Mwave/OS: A Predictable Real-time DSP Operating System

Jay K. Strosnider and Daniel I. Katcher
Department of Electrical & Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213

strosnider@ece.cmu.edu

March 1994

There is an industry trend in the development of single board computers based on DSP microprocessors. These microprocessors have specialized instruction sets that are optimized for DSP operations and architectures that are tuned for DSP program control. They also provide the structure necessary to support operating system functions. They are being used to supplement processing in PC and workstation environments with DSP functionality, such as telephony, audio and video processing, and fax and modem capabilities. This latter requirement to support multiple DSP functions concurrently has led to the expanded use of multi-tasking DSP operating systems.

DSP applications have **real-time requirements**, in the sense that individual tasks have individual time constraints that must be met for the system to operate correctly. For example, the system cannot afford to miss a deadline when performing a modem function, otherwise data will be lost. Researchers at Carnegie Mellon University (CMU) have been developing formalisms that allow application developers to *a priori* determine the timing correctness of their applications in multi-tasking environments. IBM's Mwave WindSurfer^{TM} card provided an excellent test case for this methodology. A timing model of their DSP operating system running on their underlying DSP was built and shown to accurately capture the timing performance of the Mwave card. This timing model enables the DSP developer to insure the timing correctness of arbitrary concurrency mixes of DSP tasks without explicit testing.

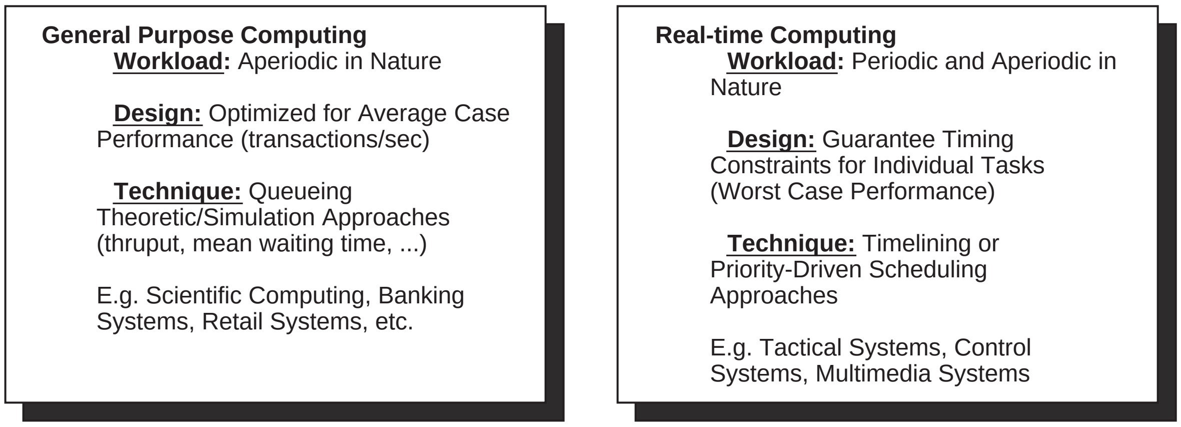
In the following article we briefly explore the differences between real-time and general purpose computing. We then discuss the basic approaches used to achieve predictable timing performance. A high level summary of our methodology, which bridges the gap between idealized scheduling theory and implementation of scheduling in operating systems, is provided. A scheduling (timing) model of the Mwave card is summarized. Lastly, we discuss the results of validation studies in which the Mwave/OS timing model was shown to be accurate to within 1% in predicting the actual timing performance of DSP applications running on the Mwave card. Thus the Mwave card provides a predictable execution environment for multi-tasking DSP applications.

Real-Time vs. General Purpose Computing

Figure 1 captures the key differences between real-time and general-purpose computing against the follow three criteria: Workload, Design Goals, and Response Time Determination Techniques. General-purpose (GP) computing environments are characterized by largely aperiodic workloads with loose, aggregate-level timing requirements. Real-time (RT) computing environments are generally dominated by continuous, periodic data streams from sensors, communication links, control loops, video, audio, etc., but also support aperiodic workloads. The key difference between GP and RT workloads is in the nature of their timing requirements. Whereas GP workloads typically have aggregate, average case timing requirements, RT workloads have explicit timing constraints that must be satisfied for each individual task.

The design criterion typically used for GP computing is to maximize average case performance with little or no concern for the worst case performance. In contrast, RT workloads require explicit timing guarantees per task which require worst case performance analysis on an individual task basis. The performance analysis techniques used in GP computing tend to be queueing theoretic and simulation oriented, whereas RT computing requires explicit, worst case timing analysis to assure the timing correctness of all tasks in multitasking environments.

The approaches for insuring real-time timing correctness can be broken down into two cat-



General Purpose Computing

Workload: Aperiodic in Nature

Design: Optimized for Average Case Performance (transactions/sec)

Technique: Queueing Theoretic/Simulation Approaches (thruput, mean waiting time, ...)

E.g. Scientific Computing, Banking Systems, Retail Systems, etc.

Real-time Computing

Workload: Periodic and Aperiodic in Nature

Design: Guarantee Timing Constraints for Individual Tasks (Worst Case Performance)

Technique: Timelining or Priority-Driven Scheduling Approaches

E.g. Tactical Systems, Control Systems, Multimedia Systems

Figure 1: General Purpose vs Real-Time Computing

egories: timelined and priority-driven scheduling, as illustrated in Figure 2. In timelined schedulers resource allocations are *statically bound* to a fixed, repeating time sequence called the timeline [1, 2]. Priority-driven approaches *dynamically* bind resource allocation at run-time using priority based arbitration. At any given time, the operating system grants the CPU to the highest priority pending task.

The priority-driven approaches are further subdivided into two categories, fixed and dynamic priority scheduling. For fixed priority schedulers the task priorities are static, ie., any time the task is active it arbitrates at the same priority for the CPU. In contrast, dynamic priority schedulers determine the priority of tasks based upon some dynamically changing criterion. like earliest deadline or least laxity. The earliest deadline approach assigns the highest priority at a given time to the task with the earliest deadline; thus, the priority of a task changes depending on its own deadline relative to the deadlines of other pending tasks.

Each of the approaches has its own strengths and weaknesses. Timelining has been used in the development of many real-time systems, because it is the easiest to understand and tends to have the lowest implementation overhead. However, the technique generally does not scale well to larger, more complex systems and has had life-cycle cost problems in the defense systems arena. Priority-driven approaches are more modern and have been gaining increasing favor in defense and real-time communities. Fixed priority scheduling is a more mature technology than dynamic priority scheduling. However, dynamic priority schedulers generally can guarantee all task deadlines at higher utilization levels than fixed priority schedulers. However, special care must be taken to ensure that a dynamic priority schedulers remains stable when when the system is overloaded.

It is interesting to note that major DSP OS vendors have each chosen a different approach for providing timing correctness. AT&T's VCOS DSP/OS uses a timeline scheduling approach, Spectron's SPOX DSP/OS uses fixed priority scheduling, and IBM's Mwave/OS uses dynamic priority scheduling. The differences provide an excellent opportunity to apply the formalisms developed at CMU to quantitatively evaluate the relative strengths and weaknesses of the three approaches. Such a study is currently being performed at CMU as part of a graduate class. The remainder of this article summarizes results to date in modeling and analysis of IBM's Mwave/OS.

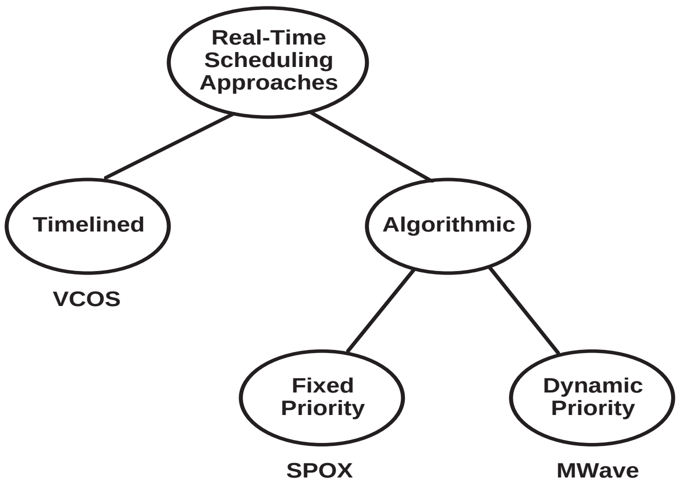


Figure 2: Summary of Real-Time Scheduling Algorithms

Operating System Scheduling Models

An operating system (OS) scheduling model is an abstraction which captures the timing properties of an operating system. These models capture the overhead and limited preemptability affects of real operating systems running on real hardware. IBM's Mwave/OS falls into the dynamic priority scheduling class (see Figure 2.) It implements the earliest deadline scheduling algorithm, a dynamic priority algorithm, which was shown to be an optimal dynamic scheduling algorithm by Liu and Layland [3]. Liu and Layland proved that as long as the utilization of the periodic task sets does not exceed 100%, all tasks are guaranteed to meet their deadlines. This may be expressed as

$$U_n = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1.0, \qquad (1)$$

where the C_i's are the task run-times and the T_i's are the task periods. It is easy to see that this scheduling algorithm is optimal - as long as one does not load the processor over 100% then all tasks are guaranteed to meet their deadlines. However, this result is subject to the following assumptions:

- Perfect, instantaneous preemption,
- Zero overhead costs.

Clearly no real operating system executes its services in zero time and is perfectly preemptable. Thus, one cannot apply the criterion in Equation 1 with any confidence. The OS scheduling model methodology bridges this gap between scheduling theory and its implementation in operating systems. To be accurate, these models must include the costs of any system services that the operating system provides. The most basic service is scheduling, but if synchronization or interprocess communication services are supported, such as mailboxes or event flags, they, too, must be characterized and captured in the scheduling models. For a detailed discussion of operating system scheduling models the reader is directed to [4, 5].

Mwave/OS Scheduling Model

Figure 3 provides a high level classification that covers most of the operating systems that are available today. Many RT OS's still rely on a periodic timer interrupt - these are timer-driven operating systems. However, if task scheduling tends to be keyed off a particular data stream that runs at a certain rate, then the OS can be characterized as event-driven. The two types of event-driven mechanisms are integrated and non-integrated. Integrated interrupts have the priority of the incoming interrupt matched to the software priority of the current application; thus, interrupts will only be handled if they correspond to a higher priority application. Non-integrated interrupts are always handled. Finally, the last aspect to consider is the design of the OS interrupt handling mechanism. Many OS's simply turn off or do not recognize interrupts when the OS is running (as opposed to the application running.) If this is so, then the OS is non-preemptable. However, if interrupts can be taken at any time, and the operating system can change direction and begin working on a different, higher priority request for service, then the OS is preemptable. In general, an OS that is non-preemptable for long segments can have an adverse effect on scheduling for tasks which have high frequency timing requirements. However, supporting finer grained preemption can result in a more complex implementation, which translates into higher overhead costs. Scheduling models provide a quantitative way in which to evaluate that trade-off from a real-time scheduling viewpoint.

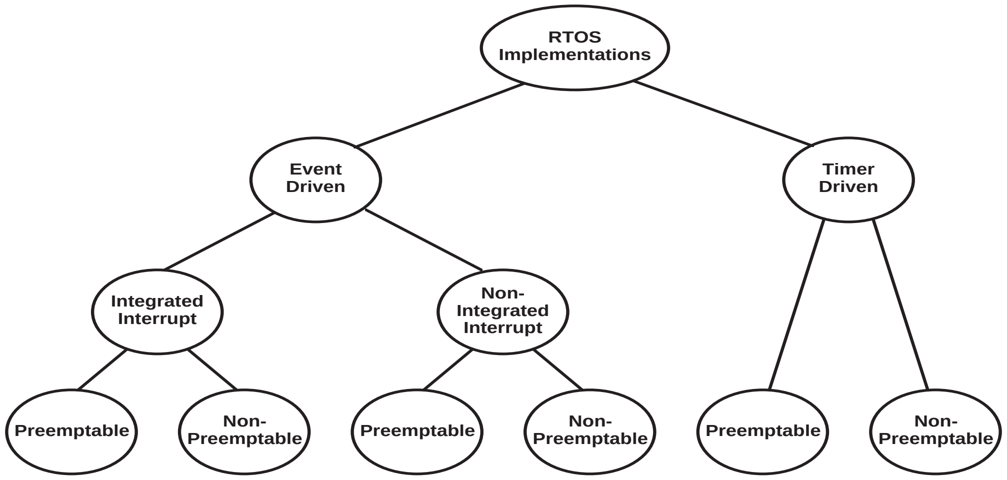


Figure 3: Classification of RT OS Implementations

The Mwave/OS system is a event-driven, non-integrated interrupt, preemptable operating system. It always handles incoming interrupts, and is almost instantaneously preemptable, with the exception of very negligible portions of its code. The Mwave system has three different event-driven interrupt sources. The rates of the interrupts are 9600 interrupts/sec (telephony applications), 8000 interrupts/sec (telephony and audio applications), and 1378 interrupts/sec (44.1 KHz CD rate with 32 buffered samples.) Each real-time task in Mwave specifies the interrupt source to which it is tied, the number of interrupts in its period (called frame size), and its worst case execution time. For instance, a task tied to the 8 KHz interrupt source with a frame size of 4 and a worst case execution time of 200 cycles, will run every 4/8000=500 $usecs$, and should use 200 cycles or less every time it runs. Tasks which share the same interrupt source, and have the same period, are grouped together into a common frame manager. Frame managers are scheduled by the Mwave operating system. A frame manager sequences through its list of tasks once per period serially until they are all completed. In this article, we assume that the transition time between tasks in a frame manager is negligible. In the discussion below, frame manager i is specified by T_i, its period, and C_i, the total worst case execution time of all its tasks.

When an interrupt occurs for a given source, an interrupt handler is invoked, which examines the frame manager at the head of the idle list associated with that interrupt source. The idle list is a list of frame managers that are waiting for their next start time. If their start time is past, they are ready to run and the scheduler is invoked. Otherwise, the interrupt handler exits. Thus a constant cost of handling an interrupt from interrupt source i is C_{inti} and the utilization of that interrupt source is $\frac{C_{inti}}{T_{inti}}$.

When the scheduler does run, it moves the frame manager that is ready to run to the run queue, where it is queued by priority as determined by its next deadline and the earliest deadline first algorithm. If this newly queued frame manager is of higher priority than the currently active frame manager, a preemption occurs; otherwise, the scheduler exits and the active task will continue to execute. On a preemption, or context switch, the state of the active frame manager and task is saved onto the run queue and the new frame manager is loaded onto the CPU. We let $C_{activate}$ be the worst case cost to move frame managers from the idle queue to the run queue and $C_{preempt}$ be the cost for performing a preemption in the case where a frame manager has higher priority than the currently running frame manager.

The only other scheduling cost occurs when a frame manager exits after it has finished its computation in its period. At that point, the frame manager is restored onto the idle queue, its next start time is computed, and the frame manager that was previously preempted is restored back onto the processor. We call this worst case cost C_{exit}. $C_{activate}$, $C_{preempt}$, and C_{exit} are each a function of the number of frame managers on each interrupt source and can be calculated by finding the worst case path through the operating system code and calculating the number of cycles it takes. Thus, for a frame manager to run, it must incur total cost equal to $C_{activate} + C_{preempt} + C_{exit}$.

Another term in the model of the Mwave system accounts for the DMA load required by the tasks when they communicate with the external interfaces or the host processor (a PC.) DMA is limited to one of every four cycles on Mwave. This is reflected in the feasibility test by adding a term for DMA load with a utilization of 25%. Because this is a heavy penalty, we are currently working to better specify the DMA requirements of tasks such that this penalty can be reduced. Finally, a high frequency frame manager can have its processing interrupted by the interrupts and arrival of lower priority frame manager that have less stringent timing requirements. We add a *blocking* term that reflects this cost, because this analysis verifies that the task set will work in the *worst case*. This term is the cost of activating $n - 1$ frame managers in the period of the highest frequency frame manager, which we denote as T_1.

Putting it all together, the final equation for the Mwave system and a set of n frame managers looks like this:

$$\sum_{j=1}^{3} \frac{C_{int\,j}}{T_{int\,j}} + \sum_{i=1}^{n} \frac{C_i + C_{preempt} + C_{exit} + C_{activate}}{T_i} + \frac{C_{DMA}}{T_{DMA}} + \frac{(n-1)\,C_{activate}}{T_1} \leq 1. \qquad (2)$$

Though seemingly complicated, the application developer and end-user need never see this detailed aspect. This scheduling feasibility test can be easily programmed for evaluation either on or off line. It could easily be incorporated into a tool that helps application developers put together DSP applications.

Another important real-time feature of the Mwave system is its hardware supported cycle counter. The Mwave system allows the application developer to specify the exact number of cycles that are required by each task in the worst case. This number is loaded into a register on the processor when it begins to execute. A cycle counter tracks the number of cycles actually used by the task. Mwave/OS uses this feature to contain task overruns and isolate timing faults to the offending task. This feature is also extremely useful in debugging the timing characteristics of real-time applications. This feature, along with its tight well characterized OS, made MWave/OS an excellent test case for bridging the gap between scheduling theory and its realization in OS's.

Validation of the Mwave/OS Scheduling Model

The Mwave/OS scheduling model has been validated on the actual Mwave card with a synthetic task set. The synthetic task set consisted of cycle burners, tasks that loop on the CPU for a specific number of cycles. We scaled the task set to its maximum schedulable utilization, the point at which any increase in cycle requirements by any task would cause at least one task to miss a deadline. This number agreed closely (to within 1%) with the utilization predicted by the model.

Providing an accurate set of feasibility tests for these systems is an ongoing challenge. A graduate level class at CMU is currently evaluating the three different DSP systems mentioned in this article, and developing models for each of them. At the end of the semester, the output of this effort should be a set of detailed models that will allow application developers to test their timing requirements with confidence on a diverse set of systems.

References

[1] T. P. Baker and A. Shaw, "The cyclic executive model and ada," *The Journal of Real-Time Systems*, vol. 1, pp. 7–25, June 1989.

[2] C. D. Locke, "Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives," *The Journal of Real-Time Systems*, vol. 4, pp. 37–53, March 1992.

[3] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, vol. 30, pp. 46–61, January 1973.

[4] D. Katcher, H. Arakawa, and J. Strosnider, "Engineering and analysis of fixed priority schedulers," *IEEE Transactions on Software Engineering*, vol. 19, September 1993.

[5] H. Arakawa, D. Katcher, J. Strosnider, and H. Tokuda, “Modeling and validation of the real-time mach scheduler,” *1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1993.