THE DESIGN OF A REAL-TIME SYSTEM FOR SIMULATORS AND TRAINERS

Borko Furht

Department of Computer Science and Engineering
Florida Atlantic University, Boca Raton, Florida 33431

ABSTRACT

In this paper, we describe the design of a host/target multiprocessor system with the distributed memory, which addresses a class of real-time problems that operate in a synchronized environment with rapid response to external events, such as real-time simulators and trainers. The unique needs of trainers and simulators are addressed through the implementation of a mirror memory and the distributed database supported by broadcasting writes and interrupts in a tightly-coupled host/target architecture. A special software development tool has been developed, referred as to the Application Worksheet, which provides real-time deadline scheduling and inter-processor triggering required in most simulator and trainer applications.

1. SYSTEM ARCHITECTURE

The prototype of the host/target multiprocessor with the distributed memory, shown in Figure 1, has been developed at Modcomp, where the author of the paper was a senior director of research and advanced development.

The system consists of a host processor, up to eight target processors, and various real-time I/O units. The distributed memory system, based on the concept of multiple slaves processors, allows the broadcast of data and interrupts over the VME bus. The host and target processors use the combination of industry-standard single board computers, such as Motorola 68030, 68040, and 88100. The I/O support is provided by high-speed parallel I/O controllers and various specialized controllers required in trainers and simulators, such as HSD, 1553B, DR11W, A/D, D/A, etc.

The mirror memory system consists of a dual-port memory with logic to create a slave function on the VME bus which supports a broadcast mechanism. The memory contains location monitors which can be programmed to generate interrupts across the VSB bus to the target computer. The combination of these two functions creates a data and interrupt broadcast mechanisms. The memory is partitioned into a read/write local memory and a read only distributed memory. Writes to the distributed memory occur across the VME bus. This allows the software to create a distributed database which is read across the VSB bus and updated across the VME bus.

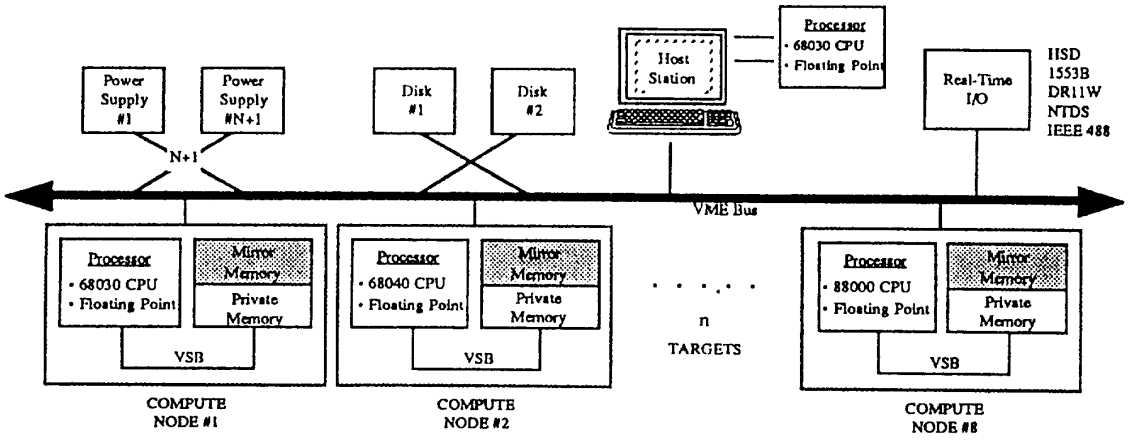

Figure 1. The host/target multiprocessor system with distributed memory

0-8186-4130-4/93 $03.00

2. SOFTWARE ARCHITECTURE

The software environment allows a user to develop, debug, test and deploy a distributed real-time application. The system software consists of a system host environment and a target computer node environment. The system host runs the REAL/IX operating system [1], while the target computer nodes operate under pSOS, a real-time executive.

A software development tool, the Application Worksheet (AW), has been developed, which provides real-time deadline scheduling and inter-processor triggering. With the AW, the user can coordinate the synchronization and execution of a large number of independent tasks, running on the host and targets. The AW implements the rate-monotonic scheduling (RMS) algorithm [2,3,4], by assigning priorities to periodic tasks based on their periods. The RMS assigns higher priorities to processes with shorter periods.

For a single-processor, the basic RMS theorem gives the condition when a set of n independent periodic tasks is schedulable (all tasks will meet their deadlines). This condition is defined as [2]:

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \ldots + \frac{C_n}{T_n} \leq n\,(2^{1/n} - 1)$$

where C_i and T_i are execution time and period of task Task(i), respectively. This theorem defines a sufficient, or the worst-case condition of the schedulability of a task set under the rate monotonic scheduler.

Application Worksheet

We applied the RSM algorithm on the host/target multiprocessor system by testing the schedulability of tasks on each processor. The RMS algorithm is incorporated into the Application Worksheet, which also permits the relocation of tasks from one processor to another, and automatically creates the execution models based on the current selections. It also enables the user to monitor frame counts, process execution times, and variables.

The AW contains the following four screens: the Task Schedule Screen, the I/O Schedule Screen, the Control Screen, the Task Monitor Screen, and the Data Monitor Screen.

The purpose of the Task Schedule Screen, shown in Figure 2, is to enter the names and trigger sources for all tasks in the system. The user can schedule tasks to run on the host or any available target processor.

After filling in the task name, the user selects a trigger for each task. A task can be triggered by a clock frequency, the completion of any other task, the completion of a scheduled I/O, an external interrupt, or its own completion. The task priority is automatically assigned based on the RMS algorithm: the relative frequency of the base trigger determines the priority. The base trigger is the first trigger derived directly from a clock. All tasks in a chain are set to the same priority by default. There are 15 different clock frequencies available as triggers, in the range from 0.1 Hz to 240 Hz.

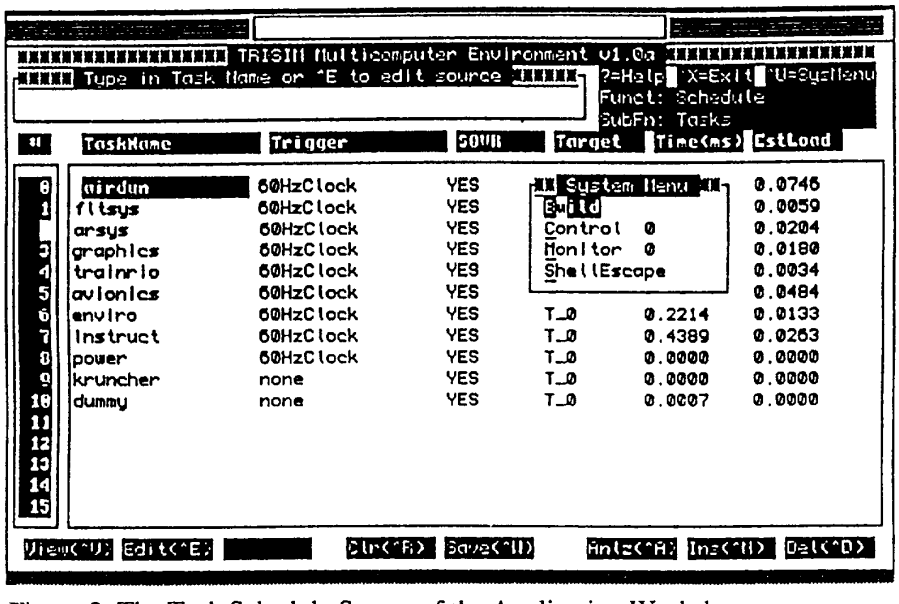

Figure 2. The Task Schedule Screen of the Application Worksheet

The user also selects a processor to execute a particular task. The task schedule screen calculates the percent of load for each task based on the estimated run time and the base trigger frequency. If the total load for a particular processor does not satisfy the RMS condition (1), the system will prompt the user, who can then change the task assignment and off-load the particular processor. If the run time of some tasks cannot be estimated, the tasks will be downloaded and run on the selected target processor, and the measured execution time will be used to assign the task priorities.

The I/O Schedule Screen is used for defining relatively constant I/O requests. In the proposed system, all I/O tasks run on the host processor. The I/O tasks can be triggered from any of the triggers available for normal tasks. Besides the name of each I/O task, the user also specifies several parameters, such as the buffer address, byte count, and device name.

The Control Screen is used to reset, start, and debug targets. Using this screen, the user can reset individual targets or all of the targets simultaneously, load and start the targets, and access the optional target software debugger.

The Task Monitor Screen, shown in Figure 3, is used for testing the application software as well as for monitoring system performance. Information presented on this screen includes the task name, the assigned target processor, the task priority, the run time, the task load (run time/period), and the overrun count. The run time is the actual amount of real time required to complete one iteration of the task. This includes all time spent running, sleeping, and servicing interrupts. This time is a function of the computational power of the target processor, the number of other tasks scheduled on this target, the priority of this task relative to the others, and the number of interprocessor interrupts. If the run time exceeds the base trigger period, then an overrun will occur. The overruns are detected, counted, and displayed on the Task Monitor Screen. If overruns are occuring to the particular task, the target processor must be offloaded by adding another processor.

The Data Monitor Screen is used to monitor variables within the tasks while they are running. In order to monitor a variable, the user must enter the name of the variable and the target upon it resides.

The AW also contains the action trigger database, which is an array of structures defining each task. There is one structure for each task, which includes information such as the task name, the task trigger, its priority, the execution time, and the processor node on which to execute the task. This database is loaded into the mirror memory, and is referenced and updated by the target processors during run time.

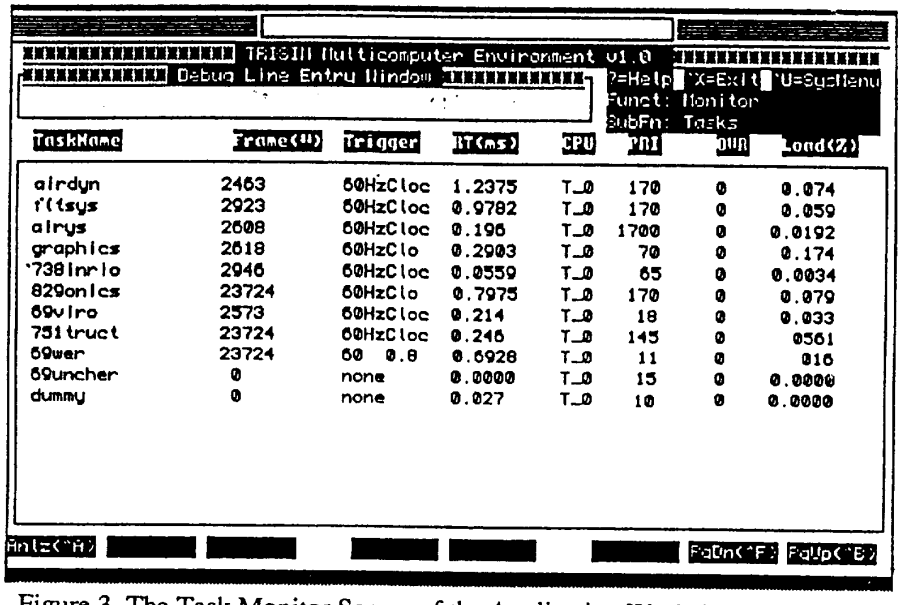

Figure 3. The Task Monitor Screen of the Application Worksheet

3. TYPICAL APPLICATIONS

The designed host/target multiprocessor system is intended for real-time simulator and trainer applications. A typical application is a flight simulator, as illustrated in Figure 4. All tasks are periodic and distributed to run on different target processors.

Programming the system

In a general-purpose computer, a typical C program, written for the UNIX operating system, designed to support the repetitive, cyclic programming required in simulation, is shown in Figure 5.

Note that the routines *initialize_clocks()*, *connect_to_shared_memory()*, *wait_for_trigger()*, and *notify_others()* are dependent on the operating system or computer hardware and mus be designed by the user. Thus, a great effort is required for building the platform, before the user can start work on the application routines, shown in boldface. If the user wants to change the frequency at which this program is triggered, the source code must be modified. In addition, if another program has to be scheduled to run each time this one is completed, the interprocess communications must be created for these purposes.

By using the Application Worksheet, each program is treated like a subroutine. If the user wants to change the trigger for any task, it is simply done in the Task Schedule Screen. The interprocess communications are built into the system, and require no effort on user's part. The same program from Fig. 5 running on the designed system is shown in Figure 6.

It is also possible to parallelize this program, and take the advantage of the multiprocessor environment. The functions *instruments()*, *engines()*, *radar()*, and *radio()* are unrelated. The only requirement is that these tasks execute within the scheduled tame frame, after which the results are written onto disk. Therefore, the program from Fig. 6 can be split into five separate programs and each program can be assigned to a different target processor. By providing the parallel execution of these tasks, the trigger frequency can be increased from 15 Hz to 60 Hz, achieving the better-quality simulation.

References

[1] B. Furht, D. Grostick, D. Gluch, J. Parker, G. Rabbat, and M.McRoberts, "Real-Time UNIX Systems: Design and Application Guide", Kluwer Academic Publishers, 1991.
[2] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", Journal of the Association for Computing Machinery, Vol. 20, No. 1, 1973, pp. 46-61.
[3] L. Sha and J.B. Goodenough, "Real-Time Scheduling Theory and Ada", Computer, April 1990, pp. 53-62.
[4] C. Warren, "Rate Monotonic Scheduling", IEEE Micro, June 1991, pp. 34.

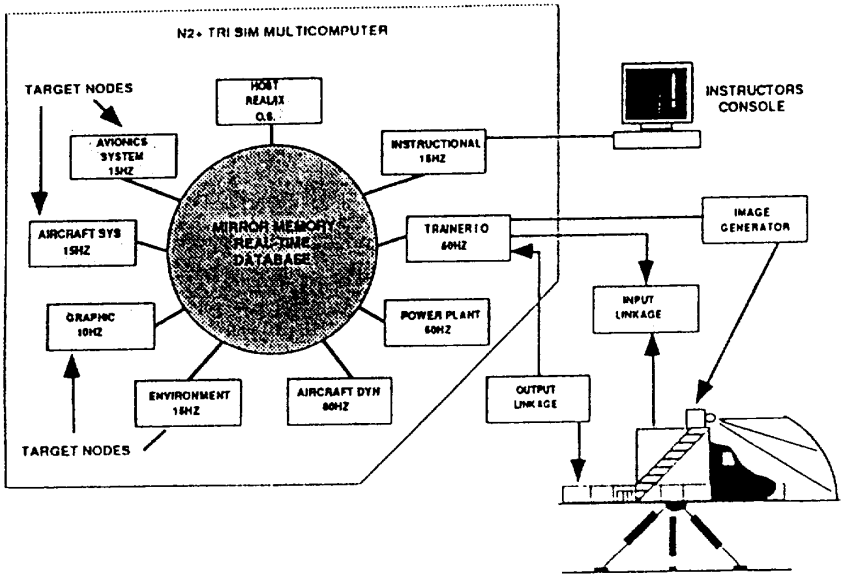

Figure 4. The flight simulator computing model

```
main()
{
        int fd;
        struct airplane *shared;

        initialize_clocks();
        shared = (struct airplane *) connect_to_shared_memory(0);

        fd = open("airplane",O_RDWR|O_CREAT);

        while(1) {
                /* call a library routine to wait for my next clock tick */
                wait_for_trigger(15HZ_CLOCK);

                instruments(shared);
                engines(shared);
                radar(shared);
                radio(shared);
                write_data_to_disk(shared);

                /* indicate that I have completed */
                notify_others();
        }
}
```

Figure 5. A typical C program which supports the repetitive cyclic programming required in simulation (general-purpose computer)

```
airsim()
{
        static int fd;
        static int init;
        static struct airplane *shared;
        if ( init == 0 ) {
                init =1;
                fd = open("myfile",O_RDWR|O_CREAT);

                /* get the address of mirror memory segment 0 */
                shared = (struct airplane *) get_mmseg(0);
        }

        instruments(shared);
        engines(shared);
        radar(shared);
        radio(shared);
        write_data_to_disk(shared);
}
```

Figure 6. The program from Fig. 5 rewritten to run under the Application Worksheet