Kernel-Level Threads for Dynamic, Hard Real-Time Environments*

Marty Humphrey, Gary Wallace and John A. Stankovic
Department of Computer Science
University of Massachusetts, Amherst, MA 01003
Email: HUMPHREY@CS.UMASS.EDU

Abstract

The design of a kernel-level thread package for dynamic, hard real-time environments is presented. A highly integrated design is used to ensure predictability. A system description language and real-time programming language are used to specify key properties of threads and thread groups. For a thread, this includes whether or not the thread spawns other threads at run-time, the type of performance guarantee the thread requires, how the thread interacts with other threads, and what processors the thread may execute on. A predictable kernel uses this information along with on-line dynamic guarantees to ensure predictable execution of threads. The first phase of the thread package has been implemented and performance measurements have indicated a 66% improvement in context switching costs.

1 Introduction

In many non-real-time operating systems, threads have been adopted as the programming model. As opposed to the UNIX-like process model, a thread model separates the address space from the flow of execution within the address space (the thread). The thread model offers many benefits over the process model. Threads allow the user to more easily develop a program that is inherently asynchronous; certain programs are better structured as sets of mostly independent sets of computations. Threaded programs are highly portable, because the number of threads in the design of the program can be independent of the number of processors available for executing the program. From an operating-systems perspective, threads offer various performance benefits, because switching from one thread to another thread in a single address space does not require remapping the address space.

Some attempts at developing real-time threads are based on modifications to non-real-time kernels that support threads. POSIX.4 [1] is a real-time extension to POSIX; definitions within POSIX.4 are directly aimed at augmenting the threads definition (PThreads [2]) with real-time properties. The threads of Real-Time Mach [3] are based on the Mach kernel [4].

The problem with founding a real-time threads package on the existing threads of a non-real-time kernel is that a key property of the real-time thread, ***predictability,*** can be difficult to achieve as the thread interacts with the non-real-time components in the kernel. In a real-time system, an action is said to be predictable if the results and duration of the action can be foreseen before the action is executed. Predictability is important for real-time threads because hard deadlines can be missed if actions such as the creation and scheduling of a thread take more time than expected. For example, predictability of memory accesses in a real-time thread in Real-Time Mach is not consistent with the time-sharing model of Mach. Mach uses a lazy evaluation scheme, in which objects are not actually allocated until after a page fault occurs. This memory management model is unacceptable for a real-time thread because accesses can take an unpredictable amount of time. Another feature of Mach that is not appropriate for a real-time thread is FIFO ordering in system-level queues, which can lead to unbounded delays [3]. Overall, the key difficulty in this approach for a real-time threads package is determining the subtle interactions of the real-time paradigm with the underlying time-sharing paradigm.

An alternative approach taken in this work is to base a real-time threads package on a predictable real-time kernel and associated development tools. The goal of this work is to create a real-time threads package for the Spring real-time operating system [5]. An integrated real-time threads solution has been developed that achieves predictability by involving an off-line specification language, a real-time programming language, kernel boot time options, and the run-time kernel. This package co-exists with a guarantee-based, on-line dynamic scheduler. This package avoids the insidious effects of timesharing operating systems such

*This work is funded by the National Science Foundation under grant IRI-9208920.

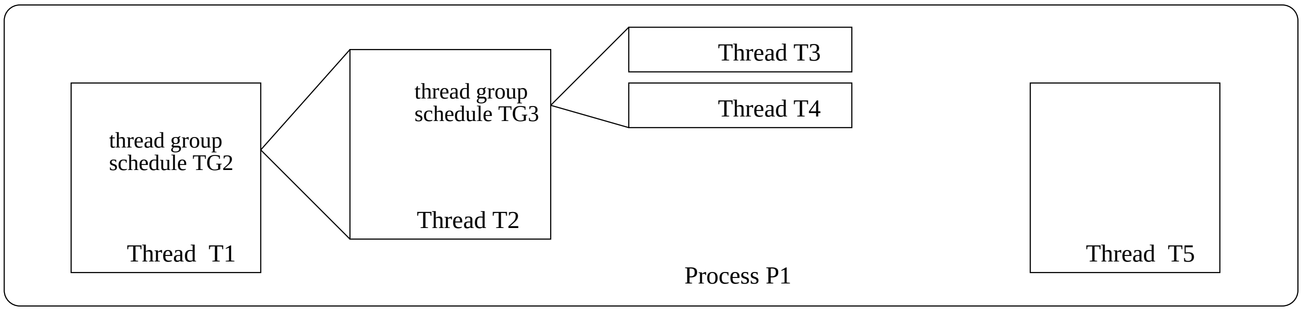

Figure 1: A Process Model

as a full POSIX-based operating system or Mach.

The contribution of this paper is the presentation of an integrated design of the real-time threads package that fully supports predictability. Important properties of a thread and thread group can be expressed in the system description language and real-time programming language. These assertions are used by the kernel to meet performance requirements. An example of a property that can be declared is whether or not the thread spawns another thread. A thread is ***sterile*** if it does not spawn another; a thread that spawns another is ***fertile.*** By recognizing that a thread is sterile, the kernel can perform certain optimizations in comparison to its support of fertile threads. Another declaration is the type of performance guarantee the thread requires. A fertile thread can require a ***Dependent*** guarantee, which means that it should not be scheduled unless every spawned thread can also be scheduled. An ***Independent*** guarantee does not require this. Other properties can be declared; these properties collectively enable the kernel to ensure predictable execution.

In the discussion we present the design of the real-time threads package that includes both sterile and fertile threads. Since to date we have implemented only sterile threads, the implementation discussed in this paper focuses on sterile threads. The value of sterile threads is both a new, predictable programming model available for application programs and improved kernel-level performance in the context switch. Results show that the context switch between threads in the same address space can require at most approximately one-third as much time as the context switch before threads were implemented.

The organization of this paper is as follows. Section 2 covers key elements of the Spring system and introduces terminology that will be used in the rest of the paper. Section 3 presents the basic language constructs and semantics for real-time threads in the Spring-C language. Section 4 describes implementation issues and the current status of the implementation. Section 5 describes various properties of our real-time threads. Section 6 presents related work in three real-time thread packages. Section 7 contains the conclusion and future work.

2 Background and Terminology

This section introduces real-time thread concepts and terminology, and then reviews the Spring real-time operating system.

2.1 Terminology

From the perspective of the real-time system programmer, the fundamental unit is the ***process.*** A process consists of an ***address space*** and one of more flows of execution through the address space, which are known as ***threads.*** If, at run-time, a thread spawns (also known as "forks") another thread, then the two threads have a ***parent-child relationship***—the thread that spawns another thread is called the ***parent thread,*** and the spawned thread is called the ***child*** thread. A thread that spawns another thread is called ***fertile,*** and a thread that does not spawn another thread is called ***sterile.***

The real-time system designer defines one or more ***thread groups,*** which are sets of threads that are related by precedence. A thread group is an "action" that the real-time system can take. These actions can either be periodic or aperiodic. In general, these actions can be viewed as either directly responding to an event in the environment, anticipating and checking for an event in the environment, or preventing an event in the environment. Each thread group has a deadline, a release time (also known as ready time), resource requirements, precedence constraints, and value, which are either defined statically or dynamically.

Figure 1 illustrates the static, compile-time structure of an example process, ***Process P1.*** Assume that there have been 4 thread groups defined: ***TG1,*** which is defined as the execution of thread *T1; TG2,* the execution of thread *T2; TG3,* the execution of thread *T3* followed by the execution of thread *T4;* and *TG4,* the execution of thread *T5.* Each thread group represents a distinct action that the real-time system can take. As shown in Figure 1, at run time, the execution of thread *T1* spawns thread group *TG2.* The execution of thread *T2* spawns thread group *TG3.*

From the perspective of the operating system, the fundamental unit is the ***task.*** While a thread can be viewed as a flow of execution that dynamically acquires and release resources at run-time, a task acquires all of its resources at the beginning of execution and releases all of them at the end of execution. A thread is translated at compile time into one or more tasks; in the case of more than one task, tasks are related by precedence. An example of the translation of a thread to tasks is shown in Figure 2. Each task is

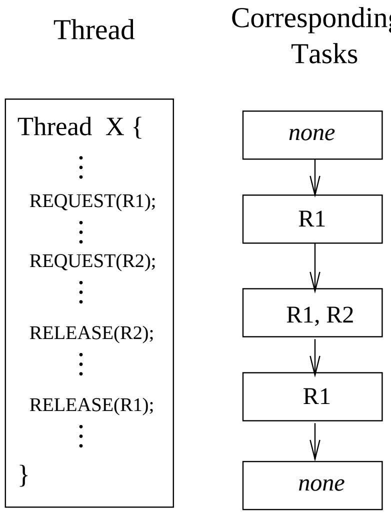

Figure 2: Mapping a Thread to Tasks

labeled with the resources it requires. An arrow from one task to another indicates a precedence relationship. Tasks are used by the Spring scheduler, which plans the use of resources (including the CPU) in a reservation-based scheme based on the worst-case execution time (WCET) of each task. The WCET of a task is also computed at compile time [6]. Predictability in the operating system is a direct consequence of realistic estimate of the WCET, predictable switching between tasks, and predictable duration of scheduling.

A real-time system can be described in terms of the environment in which it is intended to operate, which can be related to the type of deadlines associated with the actions of the real-time system. Roughly speaking, if the action must be completed by its deadline in order to obtain any value for it, the deadline is ***hard.*** If there is a timing constraint associated with the action, but if there is still value to completing the action within a reasonable amount of time after the deadline, the deadline is ***soft.*** Multimedia systems, while not quite fitting this definition, often involve soft deadlines. Actions that are not time-dependent are ***non-real-time*** actions. If the frequency of all events can be adequately predicted at design-time with acceptable uncertainty, the environment is said to be ***static;*** an off-line design for the real-time system is usually acceptable. If the events and frequency of events in the environment cannot be determined at design-time in terms of minimum and/or maximum interarrival time, the environment is said to be ***dynamic.***

The type of real-time threads that this paper will deal with are those for a dynamic, hard real-time environment.

2.2 Spring System Architecture

A single Spring node is a shared-bus multiprocessor. The current implementation of the Spring node consists of four Motorola 68020-based MVME136A boards in a VME cage. One board is referred to as the System Processor (SP), and the other three boards are Application Processors (APs), AP_1, AP_2, and AP_3. The SP is devoted to system activities such as scheduling and interacting with outside entities (such as a High-Level Scheduler [7]), while the APs execute the schedule created by the SP.

The physical memory resident on each MVE136A board is available for local reference by the processor on the MVE136A board, and can be accessed by another processor through the VME bus. In the current implementation, each MVE136A board has 4 MBytes of physical memory. The time for a 68020 to read or write from memory resident on its MVME136A board (thus, not requiring the VME bus) is 1 μs, and, on an unsaturated bus, 2 μs to access another board's memory.

In general, the execution of user code on the APs begins with a request from an outside entity to the Spring scheduler process to schedule a thread group. The scheduling request is tagged with a deadline, value, and release time. The Spring scheduler forms a task set consisting of all tasks in the existing schedule and all tasks in the new request, and attempts to schedule this set. The tasks in the new set are the task-level translation (performed at compile-time) of the threads in the thread group. If every task from its existing schedule ***and*** every task in the new request are schedulable, the new request is accepted for execution; otherwise, the request is rejected[1] and the APs continue executing the schedule that was present before the arrival of the request. By operating in this manner, the Spring system offers a two-part ***guarantee:***

- A new request will only be accepted if every task in the request can be scheduled.
- Once accepted, every task is guaranteed to be executed before its hard deadline, irrespective of future scheduling requests.

3 Language Constructs and Semantics

There are two steps for defining a thread. First, the description of the thread (and thread group) entities with respect to timing properties, layout requirements, etc., are specified in constructs of the System Description Language (SDL) [8]. Next, Spring-C [6] is used to code the application threads.

3.1 SDL

Threads and thread groups are defined in the System Description Language, SDL. SDL statements are used to record real-time and non-real-time properties for objects in the real-time system. This information is used by the Spring kernel and Spring scheduler to meet the requirements of the real-time system. An SDL statement declares the individual threads and precedence relationships of a thread group. For example, the declaration of thread group ***TG3*** from Figure 1 is:

```
SDL {
  Thread_group(TG3) {
    Thread_graph {
    Begin:  T3;
    T3:  T4;
    };
  };
};
```

This statement means that thread group ***TG3*** must begin with thread ***T3,*** and thread ***T3*** must be followed by thread ***T4.*** Other attributes for a thread group such as value can be declared if statically known; they do not have to be specified in SDL statements and can be specified in a scheduling request to the Spring scheduler.

There are three properties of a thread declared in an SDL statement: the process to which the thread belongs, the processors on which the thread can execute, and the type of guarantee required. The optional `Processors` attribute allows the user to restrict which Application Processors the thread can be scheduled on. This allows the user to meet the requirements of the real-time system design by effectively "reserving" processors for specific types of threads. The `Guarantee` attribute can be either ***Independent*** or ***Dependent.*** If a thread requires a Dependent guarantee, a run-time schedulability analysis for it must also include a schedulability analysis for all threads it can spawn—if at least one thread it spawns cannot be scheduled, it is rejected. An Independent guarantee requirement means that the child threads do not have to be considered immediately; schedulability analysis for a child thread occurs at the time of the execution of the code that spawns the child thread. This guarantee type can be used as an approach to implementing imprecise computations [9].

The key difference between the two types of guarantees is the task set that is initially scheduled. In the case of a thread that requires a Dependent guarantee, the task set to be scheduled ***includes*** the tasks of each child thread—the first task in the child thread cannot begin executing before the task in the parent that spawns the child. Precedence constraints are used to encode the start time of the first task of each child thread. The task set of a thread that requires an Independent guarantee only includes those tasks that are part of the parent thread. One of these tasks includes a statement that at run-time will directly engage the scheduler to schedule a child thread. In other words, in the case of a thread that requires a Dependent guarantee, all tasks of all child threads are present in the schedule at the time that the first task in the parent thread begins executing. This is not the case for a thread that requires an Independent guarantee.

As an example of the notion of required guarantee, consider thread ***T2*** from Figure 1. An important decision for the design engineer is whether or not thread ***T2*** constitutes "useful" work, even without the execution of thread ***T3*** and thread ***T4.*** If it does not, the scheduling of thread ***T2*** should be contingent on the scheduling of threads ***T3*** and ***T4.*** In this case, thread ***T3*** from Figure 1 would be declared as:

```
SDL {
  Thread(T3) {
    Process P1;
    Processors AP1;
    Guarantee Dependent;
  };
};
```

[1] Rather than simply rejecting the newly requested work, other options are possible, such as negotiating less service, performing distributed scheduling with other Spring nodes, or removing less important but previously guaranteed tasks in favor of the new, more important work. We will not consider these options here because they have no direct relevance to this paper.

A third type of SDL statement that is used for threads is the `Layout` statement, which includes the specification of the mapping from process to physical memory. An example layout of a Spring node named *rtl4* is:

```
Layout {
  Node_layout(rtl4) {
    Processor_layout(AP1) {
      Appl_set {
        Process_set P1;
      };
    };
  };
};
```

This statement places Process ***P1*** in the physical memory of Application Processor ***AP1.*** Note that the `process_set` attribute of the `layout` SDL statement and the `processors` attribute of the `thread` SDL statement are used for different design decisions.

3.2 Spring-C

This section discusses the thread language constructs for Spring-C. The convention adopted for the syntax is based on [10] and [11].

```
RESULT sync_thg_sched (thg, thg_params, rel_time,
           deadline, value, max_wait)
       any_t thg, thg_params;
       int rel_time, deadline, value, max_wait;
```

The `sync_thg_sched` procedure is used by a thread that requires an Independent guarantee to attempt to spawn a new thread and wait for a schedulability response. The end-to-end deadline for the thread group is `deadline`. The earliest time at which any task in the thread group can execute is `rel_time`. The value of the thread group is `value`. The maximum amount of time that the spawning thread will wait for a schedulability response is `max_wait`. Typically, code immediately after this statement in the spawning thread branches on the result of the schedulability response. This statement is predictable; the WCET is `max_wait`, which is assumed to include, in addition to the time for the scheduler to schedule the tasks of the thread, the time to actually request the scheduler to schedule thread group `thg`. The return value is `sched_yes`, or `sched_no`. Extensions are being considered that attempt to more fully explain ***why*** the thread group was not schedulable [7]. This is discussed in more detail in Section 5.3.

The `sync_thg_sched` procedure causes the direct interaction of a thread with the scheduler. The scheduler bases its actions on the guarantee requirement of each thread in the thread group that is attempting to be spawned. The structure that the schedule must schedule is generated at compile time via SDL statements and retained at run time, so there does not exist an overhead for dynamically determining what tasks need to be scheduled.

```
int async_thg_sched (thg, thg_params, rel_time,
           deadline, value)
       any_t thg, thg_params;
       int rel_time, deadline, value;
```

The `async_thg_sched` procedure is used by a thread that requires an Independent guarantee to attempt to spawn a new thread and then continue its processing, without waiting for schedulability analysis on the spawned thread. The return value is an identifier that can be used to later determine the status of the scheduling request. The WCET of this statement is predictable; it is based on the time to deliver a message to the SP processor, on which the Spring scheduler resides.

```
RESULT check_status_async_thg_sched (request_num)
       int request_num;
```

The `check_status_async_thg_sched` procedure is used to check the status of an asynchronous thread group scheduling request to the scheduler. The return value is `sched_yes`, `sched_no`, or `unknown`. The value `unknown` is the "intermediate" status of the request, returned between the time of the new scheduling request and the time at which the schedulability has been determined.

```
void thg_spawn (thg, thg_params, rel_time,
           deadline, value)
       any_t thg, thg_params;
       int rel_time, deadline, value;
```

The `thread_group_spawn` procedure is used in threads that require Dependent guarantees. The notion of synchronous or asynchronous thread spawning does not apply, because the scheduling of any child thread group has already been scheduled at the time that the parent thread was scheduled. The WCET of the statement is zero, because this statement does not result in executable code. Because this statement results in scheduling information being extracted at compile time, all parameters must be set at compile time, except for `tg_params`. Timing values are relative to the point at which the call is logically issued.

```
int thread_group_abort ()
int thread_abort ()
```

Both of these procedures allow a thread to prematurely terminate its activity. The procedure `thread_group_abort` aborts the entire thread group to which the thread belongs. Both procedures engage the scheduler to remove the remaining tasks on the current schedule. The semantic correctness of aborting remaining tasks is not guaranteed by the kernel; the user must evaluate when and if these calls can be made.

4 Implementation of Kernel Threads

This section describes the status of the creation of a threads package for the Spring kernel, which was previously a non-threaded architecture. We decided to implement sterile threads before attempting fertile threads. Fertile threads require the ability of an application to directly interact with the scheduler, which requires a more complicated implementation. This section describes the implementation of the sterile threads, which was in two phases. The first stage consisted of changing the kernel data structures to support a threaded model. This took place without altering the code for the context switch. The second stage was to alter the context switch to recognize that certain operations did not have to be performed if switching between threads in the same process.

4.1 Phase 1: Basic Changes

To implement threads, many components of the Spring real-time system design package had to be changed. The SDL compiler was made to parse and record the new thread-related SDL statements. While the Spring-C compiler was not altered, the linker was modified to create an executable file that has multiple entry points. The header of the executable file was augmented with information that maps symbolic thread names to entry points. Prior to threads, upon booting, the kernel loaded each process into physical memory as specified in the `layout` statement, and initialized the process with a process-specific procedure. After initializing the process, a single context was established for the process. For sterile threads, the kernel was changed to create a context for each thread after invoking the single process initialization procedure. Prior to threads, the scheduler was given ***process groups;*** scheduling consisted of scheduling the individual tasks of each process in the group such that the task interrelationships and task requirements were fulfilled. Dispatching involved switching to the context of the process of the next task to execute. With sterile threads, the scheduler is now given thread groups instead of process groups. The thread groups still break down into tasks, so scheduling is still on a task-by-task basis. Dispatching now involves another level of indirection; the new context is now obtained from a pointer from the task to the thread, and from the thread to the address space.

Because the basic unit of computation from the perspective of the kernel and scheduler remains the task, the predictability of the Spring kernel was retained when sterile threads were implemented.

4.2 Phase 2: Context Switch

Figure 3 illustrates the operations of the context switch in Spring prior to implementing sterile threads. The example shown is the context switch from a task in User Process 1 to a task in User Process 2. Two context switches actually occur; one from User Process 1 to the kernel process, and the second from the kernel process to User Process 2. The kernel process contains a call to the dispatcher, which determines the next user process to execute. The main difference between the two context changes is that the change from a user process to the kernel process (dispatcher) does not require loading a user map into the MMU.

Spring establishes predictable memory operations by two procedures [6]. First, page faults in the MMU are avoided by preallocating, at process creation time, a physical page for every used page in a program's address space, and loading that page in memory. Second, the contents of the translation look-aside buffer (TLB) are explicitly managed to ensure that all memory references experience TLB hits. When a context switch occurs, the mappings of all the used pages in the address space are loaded into the TLB. This approach is closely integrated with the Spring scheduler, which schedules non-preemptible segments of program execution (tasks). The non-preemptive aspect of Spring ensures that TLB contents need only change in a context switch and that the overhead of TLB maintenance will occur at predictable times with predictable cost. The ramification of this is that the context switch cost appears high. Table 1 gives the times for the various operations of context switch in the pre-threaded Spring, given the current processor in a Spring node, which is the 16.67MHz MC68020. This time is focussed in the context switch operation; in other systems it would be spread among ordinary memory references, and thus this cost would be accrued in the normal execution of application code.

The largest portion of the context switch involves loading the user map into the MMU. If the number of pages in the user address space is high, this cost can be large (the address space for the context switch in Table 1 contains four pages, which is the minimum number of pages required for a process). Conceptually, this cost can be eliminated if the next task selected is in the same address space as the last task executed.

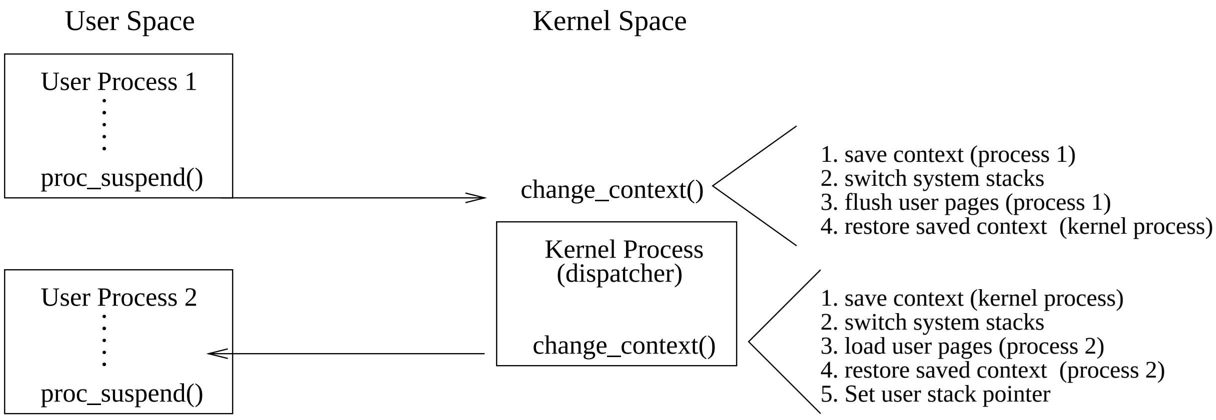

Figure 3: Context Switch in Spring, Pre-Threads

Component	Time (μsec)
Save Registers	15
Validate Page	21
Map Page	50
Flush Entry	10
Load Entry	10
Invalidate Page	17
Flush Map	8
Load Map	420
Restore Register	16
Miscellaneous	75
Total	626

Table 1: Process-Style Context Switch Times

Number of Pages	Percent Improvement
5	69%
10	73
15	76
20	79
25	81
30	83
35	84
40	85

Table 2: Improved Overall Context Switch Times

To achieve this reduction, the context switch from user process to kernel process was changed to no longer perform a flush of the user pages. This step was moved into the context switch that changes from the kernel process to the user process. This step is performed immediately before user pages are loaded. These two steps are now performed only if the address space of the new task is not the same as the address space of the previous task. The only drawback to this design is that user pages are still resident and considered valid when the kernel process is running. However, this is not a great concern, because the kernel process has no reason to access user space.

Implementing this change resulted in reducing a context change between tasks of the same process to 206 μsec. This represents a 66% improvement over the previous duration for a context switch for a process of minimum size; the improvements are greater for larger processes (see Table 2). Recognizing that the true value of this performance gain is dependent on the *frequency* that this improved context switch occurs, research is beginning on the cost-effectiveness of enabling the scheduler to exploit the faster context switch. The key will be in determining how much effort the scheduler must take in order to create schedules such that tasks of the same process are temporally adjacent.

5 Features of Spring Threads

This section discusses topics that relate to the semantics of the threads package: the synchronization of threads, the parallel execution of threads, and negotiation between an application thread and the scheduler.

5.1 Synchronization of Threads

The Spring scheduler is a reservation-based, dynamic scheduling system. An important property of a task is its resource requirements. The Spring scheduler schedules a task on a processor only if it has access to its required resources in its specified access modes (either *shared* or *exclusive*). Therefore, the constructs to lock objects that explicitly specify how long a thread is willing to wait for an object is not necessary in our model. For example, the real-time threads package of [11] contains a `RTLock` procedure:

```
RESULT
RTlock (rtlock, maxwait, time, func, arg)
LOCK rtlock;
int maxwait, time, (*func)();
any_t arg;
```

In this procedure call, `maxwait` is the maximum time the issuing thread can wait to acquire the lock. The difference here is the semantics of the call. In the procedure call `RTlock()`, an application thread can wait for `maxwait` time, without ever acquiring the lock. While the timing requirements of the thread issuing the statement are not compromised (`maxwait` is included in the WCET of the thread issuing the call), the integrity of the thread might be. The correctness of the thread execution in terms of functionality is suspect, because it never received access to a key resource. While the ramifications of this occurring may be capable of being analyzed on a per-case basis, assessing the effect of multiple occurrences of threads not being granted access to key resources may be quite difficult. The integrity of the overall system can be compromised in this situation.

The Spring model of threads do not have this shortcoming, because the scheduling procedure eliminates the nondeterministic wait associated with gaining access to resources. Also, because scheduling is reservation-based, the problem of priority inversion does not have to be addressed, as is necessary in priority-based scheduling approaches to real-time threads [2, 3].

5.2 Parallel Execution of Threads

The context of a thread resides in physical memory on the MVME136A board of the processor executing the thread. If a thread executes on the processor of a MVME136A board that contains the address space of the process to which the thread belongs, access to data in the address space does not require the VME bus. Therefore, memory accesses are faster, and another processor is not victimized. In order to execute a thread on a different processor, the code segment of the process is loaded onto that AP at boot time. By doing so, instruction fetching does not require access to the VME. However, all accesses to the data space require the VME bus. Note that although these accesses are still predictable, they require more time. The scheduler recognizes the difference in memory access time and attempts to build a schedule such that processor utilization is maximized and the cost of memory references is minimized. When a request arrives, the scheduler maps threads to processors in order to fairly distribute the workload, taking into account the desire to keep memory accesses local.

5.3 Negotiation between Application Thread and Scheduler

The real-time system designer establishes a set of actions that the system can take to perform its functionality. There can exist multiple actions for the same event, differing along the dimensions of precision, accuracy, and completeness. For example, one action can be exhaustive in its operations, requiring many resources and a significant amount of time. Another action can be a response to the same event, except that its steps are cruder, requiring much less time. The action best chosen as a response to an external event often can depend on the exact situation in which the event occurs. Designing multiple responses creates a more robust system.

Because the Spring kernel retains a significant amount of information at run-time (Spring is a ***reflective*** kernel [12]), the scheduler process has the ability to provide information in addition to `sched_yes` or `sched_no` in response to scheduling requests. This allows for a thread spawning another thread to ***negotiate*** with the scheduler over the action taken in response to an event in the environment, enabling the dynamic determination of the most appropriate response to an event in the environment.

For example, assume that there are two different actions that the system can take in response to a particular event: the first action is codified by thread group ***TGx,*** and the second is codified by thread group ***TGy.*** Assume that ***TGx*** requires more time and more resources than ***TGy.*** A typical negotiation between an application thread and the scheduler might be to first request the scheduler to attempt to schedule ***TGx*** with a tight deadline. If that fails, the application thread might request the scheduling of ***TGx*** again, except with a longer deadline. If that fails, the application thread can attempt to schedule ***TGy,*** which can be viewed as the ***fall-back action.*** In a typical negotiation, the scheduler can provide information that can be used to direct the next decision of the application thread, such as the bottleneck resource that prevented the scheduling of its last request. To perform this negotiation, the WCET execution time of the negotiation must be accounted for in the scheduling of the thread that attempts to spawn other threads. Higher-level Spring-C constructs are being developed to provide this interface to the user.

It should be acknowledged that, while this expands the freedom of application designers, this exacerbates the performance analysis concerning the ability of the system to meet its requirements.

6 Related Work

Three real-time thread packages are presented and compared to the Spring real-time threads package. The comparison is primarily on the basis of predictability and flexibility.

6.1 PThreads

POSIX 1003.4a (PThreads) [2] supplements the base POSIX standards with interfaces and functionality to support multiple flows of control within a process. The PThreads document is purely a standard and any compliant implementation of PThreads is not constrained in any way beyond providing the appropriate interfaces. Issues related to implementation are not part of the standard. For instance, some issues the standard does not address are: dynamic versus static thread creation, user-level versus kernel level threads, and the interface between scheduling at the thread level and at the kernel/processor level.

The basic PThreads thread model is priority driven and preemptive, with signal handling, mutual exclusion and synchronized waiting. A thread has attributes that can be configured at creation time, including scheduling policy, scheduling priority, scheduling scope, scheduling parameters, and stack size. Because these attributes do not fulfill many real-time application needs, an implementation may define further attributes such as deadline, start time, worst case execution time, periodic/aperiodic, critical/essential/non-real-time. Along with these attributes, there must be a set of defined semantics for how these attributes affect thread behavior.

PThreads defines only a scheduling interface and no policies. PThreads considers two scheduling models; a system model, where all threads are scheduled against all other threads in the system, and a process model where threads are scheduled against other threads within the same process. The latter requires defining an interface between scheduling at the threads level and scheduling at the kernel/processor level.

POSIX.4 [1] modifies PThreads for real-time computing. It specifies a priority scheduling mechanism with FIFO, RR and OTHER (implementation-defined) policies. The PThreads interface is consistent with this and has an option to indicate that these scheduling policies are implemented. In that case, priority is then defined as a thread attribute; system calls exist for setting and getting a thread's priority.

PThreads provides mutexes and condition variables as the primary means for synchronization. The PThreads interface does not explicitly provide a way to bound blocking on a mutex, which can obviously sacrifice predictability. Only one call, pthread_mutex_trylock(), which does not block if mutex is already locked, provides deterministic behavior for mutex locking. Bounded waiting on condition variables is possible through pthread_cond_timedwait(). However, all condition variables are associated with a mutex which must be locked by the waiting thread before performing a wait on a condition variable. Hence, to provide bounded waiting, PThreads anticipates (but does not require) the implementation of the priority inheritance protocol and priority ceiling protocol.

PThreads defines extensive interfaces for signal handling and thread cancelation. The PThread model provides per-thread signal masks, per-process signal vectors and single delivery of each signal either to a specific thread or to a process (within which the choice of receiving thread is implementation defined). Thread cancelation is well-defined with specified cancelation points and optional cancelation cleanup handlers. Finally, PThreads also provide for thread-specific data, to allow threads to have thread-specific global variables.

It should be noted that PThreads is a real-time standard only in that it is consistent with POSIX.4. Remove the considerations for co-existence with POSIX.4 and non-real-time threads remain. In contrast to the real-time threads package presented in this research, PThreads does not incorporate schedulability analysis, there is no notion of guarantee, and it is not clear how the normal timesharing parts of POSIX will impact the real-time performance.

6.2 Real-Time Mach Threads

To develop a real-time program that uses threads in Real-Time Mach [3], the user is expected to perform schedulability analysis off-line. To perform a schedulability analysis, the user analyzes the set of ***tasks*** (each of which is a separate address space) and the ***threads*** according to a scheduling policy. A task in the terminology of Real-Time Mach is analogous to the process discussed in this work. Real-Time Mach supports five scheduling policies: Rate Monotonic, Fixed Priority, Round Robin, Rate Monotonic with Deferrable Server, and Rate Monotonic with Sporadic Server. Synchronization is through mutexes and conditions (signals). Preemption of a thread within a critical region is on a lock-by-lock basis: (1) no preemption, (2) preemptible, or (3) restartable. The priority ceiling protocol can be used to control priority inversion.

Threads are kernel-level objects. The kernel maintains separate dispatch queues for real-time threads and non-real-time threads. EDF is used as the queuing policy for both RT-mutex and RT-condition primitives. The kernel provides no run-time schedulability support.

The Real-Time Mach approach to threads does not allow for truly dynamic creation of threads—the application program is not allowed to attempt to create a thread under the condition that the operating system provide immediate feedback concerning its schedulability. This is not a direct result of threads, but rather of the priority-based approach to scheduling adopted by Real-Time Mach. The language constructs are similar to those adopted by POSIX, which is also founded on priority-based scheduling.

The kernel-level threads of Real-Time Mach have recently been the basis for user-level threads [13]. User-level threads offer two benefits over kernel-level threads: the ability to switch contexts without making a system call and the ability to support a variety of scheduling models. A main goal of user-level real-time threads in Real-Time Mach is the quick, dynamic management of thread attributes. For instance, a ***deadline handler*** is attached to every real-time thread; when a thread misses its deadline, the deadline handler is invoked, which can adjust the timing attributes in user space to reduce the system load. While this approach to dynamic environments is appropriate for many real-time environments such as multimedia applications, it may not be suited for hard deadlines in dynamic environments, because a thread may begin executing and then miss its deadline.

User-level threads were considered as the basis of real-time threads in Spring but were not pursued, primarily because the notion of ***guarantee*** as currently supported by Spring, and, we believe, required by hard real-time systems, would no longer be applicable if threads were implemented at the user level. In Spring, a thread is said to be guaranteed if every task in the thread has been scheduled such that it has access to its required resources for a time equal to its WCET. Scheduling requires a centralized view in order to perform resource allocation. If threads were implemented in user space, this type of resource allocation would be difficult if not impossible. If an application could arbitrarily decide to change from one thread to another, resources would be requested and released in non-determinate manner.

6.3 Schwan's Real-Time Threads

The real-time threads package of Schwan [11] is based on the Mach C Threads interface [10]. The approach taken is that the schedulability of a thread should be guaranteed before it is actually run. This guarantee can be made at either program compilation time or at the time of thread creation. The approach here is that "higher-level operating system software" reacts according to the inability of the operating system to guarantee a thread (or a group of threads). The philosophy here is most closely related to Spring–both agree that deadline-based scheduling is more appropriate for dynamic environments than priority-based scheduling.

The description of threads can be categorized as manipulation, control, and higher-level utilities. For manipulation, there are calls to spawn threads of different types: sporadic, periodic, and event-type. Performance is improved by separating the schedulability test from the scheduling itself. Non-real-time threads are supported, and are treated in the same manner as Real-Time Mach does (separate scheduling queues). For control, there is the option of sleeping either for a maximum amount of time or until an event occurs. One thread can explicitly wake up another thread. Synchronization is with mutex locks. Like Real-Time Mach, access to critical regions is EDF. The RTthread_prefork() construct allows users to specify at the time of creation of a thread what additional threads will be created within the thread. Schedulability analysis of the thread takes this into account.

Each node, which is a uniprocessor, keeps (1) a pool of stacks for threads, (2) pools of thread descriptor and timing information blocks, (3) a local ready list sorted by deadline, (4) other structures for scheduling, and (5) a memory pool for future allocation/deallocation requests. The Timing Information Block (TIB) is resident in the forking processor, and the Thread Control Block (TCB) is resident in the forked processor.

Our approach to the design and implementation of real-time threads for a dynamic environment differs from this work in many ways. The first principle difference is that our work concerns scheduling for a single multiprocessor, while Schwan's work deals with a network of uniprocessors. The scheduling system in our design is able to opportunistically schedule threads with the intent of maximizing the utilization the processors within the single node. This includes the ability to make scheduling decisions using the recognition that the cost of access to memory is a direct consequence on the processor that is selected. This is a significantly different model than this work, which takes a distributed approach—one node in the network must decide which node in the network on which to schedule a spawned thread. If the node that is intended to run the new thread is overloaded, the thread can not be spawned. The second principle difference is the semantics of a threaded application as a function of kernel support. In our work, a thread is scheduled based on its WCET but is guaranteed access to required resources ***before*** the thread executes. The approach of [11] is to also schedule a task based on its WCET, but only take into account the amount of time required resources will be used, without directly scheduling access to those resources. If, at run time, another thread is using the resource, a thread requesting the resource will be denied access. Even though the thread will still complete before its deadline, it will not necessarily perform the desired action. This limitation is present in each of the three real-time thread packages discussed in this section; the real-time threads package for Spring does not have this limitation because it uses a reservation-based scheduling system.

7 Conclusion

A real-time threads package must be predictable and flexible in order to operate in dynamic environments. This paper has described the design and partial implementation of real-time threads for dynamic environments. By basing the threads package on an integrated design and run-time system that includes an off-line specification language, a real-time programming language, kernel boot time options, run-time kernel and a guarantee-based, on-line dynamic scheduler, predictable performance can be achieved. A range of options exist in the real-time threads package to support flexibility.

This paper has also described the implementation of sterile threads and presented results from that implementation. Sterile threads are being used as the basis for experiments conducted in the Flexible Manufacturing Testbed [14] in the Center for Autonomous, Real-Time Systems (CARTS) at the University of Massachusetts. The testbed uses the Spring kernel to achieve coordinated control of multiple workcells. The implementation of fertile threads is continuing.

References

[1] IEEE, ***Realtime Extension for Portable Operating Systems***, Feb. 1989, P1003.4/Draft6.

[2] IEEE, ***Threads Extensions for Portable Operating Systems***, Apr. 1993, P1003.4a/Draft7.

[3] H. Tokuda, T. Nakajima, and P. Rao, "Real-time Mach: Towards predictable real-time systems," in ***Proceedings of the USENIX 1990 Mach Workshop***, Oct. 1990.

[4] R. Rashid, R. Baron, A. Forin, D. Golub, M. Jones, D. Julin, D. Orr, and R. Sanzi, "Mach: A foundation for open systems," in ***Proceedings of the Second Workshop on Workstation Operating Systems (WWOS2)***, Sept. 1989.

[5] J.A. Stankovic and K. Ramamritham, "The Spring kernel: A new paradigm for real-time systems," ***IEEE Software***, vol. 8, no. 3, pp. 62–72, May 1991.

[6] D. Niehaus, ***Program Representation and Execution in Real-Time Multiprocessor Systems***, PhD thesis, University of Massachusetts, Amherst MA, 1994.

[7] M. Humphrey, "Using a reflective real-time operating system to implement a just-in-time scheduling policy for a flexible manufacturing workcell," Working notes., 1995.

[8] D. Niehaus, J. Stankovic, and K. Ramaritham, "A real-time system description language," in ***Proceedings of the 1995 IEEE Real-Time Technology and Applications Symposium***, May 1995.

[9] J. W. S. Liu, K. J. Lin, W. K. Shih, A. C. Yu, J. Y. Chung, and W. Zhao, "Algorithms for scheduling imprecise computations," ***IEEE Computer***, vol. 24, no. 5, pp. 58–68, May 1991.

[10] E.C. Cooper and R.P. Draves, "C threads," Tech. Rep. CMU- CS-88-154, School of Computer Science, Carnegie Mellon University, Pittsburg, PA, Feb. 1990.

[11] K. Schwan, H. Zhou, and A. Gheith, "Multiprocessor real-time threads," ***Operating Systems Review***, vol. 26, no. 1, pp. 54–65, Jan. 1992.

[12] J.A. Stankovic and K. Ramamritham, "A reflective architecture for real-time operating systems," in ***Advances in Real-Time Systems***, Sang Son, Ed. Prentice-Hall, 1994.

[13] S. Oikawa and H. Tokuda, "User-level real-time threads," in ***Proceedings of the 11th IEEE Workshop on Real-Time Operating Systems and Software***, Seattle, WA, May 1994.

[14] J.A. Stankovic, G. Zlokapa, and K. Ramamritham, "Real-time platforms and environments for time constrained flexible manufacturing," in ***IEEE Workshop on Real-Time Operating Systems and Software***, May 1994.