Predictable Threads for Dynamic, Hard Real-Time Environments*

Marty Humphrey and John A. Stankovic†

June 25, 1998

Abstract

Next-generation hard real-time systems will require new, flexible functionality and guaranteed, predictable performance. This paper describes the UMass Spring threads package, designed specifically for multiprocessing in dynamic, hard real-time environments. This package is unique because of its support for new thread semantics for real-time processing. Predictable creation and execution of threads is achieved because of an underlying predictable kernel, the UMass Spring kernel. Design decisions and lessons learned while implementing the threads package are presented. Measurements affirm the predictability of this implementation on a representative multiprocessor platform. The adoption of the threads package in the UMass Spring kernel results in additional performance improvements, which include reduced context switching overhead and reduced average-case memory access durations.

Keywords— Real-Time Operating Systems, Threads, Multiprocessing

1 Introduction

For many applications, real-time threads executed by a fixed-priority policy offers the necessary predictability and flexibility. However, in contrast to the small, *static* environment for which this threads model is particularly well-suited, next-generation hard real-time systems will be large, complex, distributed, and adaptive [1]. The conventional approach for designing small, embedded systems is to, in effect, perform all schedulability analysis off-line and thus preallocate resources to activities with fixed attributes such as priority and execution rate. This approach is impractical for next-generation hard real-time systems, because of the computing resources required to support the multitude of possible scenarios. For example, an aircraft reconnaissance mission might encounter a large number of possible situations, such as detection by hostile entities, component

*This work was funded by the National Science Foundation under grant IRI-9208920. This work was performed while both authors were at the University of Massachusetts at Amherst.

†Department of Computer Science, University of Virginia, Charlottesville, Virginia, 22903.

subsystem failure, weather-related faults, and dynamic mission reconfiguration. Preallocating resources in this situation is too costly because of the hardware capacity necessary to handle every possible situation. Instead of preallocating resources, the operating system must support the specification and execution of sets of computations with end-to-end constraints, dynamically and unexpectedly submitted for execution.

UMass Spring threads have been designed specifically for dynamic, hard real-time environments. Unique to this package is the ability to specify relationships between threads and define resource requirements of threads such that these requirements must be dynamically guaranteed *before* a thread begins executing. For example, the use of a particular construct of the threads package in the coding of a thread enables a user to instruct the operating system that the *spawning* thread must execute if and only if the *spawned* thread can also execute, all subject to timing requirements. This threads package also enables a user to write and execute applications that dynamically react to the inability of the operating system to guarantee the spawning of a thread; for example, a thread can make multiple attempts to spawn a thread of lessening resource requirements until the point can be reached that the operating system can guaranteed the spawned thread's execution. In this situation, the threads package enables both the *spawning* and *spawned* thread to be executed under hard real-time constraints.

The most important property of UMass Spring threads is its run-time predictability. This predictability is ensured through the support and scheduling model of the UMass Spring kernel [2]. The UMass Spring kernel uses a dynamic, planning-based approach to resource usage, thus avoiding the blocking on resources that occurs in systems that are priority-based.

This paper describes the design of the UMass Spring threads package and its implementation and measurement on a representative platform. The real-time threads package builds upon previous work in the design and implementation of the UMass Spring kernel [2]. This paper extends the presentation of the high-level design and initial implementation contained in [3]. This completed implementation is, to our knowledge, the first threads package with such rich semantics that is suitable for hard real-time environments. The UMass Spring threads package offers new flexibility, without sacrificing predictability, as compared to the real-time thread packages contained in POSIX, Solaris, Real-Time Mach, and CHAOS-arc, as discussed in Section 6.

The rest of this paper is organized as follows. Section 2 provides the background and terminology of the UMass Spring kernel. This section establishes the context for the discussion of the design and implementation of the UMass Spring threads package. Section 3 presents the design of the UMass Spring threads package, emphasizing the unique semantics that are supported. Section 4 discusses the implementation of the threads package in the operating system, independent of a target platform. Section 5 presents the results of measuring the predictability and general performance of the thread constructs on a sample target platform. Section 6 discusses related work. Section 7 contains the concluding remarks.

2 Spring Architecture and System Components

This section describes the relevant aspects of the Spring real-time operating system *before* the design and implementation of the threads package. The UMass Spring threads package both builds upon the UMass Spring design and continues the overall development of the hard real-time kernel.

2.1 Hardware

Figure 1 shows the hardware architecture for the UMass Spring system, which is called SpringNet. SpringNet is a physically distributed system consisting of a network of multi-

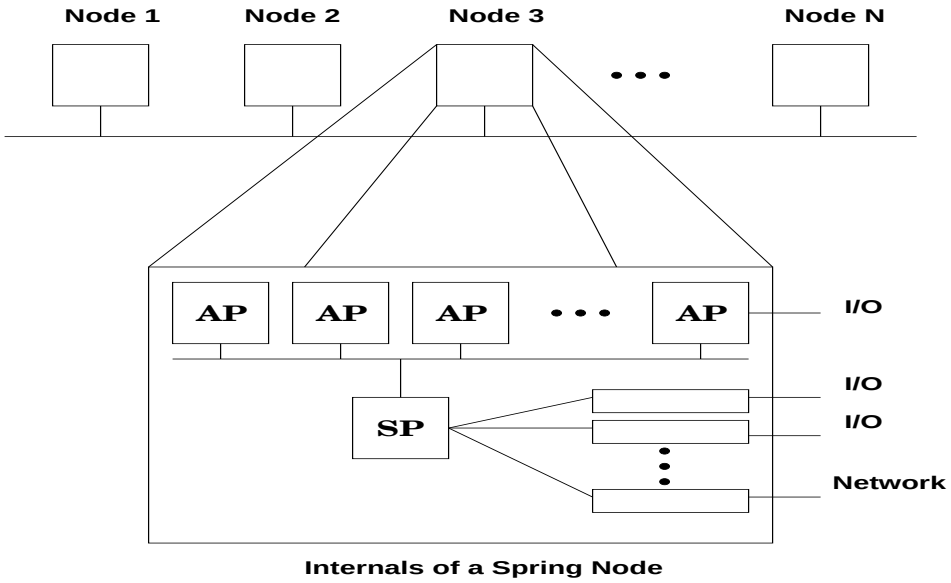


Figure 1: UMass Spring hardware architecture

processors each running the UMass Spring kernel. The thread package does not directly address the distributed capabilities of UMass Spring, so SpringNet will not be discussed any further. Interested readers should consult [4, 2].

Figure 1 also shows the expansion of a single UMass Spring node, which is a multiprocessor. One processor is the System Processor (*SP*), and is devoted to system activities such as scheduling and interacting with outside entities. The remaining processors are Application Processors (*APs*), which execute application code according to timing and functional constraints. Physical memory resident on each processor board is capable of being referenced directly by any processor in the node. A bus local to the processor board enables fast local reference, while the backplane bus must be used by a processor to access another board's physical memory, requiring a longer duration. The architecture of the Spring node is representative of such commercial systems as the MAXION multiprocessor by Concurrent Computer Corporation and certain multiprocessors by Silicon Graphics.

2.2 Software

The UMass Spring system consists of a real-time operating system, and two languages (SDL and Spring-C) and compilers used to specify user programs. Before discussing the operating system, this section presents an overview of how the user encodes applications and instructs the UMass Spring operating system of the applications' timing requirements.

To encode applications, the application developer uses Spring-C [5], which is a version of ANSI C that has been modified for use in a real-time environment. The modifications enable the compile-time timing analysis of code segments by removing the ability of the user to perform operations for which the duration cannot be predicted, such as unbounded loops. The Spring-C compiler [5] both produces object code and breaks the computation at potential blocking points into a series of precedence-related *tasks*. A blocking point occurs when a user program attempts to acquire a resource—in a conventional system, if the resource has previously been acquired by another thread, any thread that attempts to acquire the resource must block until the resource becomes available. As will be discussed, the UMass Spring system avoids blocking by explicitly planning resource use.

A task is an indivisible unit defined by its resource requirements, worst-case execution time (WCET), and precedence constraints with other tasks. The WCET of a task is determined by the Spring-C compiler by analyzing the machine-level instructions, along with a table of WCETs for the individual instructions for the target CPU. By breaking a program into well-defined, non-blocking tasks, both off-line and on-line analysis is possible where that analysis verifies that the deadlines are met[1].

Prior to the implementation of the threads package described in this paper, a process consisted of a single flow of execution. Therefore, prior to the threads package, the compilation of a process resulted in a single *task group*, which is a set of tasks related by precedence constraints. It is important to recognize that the task is finer-grained abstraction than a process (or a thread under the threaded model of computation); that is, a process could be composed of many tasks, but a task *could not* be composed of many processes.

An example of the translation of a process to its corresponding task group is shown in Figure 2, which is a robotics process used in the Spring system to move a linear table of a flexible manufacturing workcell [6]. The left side of the figure shows the shell of the code, and the right side shows the task group that results from the compilation of the code. In this process, the resource corresponding to the linear table is called *linear_table*. The use of *linear_table* in *exclusive* mode, through the **REQUEST** and **RELEASE** constructs, ensures that only one process can use the linear table at a given time. The functionality of the process is to perform some computations (*Task 1*), move the linear table (*Task 2*), and then execute some error correcting code (*Task 3*). The arrows in the right side of Figure 2 indicate precedence constraints between tasks.

While the functionality of an application program is expressed in Spring-C, the System Description Language (SDL) [7] is used to express timing and resource requirements

[1] In scientific computation compilers are often utilized to identify parallelism and the expense of special purpose compilers is justified. Similarly, for hard real-time systems sophisticated compilers are utilized to enable predictability and the analysis of predictability. This cost is also justified.

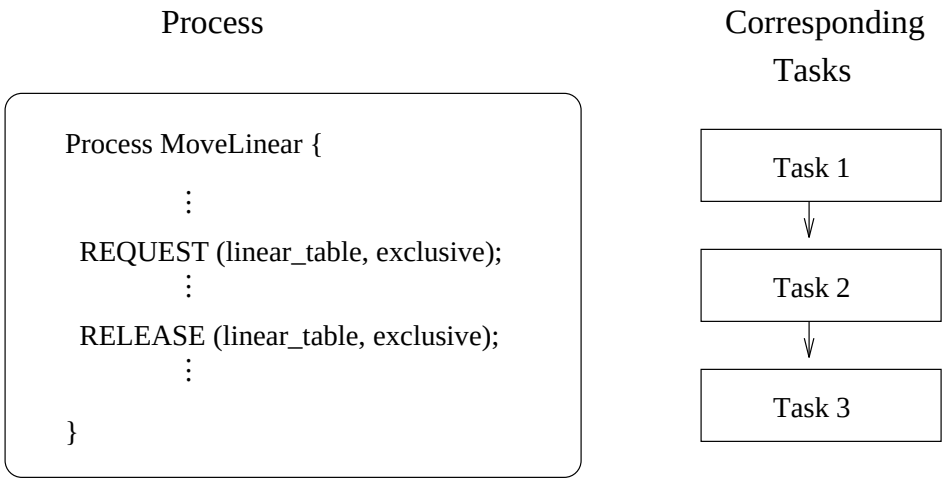


Figure 2: Mapping a process to tasks

of an application program, irrespective of the logical functionality. This information is used by the UMass Spring kernel and UMass Spring scheduler to meet the requirements of the real-time system. SDL is generated by hand (or by using an design tool) in certain situations, such as when the user instructs the Spring kernel how to allocate certain physical memory at boot-time. The second way SDL is generated is through the invocation of the Spring-C compiler. For example, while the Spring-C compiler generates the assembler code necessary to get the MoveLinear process of Figure 2 to *execute* in three separate tasks, the Spring-C compiler also automatically generates the SDL information that describes the *scheduling* information for the task group in the right of Figure 2. This SDL information describes the number of tasks, the precedence between tasks, the resource requirements of each task, and the WCET of each task.

Prior to the threads package, in order to execute a process on an AP, at the time that the kernel is booted, the process, which consists of global variables, read-only data, and code, is loaded into the physical memory of a particular AP as directed in the user SDL statements. Virtual memory is not supported in the UMass Spring kernel, because of its inherent unpredictability [5]. Execution is limited to the processors on which the process was physically loaded.

The ability of the user to specify groups of activities, with a single end-to-end deadline, was provided by the *process group* SDL construct[2]. In order to support the process group abstraction, at boot-time, the kernel translated each process group into a composite task group—consisting of the task group for each process, with precedence constraints added to retain the precedence between processes—that was used by the scheduler at run-time. The threads package replaces the process group with the thread group, which is more flexible and supports a more robust set of semantics (discussed in Section 3).

The UMass Spring scheduler is a user-level process responsible for the scheduling of user applications on the APs. It is a reservation-based, dynamic scheduling system—the scheduler explicitly plans the use of resources so that no task blocks for resource access

[2]The use of "process group" should not be confused with its use in the context of group communication.

(the resource requirements of each task is determined at compile-time by the Spring-C compiler). Prior to the threads package, the execution of user code on the APs began with a request from an "outside entity" to the UMass Spring scheduler process to schedule a process group. A scheduling request contains a deadline, importance, and release time—the earliest time at which any task in the scheduling request can execute. The importance of a scheduling request signifies the value imparted to the system when the corresponding process group is scheduled and executes according to its timing constraints. In conditions of overload, tasks can be shed or deferred according to their importance levels.

Scheduling on the SP occurs in parallel with the dispatching and execution of tasks on the APs. When a new scheduling request arrives, to achieve concurrent execution of the scheduler and the multiple dispatchers, a set of tasks is reserved for each dispatcher, where the scheduler is not free to reschedule the tasks reserved for the dispatchers [2]. The mechanism for determining which tasks cannot be rescheduled involves a *cutoff line*. Once the scheduler has determined an upper bound of its cost for scheduling, the scheduler adds this cost to the current time to determine the cutoff line. All tasks having a scheduled start time before the cutoff line are reserved for the dispatchers and thus cannot be rescheduled. Thus, each dispatcher has tasks to execute while the scheduler is attempting to reschedule the remaining tasks to guarantee the tasks of the new scheduling request. If every task from the scheduler's existing schedule *and* every task in the new request are schedulable, the new request is accepted for execution; otherwise, the request is rejected, and the APs continue executing the schedule that was present before the arrival of the request[3]. By operating in this manner, the UMass Spring scheduler offers admission control via 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 to meet a hard deadline, irrespective of future scheduling requests at the same or lower levels of importance.

The details of the Spring scheduler algorithms can be found in [8, 9, 10].

An example of a schedule produced by the Spring scheduler for three APs is shown in Figure 3. The schedule shown is for time 0 through time 15 milliseconds. Each task is labeled with its name and resource requirements. There are two resources, $R1$ and $R2$. The time at which the task is scheduled to execute is shown next to each task. By carefully planning resource usage, tasks will execute before their deadlines without having to spin or block until the needed resource becomes available.

3 Design of the Thread Package

The motivation for UMass Spring threads is to provide a threads package that is suitable for dynamic, hard real-time environments. This means that when new work arrives,

[3]Note that rather than simply rejecting the newly requested work, other options are possible, such as negotiating less service, performing distributed scheduling with other UMass Spring nodes, or removing less important but previously guaranteed tasks in favor of the new, more important work.

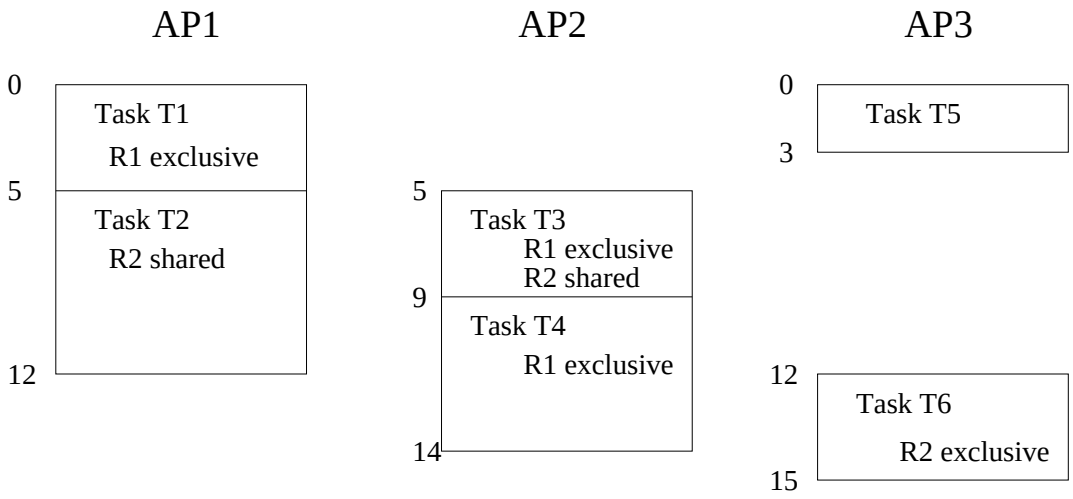


Figure 3: Representative schedule produced by the Spring scheduler

the system performs an admission control analysis with careful and accurate timing assessments based on WCETs on non-blocking components, computed by our compiler off-line. This section presents the design of the threads package, in the context of the threads package API, with an emphasis of new semantics that support flexibility (allows dynamic invocation of groups of tasks) while attaining predictability (via accurate timing assessments at admission control time). Key issues addressed in the design of the threads package include the predictability of synchronization between the scheduler and application programs spawning new threads (the scheduler and application programs execute on different processors in the Spring node), and the challenges of prematurely terminating a group of threads under hard real-time constraints.

3.1 Terminology

Terminology is introduced that will be used throughout the remainder of the paper. A *process* consists of an address space and one or more flows of execution through the address space (*threads*). If, at run-time, a thread spawns 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 parent thread may also spawn a *thread group*, which is a set of threads that are related by precedence constraints. Each thread group has an end-to-end deadline, a release time, resource requirements, precedence constraints, and value, which are either defined statically or dynamically. Thread groups are specified with SDL statements. The compilation of user programs into a series of tasks related by precedence did not change with the introduction of threads. Tasks remain the scheduled unit of execution. The planning-based approach of the Spring scheduler is fundamental to achieving predictability in the implementation of UMass Spring threads in the UMass Spring kernel.

Figure 4 illustrates the static, compile-time structure of an example process, *Process P1*. Assume that there have been 4 thread groups defined (via SDL statements, not shown): *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 4, at run time, the execution of thread *T1* spawns thread group *TG2*. The execution of thread *T2* spawns thread group *TG3*. The constructs and semantics of the spawning of a thread group are discussed next.

3.2 New Semantics

A key contribution of UMass Spring threads is the support for new semantics. In hard real-time environments, it is crucial that the *parent* thread be guaranteed to complete execution before its explicit deadline. The new semantics in UMass Spring threads centers around the guarantee of the *child:*

Dependent If the parent-child combination is viewed as as a single logical entity, in which both the parent and child must execute in order for it to be considered "useful", then the child thread group has a *Dependent* guarantee.

Independent If the situation is such that the parent must execute, and is it desirable but not mandatory that the child thread group execute, then the child thread group requires an *Independent* guarantee.

The determination of the type of guarantee required is made by the application designer. Each semantic type requires different support from the operating system in order to achieve predictability. The UMass Spring thread package directly supports each semantic type through different OS constructs.

The recognition and direct support for each of these types of semantic relations between the parent thread and the spawned thread group is an important contribution to hard real-time programming, because it enables the programmer to use only the resources that are needed for each situation. Previously, a real-time programmer had either of two options. In the first option, the user could pre-allocate resources based on a worst-case scenario, which is analogous to recognizing only the Dependent type of semantics. The problem with this is that there may not exist enough bandwidth on every resource to guarantee the worst-case scenario. The second option consists of dynamically attempting to acquire resources. This is analogous to only recognizing the Independent type of

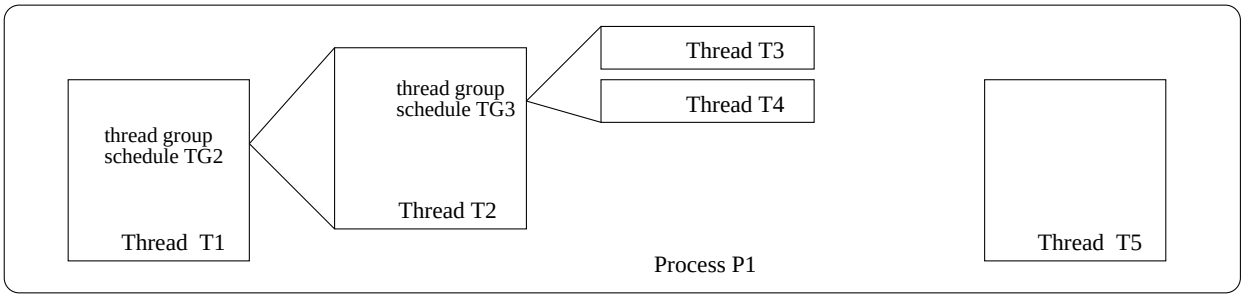


Figure 4: Static definition of a sample process

semantics. The problem is that a complex set of computations may not meet its single, overall deadline if it attempts to acquire resources on-the-fly. The support for both types of semantics facilitates semantic correctness, as well as higher resource utilization.

For a child thread group that requires an Independent guarantee, schedulability analysis should occur at the time the child thread group is spawned. However, the time required to spawn the child thread group must be taken into account in the execution of the parent thread. That is, the WCET of the parent thread must include the time to interact with the scheduler, which includes the time to submit the child thread group to the scheduler, the time for the scheduler to perform schedulability analysis, and the time to receive the schedulability response from the scheduler. If this time is not directly accounted for in the WCET of the parent, the parent may not be able to spawn the child thread group, which could sacrifice system integrity. A challenge in the design of the threads package is how to provide the parent thread with options for spawning a child thread group that requires an Independent guarantee.

A thread group that requires a Dependent guarantee requires different support. The schedulability analysis for a child thread group that requires a Dependent guarantee must occur at the time that the parent is introduced. In this case, the timing requirements of the parent must be augmented by the timing requirements of the child thread group. This approach is significantly different than the conventional real-time threads package, in which a thread is spawned irrespective of threads that it may or may not spawn in the future. Using this conventional approach results in unpredictability in these other thread packages.

The UMass threads package provides flexible mechanisms for structuring arbitrary hard real-time computation by allowing the programmer to mix the use of Independent guarantees with Dependent guarantees. If the programmer uses only Independent guarantees, the style of hard real-time computation is similar to the model of imprecise computation [11] in which there is a mandatory and optional part of computation. The imprecise computation model has been proposed to handle transient overload and to enhance fault-tolerance properties of real-time applications. Although there has been significant results toward general scheduling theories for many models of imprecise computation, there has been only limited research into platforms for implementing imprecise computation [12]. The UMass Spring threads complements the development of general theories of imprecise computation by providing the necessary constructs to implement a hard real-time application according to the imprecise computation paradigm.

The UMass Spring threads package is unique in its support for prematurely terminating a group of threads if some event occurs, either in the computing environment or in the external environment. A challenge in the implementation of premature thread group termination operations is to predictably manage the latency inherent in the multiprocessor architecture. This is addressed in Section 3.3.6, Section 4.4, and Section 5.2.

3.3 Constructs

The convention adopted for the syntax of the constructs in the UMass Spring thread package is based on [13] and [14]. The focus of the presentation is how each construct facilitates the predictable execution of hard real-time computation.

3.3.1 Creation of Processes and Threads

RESULT create_process (name, executable)
char * name;
char * executable;
The creation of a process requires the executable file to be downloaded from the file system. Because file systems inevitably incur unpredictable access times, this primitive should be executed at initialization time or in the absence of hard real-time constraints.
RESULT create_thread (process_name, thread_name)
char *process_name;
char *thread_name;
The **create_thread** construct is used to instruct the kernel to allocate kernel data structures (a Thread Control Block, or TCB) for the specified thread. Separate constructs are used to initiate execution.

3.3.2 Thread Synchronization

void REQUEST (resource, mode)
char *resource;
char *mode;
void RELEASE (resource)
char *resource;
The **REQUEST** construct is used to request access to a resource. The mode of the resource access can be either *shared* or *exclusive; exclusive* is used to achieve mutual exclusion.

Conventional threads packages include additional constructs for thread synchronization, such as condition variables and signals. These features create another degree of unpredictability in these packages. In the UMass Spring threads package, synchronization and signaling are handled via planning, which avoids this source of unpredictability.

3.3.3 Synchronous, Independent Thread Group Spawning

RESULT sync_thg_sched (thg, thg_params, rel_time, deadline, importance, max_wait)
any_t thg, thg_params;
int rel_time, deadline, importance, max_wait;
The **sync_thg_sched** construct is used to spawn a thread group that requires an Independent guarantee. "Spawn" refers to the scheduling and execution of a thread; the allocation of the TCB is necessarily performed using the **create_thread** construct. The **sync_thg_sched** construct causes the direct, synchronous interaction of a thread executing on an AP with the scheduler. 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 importance of the thread group is **importance**. In principle, the use of this construct can result in a previously-accepted thread group to be removed from the schedule, if the previously-accepted thread group is of lower importance. While the ability to specify the relative importance of different thread groups offers new flexibility in hard real-time

programming, its use can potentially harm the semantic correctness of the system and is the subject of future research. 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 schedulability response. The WCET of 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**. The return value of **sched_no** indicates either that the scheduler has not been given enough time to attempt to find a schedule or that the scheduler has completed its search and not found a schedule.

A key challenge is the ability to implement this thread construct such that its WCET is **max_wait**. Figure 5 shows the fundamentals of the usage of the **sync_thg_sched** construct in the UMass Spring kernel. The parent thread, *Thread1*, synchronously attempts to spawn

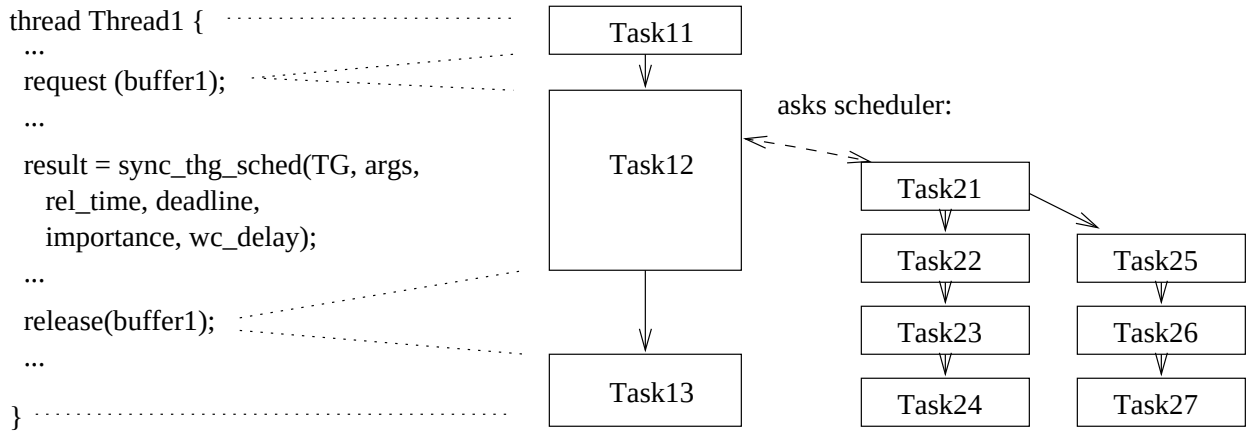


Figure 5: Spawning an "Independent" thread group

a thread group. In *Thread1*, a request/release of a resource is included to illustrate how the parent thread is mapped to tasks. The execution of *Task12* requires dynamic interaction with the scheduler on the SP. The thread group attempting to be spawned, **TG**, is shown at the right of the figure. **TG** is a run-time-specified name of the thread group. If the scheduler can build a schedule consisting of its currently-scheduled tasks and the tasks of **TG** such that *Task21* will not begin executing before 'rel_time' and both *Task24* and *Task27* will complete executing before 'deadline', the scheduler will return **sched_yes**, otherwise **sched_no**. Note that the maximum amount of time that the parent waits for a response from the scheduler is **wc_delay**.

The determination of the maximum amount of time the parent thread can afford to wait for a schedulability response is the responsibility of the application programmer. A programmer must be aware of the execution times of the tasks of the parent thread, so that a maximum amount of time can be determined to wait for a schedulability response, such that schedulability of the *parent* will not be sacrificed. Overspecifying **max_wait** adds to the WCET of the parent thread, thus reducing schedulability. Under-specifying **max_wait** reduces that amount of time the scheduler has for a guarantee decision, thus

reducing the chances of finding a feasible schedule. While the compiler can determine the WCET of the child thread group, the compiler cannot determine the characteristics of the application environment, which influences the amount of time necessary for the schedulability response (further addressed in Section 4). The determination of **max_wait** in each instance is currently done by hand. Tools are being developed to aid the programmer in the determination of **max_wait** though the use of design time analysis and simulation.

3.3.4 Asynchronous, Independent Thread Group Spawning

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

An alternative method for spawning a thread group that requires an Independent guarantee is through the **async_thg_sched** and **check_status_async_thg_sched** combination. The **async_thg_sched** construct is used by a thread to attempt to spawn a new thread group and then continue its processing, without waiting for schedulability analysis on the spawned thread group. In other words, it is not synchronizing with the scheduler. 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 scheduler. This message queue for the scheduler has been partitioned according to AP so that the application executing this statement does not have to potentially compete with another application, which would compromise predictability.

RESULT check_status_async_thg_sched (request_num)
int request_num;

The **check_status_async_thg_sched** construct is used to check the status of an asynchronous 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.

The **async_thg_sched** and **check_status_async_thg_sched** combination provides additional flexibility. In the synchronous version, the parent thread must wait, either by blocking or spinning, for a response. With the **async_thg_sched** construct, the parent can continue computation, and can check at some later point for schedulability results. The added flexibility does not sacrifice predictability; however, the use of this construct requires that the parent thread itself manage time, rather than allowing the operating system to manage time for it. The parent thread must explicitly decide when to check for schedulability results.

3.3.5 Dependent Thread Group Spawning

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

The **thg_spawn** construct is used for a thread that requires a Dependent guarantee (Figure 6). In contrast to the constructs used to spawn thread groups that require Independent

guarantees, at the time that the child thread group is logically spawned, there is no interaction with the scheduler. The WCET of the statement is zero, because this statement does not result in executable code, but rather a compile-time operation that "attaches" the spawned thread group directly to the parent thread. Because this statement results in scheduling information being extracted at compile time, most parameters must be set at compile time.

Figure 6 indirectly illustrates how the **thg_spawn** construct is implemented predictably in the UMass Spring kernel. The use of the **thg_spawn** construct results in the tasks of the child thread group being directly attached to the tasks of the parent. By scheduling the child thread group at the same time as the scheduling of the parent, the UMass Spring kernel directly supports thread groups that require Dependent guarantees. The parent thread in Figure 6 is very similar to the parent thread of Figure 5; however, in Figure 6, *Thread2* consists of 4 tasks because **thg_spawn** requires an additional task in order to prop-

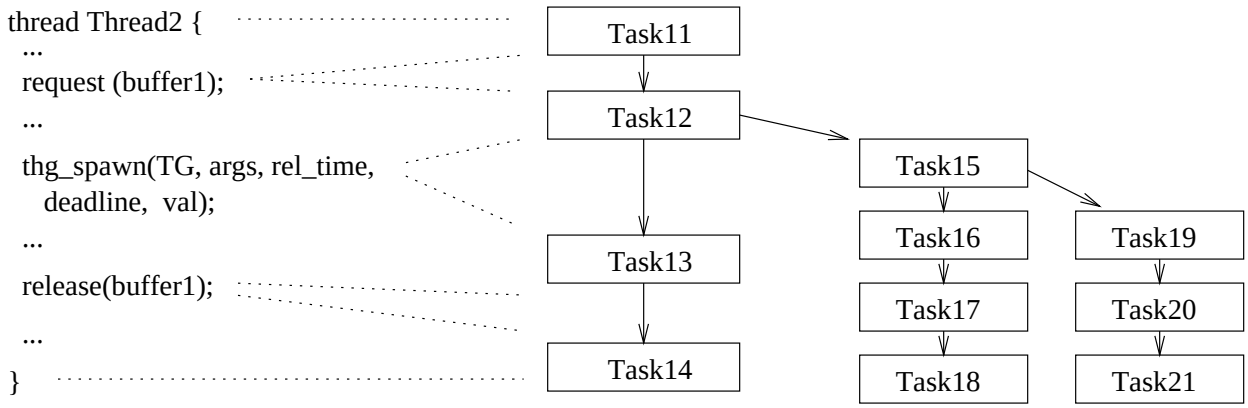


Figure 6: Spawning a "Dependent" thread group

erly define precedence constraints. The difference between the two types of guarantees is the task set that is initially scheduled: at the time of scheduling the parent (*Thread1*), in Figure 5, 3 tasks are scheduled; in Figure 6, 11 tasks are scheduled at the time *Thread2* is scheduled.

3.3.6 Premature Thread Group Termination

int thg_kill_by_id (int id)
int thg_kill_by_name (char *name)
In dynamic, hard real-time environments, it is important to be able to react to unexpected events. These two constructs can be used to stop the execution of a thread group before its normal termination. The invocation of these constructs can cause the termination of multiple threads executing or scheduled to execute on multiple APs in the Spring node. As an example of the use of these constructs, assume that a thread group is executing to control the movement of a robot. If a thread within this thread group, or some other thread, notices an unexpected object in the path of the robot, the thread group must be

prevented from executing the remainder of its scheduled movements. Failure to terminate the thread group could cause serious damage to either the robot or nearby humans.

Predictability must be assured both for the thread executing the killing operation *and* the thread group being killed. The worst-case duration required to terminate the thread group must be taken into account in the determination of the WCET of the thread executing the termination operation; otherwise, a thread executing either of these two constructs can miss its deadline. For threads being prematurely terminated, the issue is not timing predictability but rather semantic predictability—what will be the state of the system and its resources if one or more threads are terminated in the middle of computation? To facilitate semantic predictability, the termination of a thread cannot occur at an arbitrary point during the thread's execution; rather, a thread will only be killed at the completion of a task in the task group corresponding to the thread. Task boundaries define natural termination points, as a task boundary is created upon the use of the **request** construct, and a task boundary is created upon the use of the **release** construct. Thus, the consistency of data requiring mutual exclusion and protected via a **request**/**release** pair will not be compromised as the result of premature thread termination, since the thread will not be killed after the **request** and before the **release**.

Because a thread can be prematurely terminated only upon the completion of a task in the task group corresponding to the thread or before the thread begins executing, the design of a thread subject to premature termination is usually straightforward. That is, the designer of a thread has more knowledge about the operating environment than the designer of a non-real-time thread, so the designer has a better understanding of which threads might require premature termination. To determine the integrity of the system as the result of the possible premature termination of each of these threads, the state of the system at each task boundary must be evaluated (task boundaries are easy to determine). In most situations, because resources are either being acquired or released, the system state is safe. In the event that a task boundary results in an unsafe state, the thread can be redesigned.

These well-defined termination points are analogous to POSIX cancellation points (Section 6.1). In POSIX, a user has the ability to define the places at which a thread can be canceled and the ability to specify the cleanup handlers for each cancellation point that should be executed if the thread is terminated. The ability to support user-specified thread cancellation points and their corresponding cleanup handlers in the UMass Spring threads package on a per-thread basis is an important subject of continuing research. This has not already been incorporated into the UMass Spring threads package because by design the WCET of the cleanup handlers must be taken into account during the admission test. That is, the premature termination cannot sacrifice the schedulability guarantee given to a thread group that is not involved in the thread group termination—either as the thread issuing the request to terminate a thread group or as the thread group being terminated. The recognition that cleanup handlers of even moderate length can compromise the integrity of the system is beyond the scope of the POSIX standard; in the future, these cleanup handlers will be directly incorporated into the UMass Spring threads package.

3.4 Sample Usage of the Thread Constructs

Real-time thread packages have been shown to be needed in real applications and commercial packages are now available (see Section 6). UMass Spring threads extend the capabilities of real-time thread packages in several significant ways. To better understand the use of the new semantics provided by the UMass Spring threads package, a simplified example is presented. Note that this example is intended to be representative of a dynamic, hard real-time environment, but does not illustrate a complete system.

> A sensor on an autonomous vehicle running the UMass Spring kernel notices an object in the projected path of the vehicle. The physical characteristics of the sensor define that, once an object is observed, the vehicle is given 4 seconds in which to plan and schedule a suitable reaction to the presence of the object. If no specific action can be planned and scheduled, the system should halt its movements.

The key to this scenario is that the system must respond predictably to an unexpected event. UMass Spring threads enable the user to code such requirements such that predictable execution is assured. The pseudocode contains calls to spawn thread groups that require both Dependent and Independent guarantees, as shown in Figure 7 and Figure 8. Figure 7 shows the main thread that is spawned as a response to the sensor observing

```
reaction = identify_unknown_object();
if (reaction == TURN_LEFT)
  async_thg_sched("turn_left_thg", (), 0, 2000);
if (reaction == TURN_RIGHT)
  async_thg_sched("turn_right_thg", (), 0, 2000);
thg_spawn("halt_all_movements", (), 2000, 4000);
```

Figure 7: Main *React* thread in sample application

```
thg_kill_by_name("halt_all_movements_thg");

turn_vehicle_left();
```

Figure 8: *Turn Left* thread in sample application

the object. Figure 8 shows the thread to turn left. The thread to turn right is similar. The thread to halt all movements has not been shown.

The description of the pseudocode is that a periodic thread (not shown) executes to monitor the sensor. If the sensor notices an object, a thread group consisting of a single *react* thread is spawned. The first part of the *react* thread consists of code ("identify

unknown object") that performs object identification, movement, or friend-or-foe. The execution time of this code is calculated at compile-time, through code analysis. If the computation results in a decision to turn left, the *react* thread attempts to spawn the necessary work to make the vehicle turn left. Similarly, the *react* thread could result in a decision to turn right. This decision to perform a "non-default" action requires dynamic interaction with the scheduler. If the scheduler cannot create a schedule for the non-default action, ultimately because of lack of resource availability, the default action, which is to halt the movement of the vehicle, must execute. The **thg_spawn** construct facilitates this, because the actions of the *halt movement* are guaranteed at the time that the *react* thread is admitted into the system. In other words, if the *react* thread is executing, the current schedule includes the tasks for the *halt* thread. The vehicle will not both turn left and halt, because of the deadline and release constraints placed on the turning and halting actions—the *halt* thread occurs no later than 4 seconds and no earlier than 2 seconds from the introduction of the *react* thread. The actions to turn the vehicle either left or right completes no later than 2 seconds. The first action of the *turn left* or *turn right* threads is to remove the *halt* thread from the current schedule. Therefore, only one of the default and non-default actions will execute.

The schedulability of the main thread itself can be guaranteed off-line by treating this thread as a periodic thread that is built into the schedule off-line, or it can be performed on-line by making the priority of the main thread higher than any currently-executing thread. Strictly speaking, this does not adhere to all-or-nothing guarantees at the time the thread is admitted for execution. However, this can be accommodated in the event of a mode change. Also, it should be noted that, for simplicity, all thread groups in this example consist of a single thread. For example, the statement to spawn the *turn_left_thg* actually spawns only the single thread that is shown in Figure 8.

4 Implementation of the Thread Package

The previous section described the design of the threads package, focusing on the new semantics for dynamic, hard real-time computation, and finished with an example of using this package. This section discusses the implementation of the threads package in the UMass Spring kernel. Only those existing areas that required significant modifications are discussed. Specifically, the **request** and **release** constructs are not discussed because they were the basis for synchronization in the pre-threads UMass Spring kernel and were not modified.

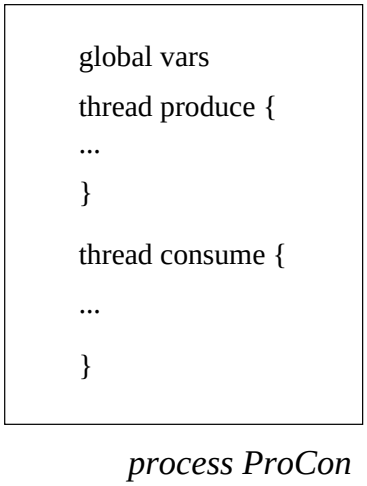
4.1 Implementation of create_process and create_thread

The purpose of the **create_process** system call is to establish and initialize the physical memory for the address space shared by all threads in the process. An SDL statement is used to instruct the UMass Spring kernel which physical memory to use. This system call should be executed only under non-hard-real-time constraints.

The **create_thread** system call allocates and initializes a Thread Control Block (TCB) in physical memory on the AP that is designated by the SDL statement as having the capabil-

ity of executing a given thread. The TCB is configured such that it points at the physical memory created by the corresponding **create_process** system call. Note that while the process itself–the address space—is resident on only one AP, multiple APs can be configured to execute threads from the process. In this case, to reduce contention on the bus, the code segment of the process is replicated on any AP that might execute a thread from the process. In other words, the **create_thread** system call loads the code segment of the process onto the AP in question, if the code segment of the process is not already present. The scheduler selects the AP on which to execute the thread.

An example illustrates how the user instructs the UMass Spring kernel to invoke these operations at boot-time, thus, establishing the working environment. Figure 9(a) shows the static definition of a sample process named *ProCon*, for *Produce/Consume*. This process contains two threads, *produce* and *consume*. Figure 9(b) shows the SDL statement used to instruct the UMass Spring kernel how to lay out memory. For simplicity, only two



(a) Pseudocode Spring-C process definition for sample process

```
SDL {
    Processor_layout(AP1) {
        Process_set   ProCon;
        Thread_set   produce(3), consume(2);
     }
    Processor_layout(AP2) {
        Thread_set   produce(1), consume(2);
     }
 }
```

(b) SDL statement to define layout of sample process

Figure 9: Spring-C and SDL Statements for Sample Process

APs are shown. Figure 10 illustrates the layout of physical memory that results from the combination of the process *ProCon* and the sample SDL statement.

The implementation of the UMass Spring threads package required modifications to SDL[4], which are reflected in Figure 9(b). The *process_set* keyword informs the kernel of the placement of the read/write segments in physical memory. In Figure 9(b), the read/write segment of process *ProCon* resides in physical memory of AP1. The *Thread_set* keyword instructs the kernel of the number of TCBs that should be established for each thread, on a per-AP basis. For simplicity, it is assumed that each thread in the system has a unique name. In Figure 9(b), for example, the layout of AP1 includes three TCBs for the produce thread, and two TCBs for the consume thread.

[4]This illustrates the relatively tight coupling that exists between levels in a real-time system.

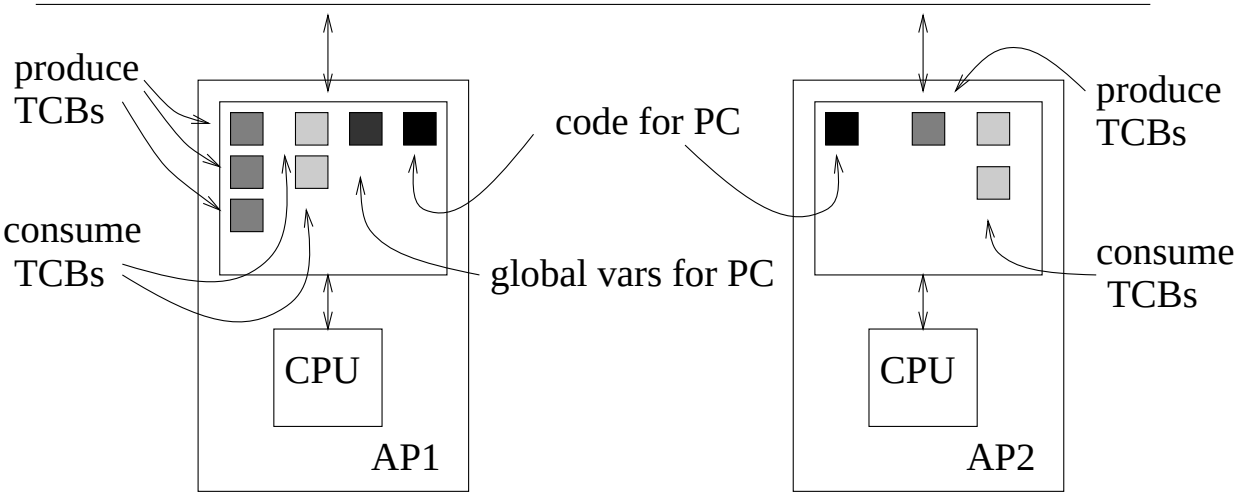


Figure 10: Resulting layout of physical memory given SDL statement and static process structure

4.2 Spawning Thread Groups with Independent Guarantees

The implementation of the **sync_thg_sched** construct required substantial improvement in the communication mechanisms between the scheduler and user applications. A shared segment was created between the SP and each AP, in order to facilitate predictable communication. For development purposes, it was decided to delay the appropriate partitioning of this shared segment into per-thread segments. That is, in the current design, there is a single segment by which all threads on an AP communicate with the scheduler. A limitation with this design is that a thread itself must search the responses of the scheduler to find the message specifically intended for it. This does not impair functionality or predictability; it does, however, increase the WCET of the function to retrieve a message from the scheduler.

The second problem addressed during the implementation of the **sync_thg_sched** construct is the ability to achieve a predictable response time from the scheduler. Previous scheduling research in the Spring project resulted in an $O(kn)$ version of the scheduling routine, where n is the number of tasks in the schedule. The basic idea is that at each of n scheduling choices, a heuristic function is applied to the top k choices in the list of unscheduled tasks sorted by increasing deadline; the highest-valued task is placed into the schedule. If k is constant, the scheduling algorithm is $O(n)$ [8]. For a particular system, the system is designed with a maximum value of n. The implementation of the Spring threads package utilized this result in two steps:

1. *As part of the kernel parameters file, the user specifies an absolute worst-case execution time for the scheduler given no pending requests*. This is the time that a user application both expects and allows the scheduler to take to schedule user requests, if no scheduling requests from other threads on the same AP or a different AP are pending. This value is used as a guideline for specifying the worst case duration that a thread will wait for a response from the scheduler (i.e. the **max_wait** param-

eter in the **sync_thg_sched** construct, and the duration between the invocation of the **async_thg_sched** and the **check_status_async_thg_sched** construct). As stated in Section 3.3.3, tools and compiler support are being developed to aid the application programmer in the selection of this value in the kernel parameters file and **max_wait**.

2. *The duration of the scheduler is ultimately dependent on the total number of applications of the heuristic function; the user-specified WCET is used to dynamically select the value of k on a per-scheduling-request basis*. The scheduler was empirically evaluated under different operating environments in order to determine the execution time as a function of the number of applications of the heuristic function. The compiler could not be used to determine the WCET of the scheduler, because, although the scheduler executes at user-level, it is not constructed like an application that executes on an AP; rather, it is a unique application that executes on the SP. Determination of a compiler-computed WCET for the scheduler will be pursued in the future, when the scheduler is redesigned to accommodate a worst-case timing analysis by the Spring-C compiler. An experimental determination of the execution time of the scheduler is feasible for development purposes, because a scheduler that executes longer than its WCET as specified in the kernel parameters file will simply return **sched_no**. In other words, the system will not fail due to timing overruns, although threads might fail to be scheduled in extreme cases.

The first step allows the user to configure the operating system for the specific environment in which the system will be deployed. A larger scheduler WCET is specified when deadlines are not very tight. The second step in the implementation allows the scheduler to dynamically adjust its scheduling technique to fit the run-time situation. Note that in the unlikely event of simultaneous requests, the system will not fail because of timing overruns; rather, one or more of the thread groups might not be scheduled.

4.3 Spawning Thread Groups with Dependent Guarantees

Whereas the implementation of most of the constructs of the UMass Spring threads involved either system calls or middleware, the implementation of the **thg_spawn** construct required direct modifications to the Spring-C compiler. The compiler was modified to create a task boundary at the point that **thg_spawn** is used, in order to properly encode precedence. In other words, it creates a scheduling point. At the time that the kernel is booted, the tasks of the thread group being spawned are physically attached to the parent thread. The current implementation views the child thread group as being indistinguishable from the parent thread, so no new *ID* is generated for the child thread group.

4.4 Thread Group Killing

The general issue in abnormal thread termination in hard real-time environments is how to achieve functional and timing correctness in a multiprocessor real-time operating system in circumstances that are abnormal with respect to timing and scheduling. This is

generally not addressed in traditional, commercial real-time operating systems because computations are assumed to be independent. In the UMass Spring operating system, thread group killing can affect multiple threads in multiple processes on multiple processors, all executing under hard real-time constraints.

The implementation of the killing operations illustrate how the most logical design is not often the best design. Because the logic of the operations is to engage the scheduler to remove selected tasks from the current schedule, the first implementation utilized a separate thread within the scheduler process to modify the pending schedule. This approach was pursued because of its promise to immediately remove those tasks that were part of the thread group to be killed—the resources scheduled for use by the tasks could be immediately reclaimed. This approach was not pursued because removing tasks required a new schedule to be built. The time to build a new schedule was too long to ensure that tasks of the killed thread would be not executed. In addition, interrupting the scheduler at arbitrary points made the scheduler's duration unpredictable.

Instead, **thg_kill_by_id** is a system call that writes the *id* of the thread group to be killed into segments shared between the AP dispatchers as well as the scheduler. The AP kernel dispatches the next task only if its *id* has not been specified as being killed. The scheduler, as part of its normal operations, both removes tasks from the current schedule that have been killed and resets data structures in the AP dispatcher segment. Threads can be killed before they start executing, or during their execution (after waiting for the currently-executing task in the thread to complete executing).

An additional consideration not present in non-real-time operating system is that the dispatcher had to be modified to reset the task's TCB in the event that it is about to dispatch a task that has been killed. A TCB must be reset because it is a static, reusable data structure allocated and initialized at boot-time. A normal termination resets the TCB, but a killing requires "special" resetting. The scheduler cannot reset the TCB because the TCB is not valid in scheduler space. The AP has the time to reset the TCB, because it would have been executing the task had it not been killed. As long as the resetting of the TCB takes less time than the smallest task execution time then this operation is allowable under hard real-time constraints.

5 Performance/Predictability of the Thread Package

This section presents the performance evaluation of the constructs of the UMass Spring thread package as implemented in the UMass Spring kernel executing on four 16.67 MHz 68020-based MVME136A boards in a VME cage[5]. The focus in this section is the ability of the implementation to be predictable and provide the semantics as defined by the UMass Spring threads package. The **create_process** system call and **create_thread** system call are not discussed, because they are not expected to be used at run-time under normal circumstances. The measurements contained in this section were performed using

[5]It is important to note that while the 68000 family of processors is old, many real-time systems still use such processors. Further, the real scientific issues revolve around predictability and not raw speed. The main concepts investigated here are not affected by using the 68020 CPUs. See [15] for a discussion of why real-time computing is not fast computing.

clocks that exist on the particular MVME136A board. The granularity of the clocks is 0.5 nanoseconds.

5.1 Independent Thread Group Spawning

The spawning of a thread group that requires a Dependent guarantee is predictable because the scheduling of the spawned thread group occurs at the time that the parent is scheduled. However, spawning a thread group that requires an Independent guarantee requires the real-time coordination between the parent thread, which is executing on an AP, and the scheduler, which is executing on the SP. Only the measurement of the asynchronous thread group spawning and status checking is discussed; one way in which to view the synchronous thread group spawning is as a combination of these two constructs.

The measurements of the two operations to perform asynchronous thread group spawning is shown in Table 1. The sample size is 100. The times are in μseconds. The duration is the time required to issue the system call and then return to user space. The values in

Table 1: Timings of constructs for asynchronous spawning of thread group

	minimum measured	*maximum measured*	*computed WCET (one message)*	*computed WCET*
async_thg_sched	519	590	—	792
check_status_async_thg_sched	32	52	98	934

the last column are the compiler-computed WCET for each of the two operations; these values are used to compute the WCET of any thread executing either of these operations. The table also shows the maximum and minimum measured duration for each of the two operations. The reason for the third column—computed WCET (one message)—will be described shortly.

There are two explanations for the difference between the compiler-computed WCET and the measured maximum duration. First, the technique used to compute the worst-case execution time for these experiments does not take into account pipelining effects during program execution, because a detailed model of the pipeline was unavailable [5]. Second, the compiler computes a worst-case behavior, while the measurements for this table did not stress the worst-case. Wide variances in execution time are inherent in a thread package for dynamic systems, as the design must take into account rarely-occuring events. In contrast, small, static systems often achieve their predictability at the cost of flexibility.

In general, worst-case behavior for these experiments was difficult to produce, because it often required the precise occurrence of multiple events on multiple processors. For **check_status_async_thg_sched**, instead of attempting to create and measure worst-case behavior, measurements were made on average-case behavior and compared to compiler-computed WCET for these scenarios. The code of **check_status_async_thg_sched** steps through a message queue associated with the scheduler that is length 10 until finding a message related to the schedulability of the thread group in question. The time

required to do this as calculated by the compiler is 934 μseconds. In most cases, there will be only one message or no messages in this queue. The shortest duration occurs when there are no messages in the queue, which was measured at 32 μseconds. Modifying the code of **check_status_async_thg_sched** to reflect the absence of messages (leaving only the code in the body that is executed when no messages are present) resulted in a compiler-computed WCET of 75 μseconds. In other words, the compiler computed that it would take 75 μseconds, while measurements indicate that it took 32 μseconds. Similarly, the compiler-computed WCET for the case of only one message in the message queue is 98 μseconds (shown in Table 1) as compared to the measured 52 μseconds (there were never 2 or more messages in the message queue for those cases measured). The fact that the compiler-computed WCET is reasonably accurate for these two cases supports the argument that the computed WCET for the true version of **check_status_async_thg_sched** is not unduly pessimistic, even though nothing close to it was ever measured by these experiments. Note that the "Computed WCET (one message)" entry has not been shown in Table 1 for **async_thg_sched** because the algorithm for **async_thg_sched** is not subject to a large discrepancy between worst-case and average-case behavior.

While these times measure primitive operations, a separate, broader issue is the ability of the user code to dynamically interact with the scheduler to spawn new work: for example, the worst-case execution time of **check_status_async_thg_sched** is not the time to receive a schedulability response from the scheduler, but rather the time it takes to *check* for a response from the scheduler. Figure 11 and Figure 12 show the measurement of actual execution duration of the scheduler as a function of the WCET of the scheduler as specified in the kernel parameters file. To understand the data contained in these plots, consider the plot in Figure 12 where the user specifies that the scheduler cannot take more than 450 milliseconds. The plot shows that if the size of the schedule is less than 42 tasks, then $k = \infty$, and every unscheduled task is evaluated at every scheduling point. When the schedule contains more than 42 tasks, k must be set to bound the time to 450 milliseconds. Figure 11 also shows that 50 milliseconds is not a reasonable value—given only 50 milliseconds to schedule, if the number of tasks is large, the response will not generally arrive before user code checks for a response. Note that the maximum actual duration of the scheduler is approximately 50 milliseconds *less* than the user-specified WCET, because a "window of safety" has been built into the dynamic calculation of k. In other words, the scheduler is given 50 milliseconds less than the user-specified WCET in case the scheduler takes longer than it is estimated that it would. This margin of error is included to further safeguard that the scheduler completes by the user-specified WCET.

Overall, Figure 11 and Figure 12 show that this approach can lead to predictable performance: given a WCET, the scheduler will schedule and respond before the user-specified WCET. Two additional observations are made. First, it must be stressed that the target set of applications are designed to execute in dynamic real-time environments, consisting of many applications comprised of many computations; each application may have a single end-to-end deadline on the order of seconds (for the current hardware platform). For these applications, 450 milliseconds is a reasonable amount of time to wait for a determination of schedulability. Additionally, the duration of the scheduler will be significantly reduced on updated hardware, increasing the usability of the threads package for complex real-time applications. Second, the selection of the value for the user-defined

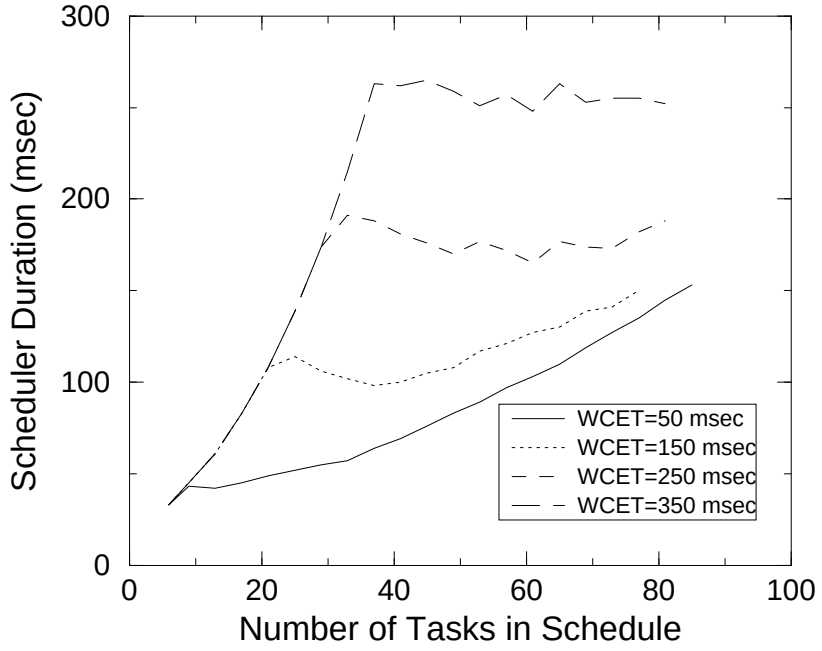


Figure 11: Ability of scheduler to terminate before its deadline: deadline=50-350 msec

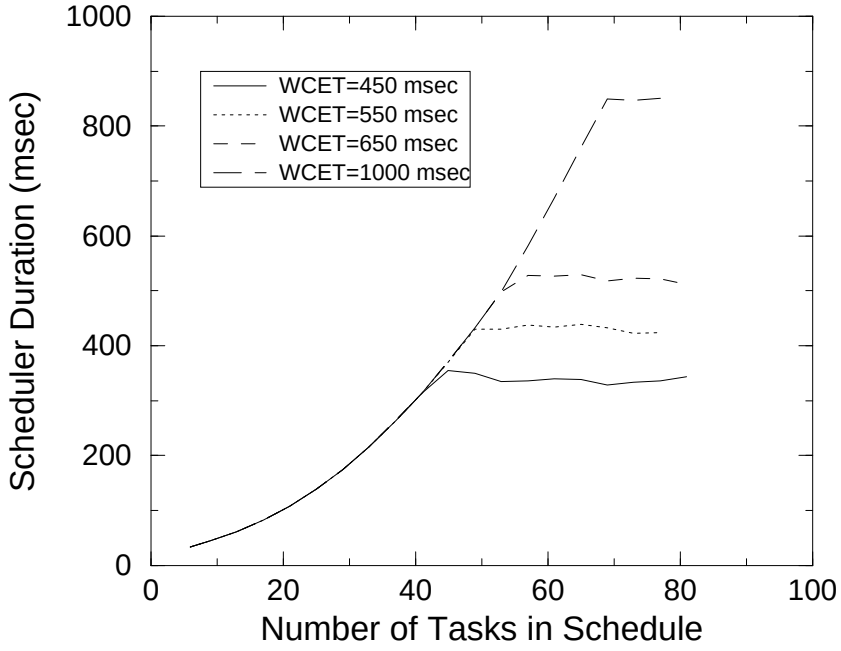


Figure 12: Ability of scheduler to terminate before its deadline: deadline=450-1000 msec

WCET of the scheduler as specified in kernel parameters file is currently left to the user, based on the characteristics of the environment. Further research is required to develop tools that will facilitate the selection of the best value for the particular operating environment.

5.2 Thread Group Killing

Measurements were performed to determine the duration and predictability of the operations to kill a thread group. The duration is the time required to perform the system call and then return to user space. However, because the code does not actually return to user space after making the system call, because the compilation of a routine to kill a thread group ends the current task, the duration is approximated as the difference between the end of the system call in kernel space and immediately prior to the system call in user space. A single clock is used—accessed in kernel space in one call and accessed in user space in another call. The results of the measurements are shown in Table 2. The sample size is 50.

Table 2: Timings of constructs for premature termination of thread group

	minimum measured	*maximum measured*	*computed WCET (one message)*	*computed WCET*
thg_kill_by_id	360	369	412	1116
thg_kill_by_name	454	511	907	3706

In Section 5.1, there was a large discrepancy between the worst-case measurement of **check_status_async_thg_sched** and the compiler-computed WCET; the same explanation can be applied to the two constructs measured in Table 2. In the case of **thg_kill_by_id**, the compiler computed the execution time when it was necessary to place the id of the thread group in the last position of the an array used for communication between the application threads and the scheduler. The compiler-generated WCET for code that places the id into the first position of the array, which is always the case for the measurements, is 412 μseconds, which is reasonably accurate. Similarly, the compiler-generated WCET for the measured case of **thg_kill_by_name** is 907 μseconds. In both of these constructs, the worst-case behavior accounts for unexpected events in dynamic environments, even though these events will rarely occur. Scheduling of threads that use these constructs is based on the worst-case behavior, so that the system is predictable even when these rare events occur. Measures can be taken to improve system performance when computation does not require the worst-case behavior, as discussed in Section 5.3.

It was noted that in the event that a thread is killed, the TCB must be reset. As long as the WCET of the operation is less than the scheduling granularity, the schedule created by the scheduler is dispatched correctly by the dispatchers. To measure the duration and predictability of this operation, the kernel operation invoked in the dispatcher to reset the TCB was measured. For 100 samples, the minimum time measured was 824 μseconds, and the maximum time is 827 μseconds. It is assumed that 827 μseconds is a

reasonable estimate of the worst-case execution time, given that the code is a straight-line segment with no branches. 827 μseconds is significantly less than the current scheduling granularity (10 milliseconds), so predictability is retained if the AP dispatcher resets the TCB.

5.3 Additional Performance Improvements

The primary goal of the UMass Spring threads package is to support new semantics for hard real-time computation, such that the execution of these constructs is predictable. An additional goal is to improve the *average-case* performance of the kernel, irrespective of worst-case performance estimates. When activities take less than their worst-case durations, resources can be reclaimed and reused [16].

The first improvement is the average duration of a context switch. In the UMass Spring kernel, the contents of the Translation Lookaside Buffer (TLB) are explicitly managed in order to avoid "misses" during memory references. That is, when a context switch to a new address space occurs, the TLB is flushed and reloaded with the TLB contents appropriate for the new process (this operation is referred to as an "expensive" context switch; the alternative is referred to as a "fast" context switch). The alternative to this explicit management is to incur TLB misses as a thread executes, which could lead to unpredictable memory access times.

By introducing new scheduler heuristics that seek to minimize the number of expensive context switches, the UMass Spring threads package reduced average-case context-

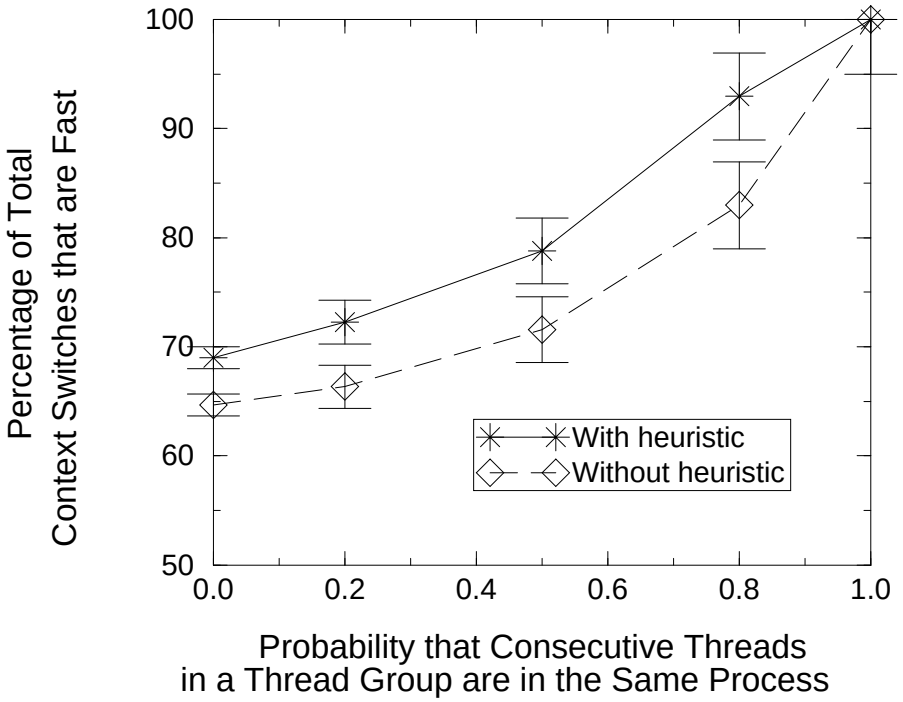


Figure 13: Scheduling to avoid "expensive" context switches

switching overhead. A heuristic was added to the scheduler to prefer to stay *within process* when determining the next task to schedule. Figure 13 shows the effects of this heuristic, as compared to the version of the scheduler before the implementation of the heuristic. The percentage of context switches that are fast is plotted against the probability that consecutive threads in a thread group are in the same process, which is a rough indication of the user's use of multiple processes. For example, 0.0 is consistent with a coding style in which every process has a single thread, and 1.0 is consistent with a coding style in which there is a single process and multiple threads. The simulation parameters used were: 9 TCBs per thread per AP; 3 tasks per thread; task WCET was drawn from a uniform distribution from 50 milliseconds to 1 second; 5 thread groups; 3 threads per thread group; Poisson arrivals of thread groups, with $\lambda = 2.5 * \sum(\text{task WCET})$; deadline was 3 $\times$ the sum of the WCET of tasks. In the formula for λ, the factor of 2.5 is used to set a rate of interarrival such that on average there was 1.5 seconds of idle time for every 1 second of potential computation. Each data point is the average (with 95% confidence intervals) over 50 runs, where each run is defined as 200 seconds of system operation. In each run, 95-100 thread groups arrived; 100% of the thread groups were schedulable. Note that if only processes were used (probability that consecutive threads in a thread group are in the same process is 0.0), there were still fast context switches, when a context switch is performed between one task in a thread and the next task in the same thread.

A second performance gain is a reduction of the average-case cost of memory references. For simplicity, at compile time, the WCET calculation of a task assumes that all memory references to global variables require the VME. However, the VME is not required if a thread executes on the processor in which the address space resides, because an on-board bus is used. For example, in Figure 10, the execution of a *Produce* thread on AP1 is local, because the global variables of process *ProCon* are resident on AP1. To decrease average-case memory reference costs, two scheduler heuristics were added that seek to execute threads on the same AP on which the address space resides. TCBs are viewed as ordinary resources by the scheduler; as such, the scheduler maintains a list of the "earliest available time" for each TCB, and dynamically schedules based on this information. The two heuristics are:

Only Local Choose a TCB on the AP on which the process address space resides, if one exists. Otherwise, choose the TCB that is available closest to, but not earlier than, the release time of the thread. This is analogous to first sorting by AP, and then doing a *stable sort* [17] by available time. The first in the list is then chosen.

Earliest Choose a TCB on the AP on which the process address space resides, if it is available at the time it is needed. Else, choose the TCB that is available closest to, but not earlier than, the release time of the thread. This is analogous to first sorting by available time, and then doing a stable sort by AP. The first in the list is then chosen.

The intuitive difference between the two heuristics is that the **Only Local** heuristic will greatly prefer executing threads locally, perhaps at the cost of schedulability. Figure 14 shows the effect of the two heuristics on schedulability as the tightness of deadlines of

the arriving thread groups is decreased under moderate load conditions. "Deadline Multiplier" is the laxity for each scheduling request, as a function of the sum of the WCETs of the tasks in the scheduling request. Higher values for the deadline multiplier increases the inherent schedulability of a request. Figure 15 shows the effect under heavy load conditions. For these simulations, various parameters were modified: number of processes (1-10, uniform); threads per process (1-10, uniform); tasks per thread (1-8, uniform); task WCET (50-500 milliseconds, uniform); threads per thread group (1-5, uniform). In order to reduce the effects of lack of physical TCB structures on schedulability, each AP supported the execution of 9 instances of each thread, which was more than needed. In addition, the number of thread groups was fixed (5) in order to roughly fix the load on the system. The probability that consecutive thread groups in a thread belonged to the same process was fixed at 0.7. A moderate load is defined as an average interarrival time of 5.0 $\times$ the sum of the WCETs of the tasks, while a heavy load is defined as an average interarrival time of 2.5 $\times$ the sum of the WCETs of the tasks. The graphs show the conflict between wanting to reduce memory access costs and the need to meet deadlines. The **Only Local** heuristic achieves the former, as all memory references are local. However, using this heuristic greatly sacrifices schedulability. Therefore, the **Earliest** heuristic is clearly the better choice of the two. It has the advantage of reducing memory access costs without reducing schedulability. With more local executions, the average case durations of tasks will decrease, which will lead to more opportunity to perform resource reclaiming. Additional details are found in [18].

6 Related Work

Four approaches for real-time threads are discussed and contrasted with UMass Spring threads: the real-time threads of the POSIX standard, Solaris real-time threads, Real-Time Mach threads, and the threads of CHAOS-arc. For completeness, the main properties of several non-real-time thread packages are also briefly discussed and contrasted to real-time thread packages.

6.1 POSIX Threads

The POSIX standard [19] includes amendment P1003.1c for threads (Pthreads), which by default are used for non-real-time processing. Because the PThreads document specifies only the API, 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 example, the standard does not require that the threads be supported in kernel-space, nor does it require that the threads be supported in user-space.

The basic PThreads thread model is priority driven and preemptive, with signal handling, mutual exclusion, and synchronized waiting [20]. PThreads provides semaphores, locks and condition variables as the primary means for synchronization. In general, the scheduling policy used in non-real-time is round robin, which is denoted SCHED_RR. A thread has attributes that can be configured at creation time, including scheduling policy, scheduling priority, scheduling scope, scheduling parameters, and stack size.

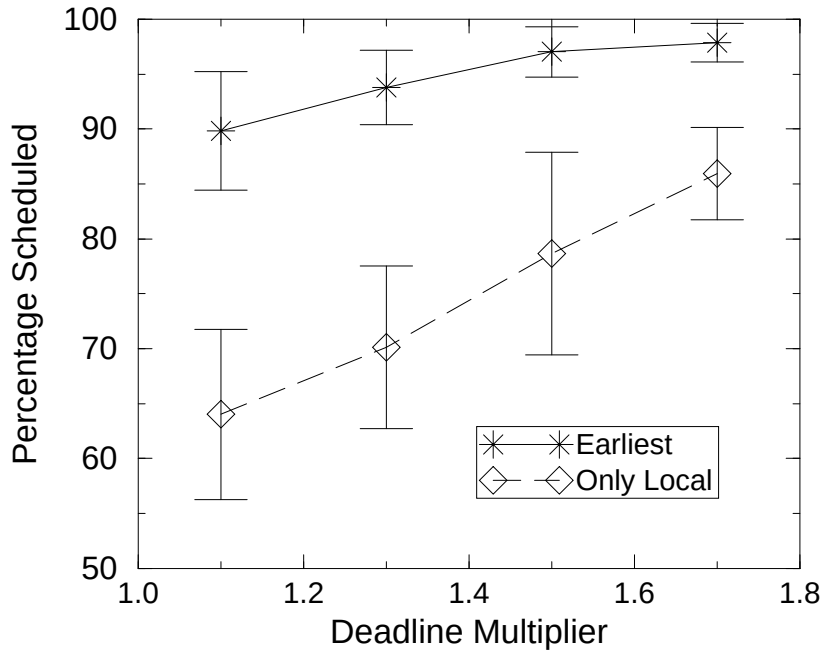


Figure 14: Effect of different scheduling heuristics when operating under a moderate load

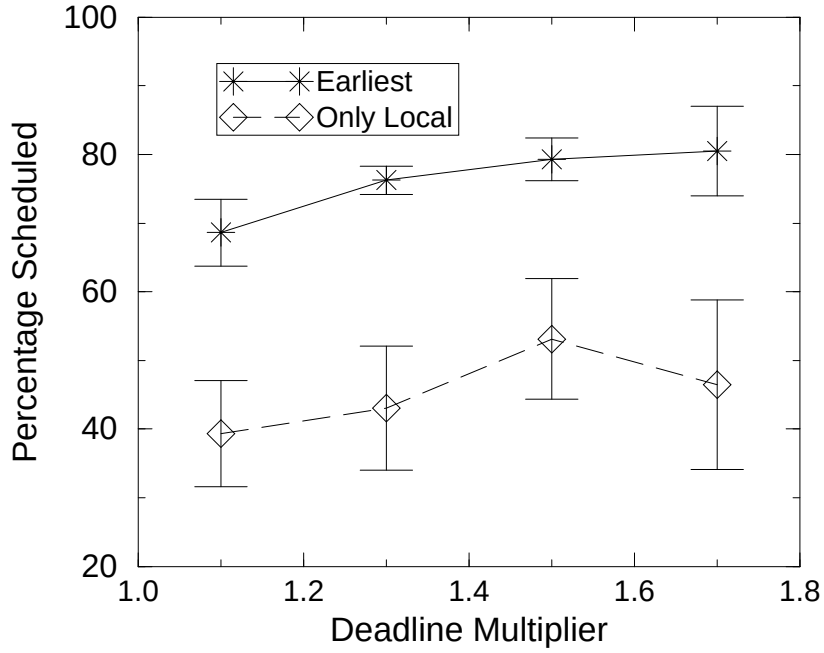


Figure 15: Effect of different scheduling heuristics when operating under a heavy load

PThreads defines extensive interfaces for signal handling and thread cancellation. 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 cancellation is well-defined with specified cancellation points and optional cancellation cleanup handlers. PThreads also provide for thread-specific data, to allow threads to have thread-specific global variables.

The POSIX realtime amendment, P1003.1b, also contained in [19], extends the base POSIX standard in order to achieve "bounded response time". The two primary scheduling policies defined in POSIX for real-time computing are SCHED_FIFO (first-in, first-out), and SCHED_OTHER, which can be used to support implementation-defined policies. Functions are provided in P1003.1b for setting the real-time thread scheduling policy and for changing its priority. To eliminate nondeterministic delays due to page faults, POSIX allows a thread to lock certain pages of memory. A nanosecond clock and interval timers are provided.

The PThreads interface does not explicitly provide a way to bound blocking on a mutex, which impacts predictability. Only one call provides deterministic behavior for mutex locking (pthread_mutex_trylock(), which does not block if mutex is already locked). Bounded waiting on condition variables is possible through pthread_cond_timedwait(). To provide bounded waiting, PThreads anticipates (but does not require) the implementation of the priority inheritance and priority ceiling protocols. If either SCHED_FIFO or SCHED_OTHER are used, threads waiting on condition variables or mutexes are awakened in priority order.

The POSIX standard also includes amendment P1003.1i, which adds thread synchronization mechanisms common in multiprocessor thread applications. These additions are designed to correct omissions from the original PThreads standard. Examples of these additions include barriers and spinlocks for fine-grained parallelism, and read/write locks.

An earlier version of the POSIX Pthreads standard was used as the basis for MiThOS, which is a small kernel for multi-threaded embedded applications [21]. MiThOS implements priority scheduling and round-robin priority scheduling, but in addition supports deadline scheduling for a thread. That is, the scheduling attributes of a Pthread are extended to allow the specification of an absolute start time, an absolute deadline, and a relative period.

Many key differences exist between UMass Spring Threads and POSIX PThreads for real-time environments. First, POSIX threads are scheduled according to priority, while UMass Spring threads use an underlying planning-based scheduler. Second, any deadlines on POSIX threads will be implicitly ignored during execution, unless additional mechanisms are built to notice when a thread executes past its deadline. In UMass Spring threads, a thread will not execute longer than its compiler-calculated worst-case execution time. Third, POSIX real-time threads are a combination of real-time and non-real-time constructs; in many cases, it is unclear how the real-time and non-real-time components will interact, potentially sacrificing predictability. Fourth, POSIX does not support the specification of groups of activities with single end-to-end deadlines, nor does it allow the ability to specify different semantics on a collection of tasks (such as supported with the Dependent vs. Independent option for creating threads in the UMass Spring threads package.)

6.2 Solaris Real-Time Threads

Threads in the Solaris operating system, which are also referred to as UNIX International (UI) threads, are very similar to POSIX threads. Many of the functions in the Solaris threads package have a counterpart in the POSIX threads package, but there are some notable differences [22]. Features in POSIX threads that are not contained in Solaris threads include attribute objects that can be used to establish characteristics for each thread, cancellation semantics, and scheduling policies.

The most significant feature of Solaris threads not contained in POSIX is the ability to set concurrency through the use of Lightweight Processes (LWPs) [23]. Each LWP is viewed as a virtual CPU, available for executing user code or system calls. Each LWP in the system is separately scheduled and dispatched by the kernel. LWPs add an extra level of control over threads by allowing an application programmer to select the number of LWPs per application as appropriate. In addition, a programmer may choose to bind a thread to a LWP for performance reasons. Without this binding, multiple threads can be multiplexed onto a single LWP.

Realtime scheduling is based on the scheduling classes [24] (this is similar to realtime scheduling in Windows NT [25]). The classes in order of priority are: Interrupt threads, realtime threads, system threads, and timesharing (TS) threads. Scheduling within a class is priority-based and preemptive.

The differences between POSIX threads and Spring threads also apply to Solaris threads, because Solaris threads are very similar to POSIX threads (priority-based scheduling, mutexes, semaphores, etc.). While the additional level of abstraction as provided by LWPs improve flexibility for non-real-time computation, the ability to create and regulate the combination of threads and LWPs is challenging for hard real-time computation.

6.3 Real-Time Mach Threads

There are many similarities between the threads of Real-time Mach and POSIX threads, because both are based on priority scheduling. Real-Time Mach supports five scheduling approaches: Rate Monotonic, Fixed Priority, Round Robin, Rate Monotonic with Deferrable Server, and Rate Monotonic with Sporadic Server. Mutexes and conditions (signals) are used for synchronization. Preemption of a thread within a critical region is on a lock-by-lock basis: no preemption, preemptible, or 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. Earliest Deadline First (EDF) [26] is used as the queuing policy for both RT-mutex and RT-condition primitives. The kernel provides no run-time schedulability support—an application program cannot attempt to create a thread under the condition that the operating system provide immediate feedback concerning its schedulability.

Real-Time Mach supplies many important mechanisms for real-time; however, the user must judiciously use priority inheritance and account for worst-case blocking, lock specific pages in memory, and generally account for all effects of blocking on predictability on their own. The Spring OS provides direct support for predictability by handling

blocking issues via planning-based admission control. This paradigm difference affects various implementation concerns and permits a system-supported predictability.

The kernel-level threads of Real-Time Mach have been extended to support user-level threads [27]. 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 the implementation of UMass Spring threads 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 sustainable if threads were implemented at the user level, because of the lack of a system-wide perspective when making scheduling decisions (at the user-level). If an application could arbitrarily decide to change from one thread to another, resources would be requested and released in an unpredictable manner.

6.4 CHAOS-arc Real-Time Threads

The real-time threads package of [14] are used as the basis for the CHAOS-arc operating system [28]. 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. A "higher-level operating system software" reacts according to the inability of the operating system to guarantee a thread (or a group of threads). This philosophy is closely aligned with the design of UMass Spring Threads—both agree that deadline-based scheduling is more appropriate for dynamic environments than priority-based scheduling, and both use a planning-based approach.

The description of CHAOS-arc threads can be categorized as manipulation, control, and higher-level utilities. To manipulate threads, there are calls to spawn threads of different types: sporadic, periodic, and event-type. The schedulability test is distinct 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.

The design and implementation of UMass Spring threads differs from this work in many ways. The first principle difference is that the UMass Spring threads package is designed for a single multiprocessor, while the real-time threads model of CHAOS-arc was originally designed for a network of uniprocessors. The scheduling system underlying UMass Spring threads is able to opportunistically schedule threads with the intent

of maximizing the utilization of the processors within the single node. This includes the ability to make scheduling decisions recognizing that the cost of access to memory is a direct consequence on the processor that is selected (see Section 5.3). The approach of CHAOS-arc threads is distributed—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 cannot be spawned. The second principle difference is the semantics of a threaded application as a function of kernel support. In the approach of UMass Spring threads, a thread is scheduled based on its WCET but is guaranteed access to required resources *before* the thread executes. The approach described in [14] also schedules a task based on its WCET, but does *not* directly schedule access to required resources. The ramifications of this is that an executing thread can be denied access to a resource if that resource is held by a second thread. 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 UMass Spring threads package does not have this limitation because the design and implementation of UMass Spring threads is integrated with a reservation-based scheduling system.

6.5 Non-Real-Time Thread Packages

The concept of threads has existed for many years. Initially, threads were supported in the kernel of general purpose operating systems, e.g., the Mach threads. Such threads proved to be inefficient and later work demonstrated that user level threads perform best [29, 30]. However, many of the assumptions and requirements that make non-real-time thread packages suitable to run at the user level are not true for real-time threads. For example, user level thread packages enable a particular application to tailor the scheduling policy to its own needs, but there is no direct support for meeting system-wide requirements across applications. In a hard real-time system there is only one application and its scheduling is always tailored to its needs. Further, this is done in a system-wide manner. The kernel seems to be the correct place for supporting the system-wide performance view necessary in real-time systems.

The non-real-time thread packages also enable different thread packages from different applications to interact. This is not a serious need in most hard real-time systems where choosing one package will suffice.

The user level packages also stress average case efficiency via a set of mechanisms that include: shared memory that provides efficient sharing of runtime information between the kernel and application, signals or upcalls to permit the kernel to notify the application thread package of system changes that affect its scheduling, and a standard interface between the kernel and the application level schedulers. In our package we have efficient sharing because the key scheduling information is pre-analyzed off-line and stored in the kernel. Upcalls are avoided in our solution and these are a source of unpredictability in user level packages. Without different user level schedulers a standard kernel-user level interface is not needed.

7 Conclusions

Computation in hard, real-time environments requires the ability to specify timing constraints and semantic interrelationships not currently available with real-time threads packages. The novel differentiation between *Dependent* guarantee and *Independent* guarantee supported by UMass Spring threads allows computations to be arbitrarily grouped, with the specification of unique run-time execution requirements. Constructs are provided to enable the user to perform the premature termination of groups of activities, such that semantic correctness is assured. The UMass Spring threads package presents new flexibility to the user that had not been previously available.

The implementation of the UMass Spring threads package specification was shown to be implementable such that predictability is achieved. Key issues addressed were the use of predictable thread constructs, the synchronization between the scheduler and user threads, the predictability of the scheduler, implementation issues of the spawning of a thread group that requires a Dependent guarantee, and the semantic correctness of the operations to prematurely terminate an executing thread group. Measurements on a development platform of four 68020s affirm predictable operations.

Acknowledgments

The authors would like to thank Gary Wallace for advice concerning the implementation and measurements described in this paper. In addition, the authors thank Krithi Ramamritham, Doug Niehaus, and the anonymous reviewers for their helpful suggestions.

References

[1] John A. Stankovic and Krithi Ramamritham, "Editorial: What is predictability for real-time systems?," *The Journal of Real-Time Systems*, vol. 2, pp. 247–254, 1990.

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

[3] Marty A. Humphrey, Gary Wallace, and John A. Stankovic, "Kernel-level threads for dynamic, hard real-time environments," in *Proceedings of the 1995 IEEE Real-Time Systems Symposium*, Pisa, Italy, Dec. 1995.

[4] John A. Stankovic, Douglas Niehaus, and Krithi Ramamritham, "SpringNet: An architecture for high performance distributed real-time computing," in *Workshop on Parallel and Distributed Real-Time Systems*, Apr. 1993.

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

[6] Marty A. Humphrey and John A. Stankovic, "Multi-level scheduling for flexible manufacturing," Tech. Rep. 95-86, Computer Science Dept., University of Massachusetts, Amherst, MA, Jan. 1995.

[7] Douglas Niehaus, John A. Stankovic, and Krithi Ramamritham, "A real-time system description language," in *Proceedings of the 1995 IEEE Real-Time Technology and Applications Symposium*, May 1995.

[8] Krithi Ramamritham, John A. Stankovic, and Perng-Fei Shiah, "Efficient scheduling algorithms for real-time multiprocessor systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 2, pp. 184–195, Apr. 1990.

[9] Krithi Ramamritham, John A. Stankovic, and Wei Zhao, "Distributed scheduling of tasks with deadlines and resource requirements," *IEEE Transactions on Computers*, vol. 38, no. 8, Aug. 1989.

[10] Wei Zhao and Krithi Ramamritham, "Simple and integrated heuristic algorithms for scheduling tasks with time and resource constraints," *Journal of Systems and Software*, vol. 7, no. 3, pp. 195–205, Sept. 1987.

[11] Jane W.S. Liu, Wei-Kuan Shih, Kewi-Jay Lin, Riccardo Bettati, and Jen-Yao Chung, "Imprecise computations," *Proceedings of the IEEE*, vol. 82, no. 1, pp. 83–94, Jan. 1994.

[12] David Hull, Wu-Chun Feng, and Jane W.S. Liu, "Operating system support for imprecise computation," in *Flexible Computation in Intelligent Systems: Results, Issues, and Opportunities*, Cambridge, MA, Nov. 1996.

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

[14] Karsten Schwan, Hongyi Zhou, and Ahmed Gheith, "Multiprocessor real-time threads," *Operating Systems Review*, vol. 26, no. 1, pp. 54–65, Jan. 1992.

[15] John A. Stankovic, "Misconceptions about real-time computing," *IEEE Computer*, vol. 21, no. 10, pp. 93–108, 1988.

[16] Chia Shen, Krithi Ramamritham, and John A. Stankovic, "Resource reclaiming in multiprocessor real-time systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 4, pp. 382–398, Apr. 1993.

[17] Thomas H. Corman, Charles E. Leiserman, and Ronald L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA., 1990.

[18] Marty Humphrey, *Real-Time Operating Systems: Predictable Threads and Support for Multi-Level Scheduling*, Ph.D. thesis, University of Massachusetts, Amherst MA, 1996.

[19] Technical Committee on Operating Systems and Application Environments of the IEEE, *Portable Operating System Interface (POSIX)—Part 1: System Application Program*

Interface (API), 1996, ANSI/IEEE Std 1003.1, 1995 Edition, including Real Time Extensions (IEEE Std 1003.1b-1993 and Realtime Technical Corrigenda: IEEE Std 1003.1i-1995) and Threads Extensions (IEEE Std 1003.1c-1995).

[20] David R. Butenhof, *Programming with POSIX threads*, Addison-Wesley, Reading, MA, 1997.

[21] Frank Mueller, Viresh Rustagi, and Theodore P. Baker, "Mithos – a real-time microkernel threads operating system," in *Proceedings of the 1995 IEEE Real-Time Systems Symposium*, Pisa, Italy, Dec. 1995.

[22] SunSoft, "Pthreads and solaris threads: A comparison of two user-level thread apis," Sun Microsystems White Paper, May 1994, Early Access Edition (May 1994); pthreads based on POSIX 1003.4a/D8.

[23] John R. Graham, *Solaris 2.x: internals and architecture*, McGraw-Hill, New York, 1995.

[24] Sandeep Khanna, Michael Sebree, and John Zolnowsky, "Realtime scheduling in SunOS 5.0," in *Proceedings of the 1992 Winter USENIX Conference*, Jan. 1992.

[25] Krithi Ramamritham, Chia Shen, Oscar Gonzalez, Subhabrata Sen, and Shreedhar Shirgurkar, "Using Windows NT for real-time applications: Experimental observations and recommendations," in *Proceedings of the Fourth IEEE Real-Time Technology and Applications Symposium*, Denver, CO, June 1998.

[26] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

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

[28] Ahmed Gheith and Karsten Schwan, "Chaos-arc: Kernel support for multi-weight objects, invocations, and atomicity in real-time applications," *ACM Transactions on Computer Systems*, vol. 11, no. 1, pp. 33–72, Apr. 1993.

[29] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos, "First-class user-level threads," in *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, Oct. 1991.

[30] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy, "Scheduler activations: Effective kernel support for the user-level management of parallelism," in *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, Oct. 1991.