Implementing a Predictable Real-Time Multiprocessor Kernel – The Spring Kernel.

L. D. Molesky, K. Ramamritham, C. Shen, J. A. Stankovic, and G. Zlokapa

Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

May 1990

1 Introduction

The Spring paradigm [6] advocates *predictable* [5] real-time computing. The purpose of predictable real-time computing is to allow the timing properties of both individual tasks and the overall system to be assessed. The construction of predictable systems can be viewed from the bottom up – predictable architectural features facilitate the construction of predictable OS software, which leads to building predictable real-time application software.

Among the architectural requirements for predictable real-time systems are bounded instruction execution and memory access times, and bounded inter-process and inter-node communication costs. These architectural features facilitate the construction of predictable OS features such as bounded dispatching, scheduling and synchronization costs, bounded OS primitives, and bounded code execution times.

The design and implementation of the Spring real-time multiprocessor kernel supports these features. In this short paper we describe the Spring kernel support for bounded dispatching, scheduling and synchronization.

2 Overview of the Spring System

The Spring system [6] is physically distributed and is composed of a network of multiprocessors. Each multiprocessor contains one or more application processors, one or more system processors, and an I/O subsystem. System processors offload the scheduling algorithm and other OS overhead from the application tasks both for speed, and so that this overhead does not cause uncertainty in executing guaranteed tasks. All system tasks are resident in the memory of the system processors. The I/O subsystem is a separate entity from the Spring kernel and it handles non-critical I/O, slow I/O devices, and fast sensors.

Version 1 of the Spring kernel concentrates on the multiprocessor aspect of the Spring system.

This work is funded in part by the Office of Naval Research under contract N00014-85-K-0398 and by the National Science Foundation under grant DCR-8500332.

A Spring node is a multiprocessor consisting of up to five [1] Motorola 68020 based MVME136A boards. The MVME136A boards support features which are typical of shared bus multiprocessors – an asynchronous bus interface, architectural support for *test-and-set* like operations, and a local memory. This memory can either be accessed remotely over the VME bus by (typically) another processor, or locally by the processor which has mapped this local memory. Additional support for multiprocessing is provided through the use of the MPCSR (MultiProcessor Control/Status Registers). The MPCSR provides the ability to generate interrupts on a selected board, and/or a simultaneous interrupt to multiple boards.

One node consists of a system board (which executes the scheduler) and multiple application boards. The dispatchers, one per application board, are responsible for the dispatching of application tasks. The scheduler and dispatcher processes are thus designed to run in parallel. External events represent invocations of application tasks with arrival times, deadlines, resource requirements, and other attributes. When a task arrives, the scheduler attempts to dynamically guarantee that the new task will meet its deadline. As tasks are guaranteed, the scheduler adds them to a system task table (STT); these tasks are also linked into dispatcher queues. Since the STT resides on the system board, a dispatch queue reference performed by the dispatcher accesses the shared bus.

Tasks are classified into three categories – critical, essential, and non-essential [6]. The online guarantee is used for *essential* tasks. These tasks have deadlines and are important to the operation of the system, but will not cause a catastrophe if they are not finished on time. It is necessary to treat such tasks in a dynamic manner as it is impossible to reserve enough resources for all contingencies with respect to these tasks.

The memory model underlying the Spring kernel design is a *local* memory model. Each processor is equipped with a local memory module; every processor can also access all other memory modules via a common bus. This models multiprocessor systems in which each processor has local memory for task code and private resources, while at the same time there are other resources, such as shared data structures, files, and communication ports, which can be used by tasks residing on different processors. This model does in fact match the 68020 based multiprocessor architecture that Spring runs on. Since each processor has its own local memory, the assignment of tasks to processors, done statically, determines which processor's memory the task code is resident. To avoid unpredictable blocking of tasks due to resource contention at run time, our scheduling algorithm integrates tasks' timing constraints and their resource requirements [4].

[1] Although eight slots exist on the backplane, only five boards can be used because of power supply limitations.

3 Foundations of the Spring OS: Scheduler and Dispatchers

Predictability of the underlying real-time OS is necessary to achieve predictability of software (application tasks) running on top of this OS. This section describes the design and implementation of significant components of the multiprocessor real-time OS – the *scheduler* and the *dispatchers*. To ensure predictability of application tasks, both the scheduler cost and the dispatching costs must be bounded. Version 1 of Spring supports a scheduler which executes in time O(N) [4] where N is the number of tasks at the node. However, the execution time of the scheduler is capped to a fixed worst case time. This will be discussed further in section 3.1. The dispatching cost is bounded by a constant. Multiple dispatchers operate concurrently with no inter-dispatcher interference. Dispatchers and the scheduler require concurrent access to the STT. Correctness of this access is maintained via the use of critical sections, while predictability is ensured by constructing all critical sections to execute in constant time.

The STT is a key data structure in the Spring kernel. Tasks which have been guaranteed are placed in the STT by the scheduler. The STT, residing in the system board memory, contains dynamic task (invocation) information and information for OS management and scheduling. The OS management and scheduling fields include fields for maintaining scheduler data structures, as well as fields for constructing linked lists which order the STT.

Concurrent execution of dispatchers is achieved by partitioning the STT based on the processor to which tasks are assigned. Each per-processor partition of the STT is known as the dispatch queue. Since a task is assigned to exactly one processor, the multiple dispatcher processes can concurrently access their dispatch queues without interference (the intersection of all dispatch queues is null). To facilitate correct and efficient dispatching, the STT is sorted according to the scheduled start time of each task. This design provides a dispatcher with a constant time access to its dispatch queue to determine which task to execute next. Concurrent execution of the scheduler and the multiple dispatchers is achieved by reserving a set of tasks for each dispatcher. The scheduler is not free to reschedule the tasks reserved for the dispatchers. Thus each dispatcher has tasks to execute while the scheduler is attempting to reschedule in order to guarantee a new task.

The method of partitioning tasks between the scheduler and the dispatchers involves the calculation of a *cutoff line*. Once an upper bound of the scheduler's cost for guaranteeing a task is determined, this cost is added to the current time to determine the cutoff line. All tasks having a scheduled start time prior to the cutoff line are reserved for the dispatchers, and thus cannot be rescheduled.

The online guarantee is designed to allow concurrent operation with the multiple dispatchers. When a new, dynamic task arrives, the guarantee algorithm is invoked. The online guarantee does not alter the current schedule, it instead operates on copies of the task invocation information. This convention facilitates the return to the original STT if the guarantee fails.

3.1 Periodic Invocation of the Scheduler

Since the system processor is used for all system tasks, to ensure responsiveness for all system level activities, the scheduler as well as other tasks are invoked periodically. (In addition, if possible, the scheduler may also be invoked asynchronously upon the arrival of a new task.) Thus, the scheduler executes for, at most, a fixed amount of time, namely for the computation time allocated to it, every period. Periodic scheduler invocation affects the design of the real-time OS which bounds the task *guarantee* time, and the runtime (online) decision of how many additional tasks (extracted from the candidate queue, the queue of tasks waiting to be guaranteed) should be used for the next guarantee invocation.

Given that the scheduler has execution time which is O(N), (the number of tasks being processed), knowing the constant of proportionality and the fixed overheads, we can determine how many tasks can be guaranteed by the scheduler during each periodic invocation. Suppose this is Nmax. We call Nmax the "cap on the length of the STT". Suppose at a given time, the number of tasks already in the STT is S. Then at most Nmax - S tasks from the candidate queue can be considered for guarantee at this time. (Since the scheduler is invoked periodically, between two invocations, multiple task requests may get enqueued in the candidate queue.) Of course if not all Nmax - S tasks are guaranteeable, how to choose a subset of this set for subsequent attempts is an interesting question.

It is likely that the task invoking a nonperiodic task will impose a deadline not only on the invoked task, but also on the guarantee. In addition, some invokers may desire to know how long to wait to find out if the invoked task has been guaranteed or not. In the former case, whenever the scheduler is invoked, it has to examine the candidate queue to see if deadlines on the guarantee can be met given the discussion above. In the latter case, knowing the current length of the STT, etc., it is possible to determine the scheduler's response time.

3.2 Maximizing Concurrency between the Scheduler and Dispatchers

While the scheduler's execution time is a function of N, the number of tasks in the system (capped by Nmax), the dispatcher execution time need not be dependent on N. Because the worst case dispatching costs must be included in each task's worst case computation time, an efficient *worst case* design of the dispatcher is very important. In a multiprocessor implementation, worst case blocking time (in our case due to mutual exclusion with critical sections) can be the overwhelming cost of the dispatcher. Version 1 of the Spring multiprocessor OS uses dispatchers with constant worst case computation times i.e., the worst case computation times of tasks are *not* effected by the number of tasks in the system.

When an application task completes its execution, it must be deleted from the system. The most natural implementation is to have the local (running on the same processor where the task

just completed) dispatcher delete the finished task from the system. This is not however the best implementation when the predictability of the multiprocessor OS is important, since, in order to maintain correctness with the scheduler, this design forces excessive mutual exclusive access to STT by the dispatchers. Specifically, if dispatchers were allowed to perform the deletions, the computation of the cutoff line would be required to be in a critical section (since pointers could become invalid during this computation). The computation of the cutoff line requires O(N) steps if the tasks are in a linked list, or O(log N) steps if the tasks were arranged contiguously. Thus, the scheduler could have locked the dispatch queue immediately prior to a dispatcher, causing the dispatcher to wait for an amount of time that is a function of the number of tasks in the system. This is unacceptable.

By having the scheduler, instead of the dispatcher, delete tasks from the STT, the worst case computation time of the dispatcher can be significantly reduced. The mechanics behind the convention of task deletions performed by the scheduler involve separate maintenance of dispatch queue pointers by the scheduler and the dispatchers. When a dispatcher notices that a task has finished, it implicitly marks the finished task by altering the head of the dispatch queue. The scheduler maintains a separate (shadow) copy of the dispatch queue head which is never altered by the dispatcher. When the scheduler is invoked, it first deletes all tasks which lie between the dispatch queue head and its shadow. Mutual exclusion is reduced to constant time – only modifications of the dispatch queue head need be done inside a critical section.

4 Real-Time Semaphores – Low Level Support for Predictability

The system components that are potentially the most elusive to guarantee predictability are those low level components which are shared by the multiple processors. On a multiprocessor, both shared memory and the shared bus connecting the processors fall into this category.

In a multiprocessor system, unless *bus access* time is bounded, any reference to remote memory cannot be predictable. For this and other reasons (discussed in [3]), any predictable real-time system which uses the asynchronous VME shared bus must be configured in round-robin mode. Round robin mode alone with processors busy-waiting on the semaphore (usually implemented with test-and-set) is however not sufficient to provide bounded waiting. It can be shown [3] that one or more processors can starve when two or more processors contend for a semaphore: It is possible for a subset of the processors to perpetually exchange the lock, starving one or more processors waiting for the lock.

To solve this problem, we have developed solutions for the construction of *real-time semaphores*[3] – semaphores which efficiently support bounded access. A software solution which improves the *bounded waiting* solution given in [1] as well as a hardware solution have been developed. The real-time semaphore is based on the P() and V() operations [2], using an extended test-and-set like operation, *test-and-set-or-branch*. The construction of real-time semaphores is

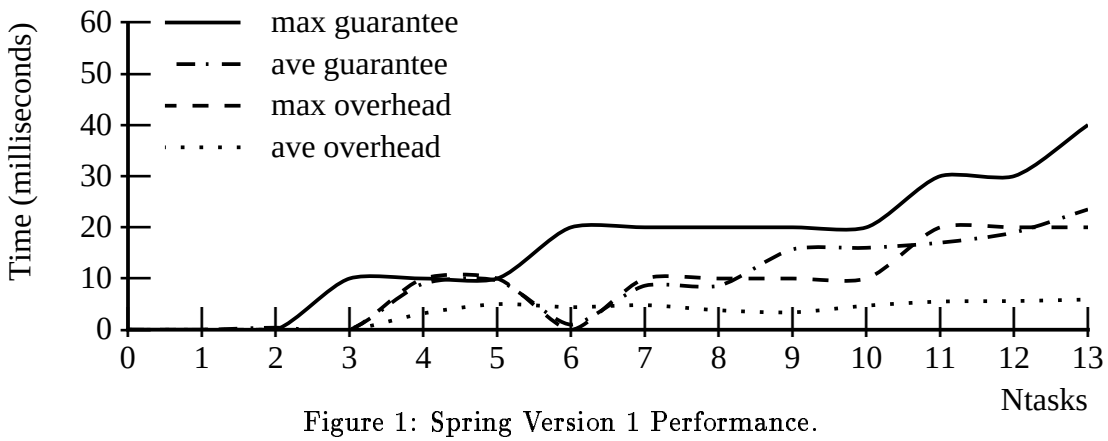

Figure 1: Spring Version 1 Performance.

based on the *Deferred Bus Theorem* (see [3] for the proof of the theorem):

> If the total worst case non-bus master time of the busy-wait loop (in P()) is less than the best case bus master time of the release instruction, and if processor p_j is the closest processor (in the round robin ordering) busy-waiting for semaphore s when processor p_i releases s (in V()), then p_j will be the next processor to acquire s.

Operations for enforcing mutual exclusion operations such as P() and V(), if constructed in a bounded fashion, can provide the framework for other, higher level, bounded operating systems primitives. This boundedness forms a basis for the predictability of the Spring real-time multiprocessor OS.

5 Performance

The performance of the scheduler (running in a 16 MHz. 68020) in Spring version 1 is illustrated in figure 1. Both the average and worst case computation times of the guarantee algorithm and the overheads are plotted. The costs of the guarantee algorithm are separated from the costs of the overheads, the total scheduler cost being the sum of the two. The overheads consist of scheduler activities before and after invocation of the guarantee algorithm (such as the computation of the cutoff line and task deletions). The guarantee algorithm, as described in [4], is invoked with no backtracking for a system with seven resources.

As discussed in section 3.1, the periodic invocation of the scheduler imposes a fixed computation time for the scheduler to run. Depending on the selected period and length of this fixed computation time, a cap on the maximum number of tasks which are guaranteed in this fashion (using the heuristic guarantee algorithm described in [4]) will be derived. Practical optimizations are currently underway, and include alternative scheduling algorithms, and restricted data structure access. One scheduling optimization would be, instead of performing a total reschedule of all tasks in the system, attempting to insert a single task into the existing schedule. The examination of only portions of

sorted system tables is an area of optimization pertaining to restricted data structure access. By speeding the guarantee, these optimizations may allow us to deal with more tasks than the more general techniques which have been implemented.

6 Conclusion

Our approach to constructing a real-time OS is to achieve predictability from the bottom up. We have discussed how bounded access to a shared bus facilitates the construction of real-time semaphores. Real-time semaphores in turn form a foundation for the construction of a *concurrent*, predictable real-time multiprocessor OS. At the next level, the predictability of user level tasks is facilitated by the predictable OS. Subtleties arising in supporting the online guarantee complicate the construction of a predictable multiprocessor OS which is concurrent. These subtleties are resolved in part by offloading activities from dispatcher to the schedulers, integrated with judicious use of critical sections by the real-time OS. The feasibility of this approach has been demonstrated with the shared bus multiprocessor implementation of Spring version 1.

References

[1] J. E. Burns. Mutual Exclusion with Linear Waiting using Binary Shared Variables. *SIGACT News*, 10(2), Summer 1978.

[2] E. W. Dijkstra. The Structure of the "THE"-Multiprogramming System. *Communications of the ACM*, 11(5), May 1968.

[3] L. D. Molesky, C. Shen, and G. Zlokapa. Predictable Synchronization Mechanisms for Multiprocessor Real-time Systems. Technical Report 89–106, University of Mass., November 1989.

[4] K. Ramamritham, J. A. Stankovic, and P. Shiah. O(n) Scheduling Algorithms for Real-Time Multiprocessor Systems. In *the 9th International Conference on Parallel Processing*, June 1989.

[5] J. A. Stankovic. Misconceptions About Real-Time Computing. *IEEE Computer*, 21(10), Oct. 1988.

[6] J. A. Stankovic and K. Ramamritham. The Spring Kernel: A New Paradigm for Real-time Operating Systems. *Operating Systems Review*, 23(3), July 1989.