance of multiprocessor software; the requirements for byzantine resilience in the non-dedicated redundancy approach; a quantification of the benefits versus the performance penalty for task-level voting planes; and the potential for hardware support for the voting process.

Bibliography


