Virtual Real-Time Scheduling

Malcolm S. Mollison and James H. Anderson
Department of Computer Science
University of North Carolina at Chapel Hill

Abstract

We propose a new approach for the runtime scheduling of real-time workloads. This approach, which we call virtual scheduling*, decouples real-time scheduling from the underlying real-time operating system (RTOS) kernel. Such a decoupling provides for the use of scheduling algorithms on an RTOS platform that does not support them natively. This allows new scheduling functionality to be offered to industry practitioners without sacrificing the ideal of the stable, predictable, time-tested, and (mostly) bug-free RTOS kernel.*

1. Introduction

In recent years, significant research effort has been expended in the development of novel scheduling algorithms and synchronization protocols for multiprocessor real-time systems. Much attention has also been given to determining whether these techniques perform well on real hardware platforms, despite potential obstacles such as scheduling overheads. Overall, the results are promising: a number of interesting multiprocessor scheduling and synchronization techniques have proven viable. For example, recent work has shown that the *clustered earliest-deadline-first algorithm* (C-EDF) performs well on large multicore machines [3], and an asymptotically optimal locking protocol that can be used with this algorithm has been given [5].

Unfortunately, real-time operating systems (RTOSs) have not kept pace with such developments. Support for any real-time scheduling algorithm or locking protocol developed within the last twenty years[1] is practically nonexistent in both commercial and open-source RTOSs, at least without modification by the end user (which, in general, is impractical). This leaves industry practitioners without a means to make use of recent advances in the state-of-the-art, and thus unable to deploy the more sophisticated embedded and real-time applications that have otherwise been made possible.

Thus, in terms of real-time scheduling and synchronization, there is a large gap between technology (including theory) that exists in academia, and technology that is "consumable" by industry practitioners. This gap is likely only to widen in future years. Even as some advances are incorporated into consumable technology (if they are), new advances will be made by researchers.

We believe that RTOS vendors are strongly disincentivized from incorporating these kinds of advances into existing kernels, which would explain the gap described above. Any changes will alter the timing characteristics of the kernel, and could introduce new sources of timing indeterminism and new bugs. From the perspective of RTOS consumers whose immediate needs are met by existing kernels—that is, most likely, the vast majority of them—making changes to the kernel is a *bad* thing. Over time, RTOS customers may begin to call for new features. However, it is likely that only the most generic features—that is, those that can be used by customers spread across many different sectors of industry—will be incorporated. This runs counter to increasing interest in real-time scheduling for more niche domains, such as adaptive systems and cyber-physical systems.

RTOS consumers are also disincentivized from modifying existing RTOS kernels to achieve more advanced functionality. The complexity of any non-trivial operating system is such that any kernel modifications are hazardous, unless done by someone with specialized expertise in that particular kernel, and with very extensive testing.

Thus, we believe that it is likely that the gap described above will only continue to widen, unless a way around these problems can be found. Specifically, a technique to avoid these problems would allow real-time scheduling and synchronization to be modified or customized totally independently of the RTOS kernel.

Fundamentally, solving these problems means decoupling real-time scheduling and synchronization from the kernel. This implies the creation of a layer of middle-

Work supported by the NC Space Grant College and Fellowship Program; NC Space Grant Consortium; NSF grants CNS 0834270, CNS 0834132, and CNS 1016954; ARO grant W911NF-09-1-0535; AFOSR grant FA9550-09-1-0549; and AFRL grant FA8750-11-1-0033.

1. The stack resource protocol, commonly available in real-time operating systems (albeit under other names), was developed around 1990 [2].

ware software running "atop" the kernel that provides for real-time workloads to be scheduled according to new scheduling paradigms, while making use of existing kernel services, including the existing scheduler provided with a given RTOS kernel. Such a software layer would provide the effective illusion of the presence of a more capable kernel scheduler than actually exists in the system.

This classic pattern—in which a new layer of indirection allows for both better management of complexity and the addition of new features—is well known in software engineering and computer science. One ubiquitous example of this pattern is virtual memory, which provides the illusion to processes of access to a full memory address space and allows for new memory management features. Another example of this pattern is the use of process virtual machines (such as the Java Virtual Machine, or JVM), which gives the illusion of a standard hardware platform when none exists, and also provides features (such as garbage collection) not available natively. Because we draw inspiration from this pattern, and in light of existing terminology for describing it, we call our approach *virtual real-time scheduling*.[2]

The goal of this paper is to explore the viability of the virtual real-time scheduling approach. First, in Section 2, we explore the support for real-time workloads provided by generic, existing RTOS kernels, and cover necessary background material and related work. In Section 3, we show (by constructing an example) that a virtual scheduler supporting a wide variety of real-time scheduling algorithms and synchronization protocols can itself be supported atop a generic POSIX-compliant RTOS. In Section 4, we explain how to modify the virtual scheduler from the previous section to support memory protection (*i.e.*, space partitioning) between tasks. In Section 5, we describe future work. In Section 6, we conclude.

2. Foundations for Virtual Scheduling

In the following subsections, we describe the underlying RTOS functionality that will serve as a foundation for virtual scheduling. Interspersed with this discussion is necessary background material and information about related work.

2.1. OS Support for Multiprocessor Real-Time Scheduling and Synchronization

In this paper, we adopt a widely-used formalism for representing real-time *task systems* known as the *sporadic task model*. Under this model, each task T has an associated *worst-case execution time* (WCET), $T.e$, and *minimum separation time*, $T.p$. Each successive *job* of T is released at least $T.p$ time units after its predecessor, and a job released at time t must complete by its *deadline*, $t + T.p$. The first job of each task is released at an arbitrary time after the release of the task system. The *utilization*, or long-run processor share required by a task, is given by $T.u = T.e/T.p$.

A task system of n tasks is *schedulable* if, given a scheduling algorithm and m processors, the algorithm can schedule tasks in such a way that all its timing constraints are met. For *hard real-time* task systems, jobs must never miss their deadlines, while for *soft real-time* task systems, some deadline misses are tolerable. Specifically, we require here that the tardiness of jobs of soft real-time tasks be bounded by a (reasonably small) constant.

Scheduling algorithms. Approaches to scheduling real-time tasks on multicore systems can be categorized according to two fundamental (but related) dimensions: first, the choice of how tasks are mapped onto processing cores; and second, the choice of how tasks are prioritized. In each case, there are two common choices. Tasks are typically mapped onto cores either by *partitioning*, in which each task is assigned to a core at system design time and never migrates to another core; or by using a *migrating* approach, in which tasks are assigned to cores at runtime (and can be dynamically re-assigned). Tasks are typically prioritized using either *static priorities*, in which case priorities are chosen at design time and never change; or *dynamic priorities*, in which case tasks' priorities relative to one another change at runtime according to some criteria specified by the scheduling algorithm.

In this paper, migrating, dynamic-priority algorithms are of particular interest. An example is the clustered earliest-deadline-first (C-EDF) algorithm, which was mentioned in the Section 1. Under C-EDF, before system runtime, each core is assigned to one of c clusters, where $1 \leq c \leq m$; and each of the n tasks is assigned to one of the clusters.[3] Let d denote the cluster size, *i.e.*, m/c. At system runtime, within each cluster, the d eligible tasks with highest priority are scheduled on the d available cores. (The word "eligible" is used to exclude tasks that are waiting for some shared resource to become available.)

In contrast, almost all RTOSs support only static-priority scheduling. These include VxWorks [12], which is widely considered to be one of the industry leaders in terms of RTOS market share, and Linux, which can be used to run certain real-time workloads. Some existing RTOSs do support dynamic-priority scheduling, such as ERIKA

2. More pragmatically, our terminology serves to distinguish our approach from existing middleware software which supports real-time tasks *without* providing new functionality beyond that provided by the "real" (kernel) scheduler. This practice is detailed in Section 2 under "Related Work."

3. C-EDF partitions tasks, rather than migrating them, if and only if $c = m$.

Enterprise [7]. However, all existing RTOSs essentially limit their users to the particular scheduling algorithm(s) chosen by the RTOS implementers.

In this paper, our concern is to enable migrating, dynamic-priority scheduling algorithms to be run atop kernels that natively only support static-priority scheduling.

Synchronization protocols. A real-time *synchronization protocol* is used to arbitrate among tasks that share resources that cannot be simultaneously accessed by any number of tasks, such as a critical section of code or a shared hardware device. These protocols typically attempt to prevent *priority inversions*, in which lower-priority tasks are allowed to execute in favor of higher-priority tasks due to resource-sharing dependencies. The possibility of priority inversions in a system must be accounted for in schedulability analysis. To the best of our knowledge, no existing commercial RTOS supports any synchronization protocol more recent than the *stack resource protocol*, which was developed for uniprocessor systems around 1990 [2]. Since then, a number of other multiprocessor locking protocols have been developed. These include the *multiprocessor priority-ceiling protocol* (MPCP) [9], the *distributed priority-ceiling protocol* (DPCP) [10], the *flexible multiprocessor locking protocol* (FMLP) [4], and the *O(m) locking protocol* (OMLP) [5]. These protocols are designed for use with migrating, dynamic-priority scheduling algorithms such as those discussed in the previous subsection. The example virtual scheduler described in this paper can support these protocols.

2.2. OS Support for Real-Time Tasks

In this paper, we use the term "task mechanism" to denote a mechanism by which tasks—that is, separate segments of code, runnable independently of one another—can be supported. The range of task mechanisms available on a given system depends upon the underlying CPU architecture, and upon the abstractions made available for programmers by the operating system. These mechanisms are, at least for our purposes, similar across all modern systems. Not all of these mechanisms are accessible to the kernel scheduler, but all of them *are* accessible to the virtual scheduler. Any virtual scheduler implementation must take careful account of how these abstractions are leveraged, because this will have an impact upon both the performance characteristics of the virtual scheduler and the features it may offer.

Task mechanisms commonly available in modern computer systems are given in the list below. Note that terminology used to describe these mechanisms is far from unified; we attempt to use terminology that is specific enough to avoid and potentially confusing overlap with existing usage.

- A *function*. Tasks implemented using functions share the same stack. This prohibits arbitrarily preempting and switching between tasks. In particular, tasks must run in a nested order in order to preserve the stack. This seriously limits the scheduling algorithms and synchronization protocols that can be supported under this task mechanism.
- A processor *context*.[4] Each context has its own stack, allowing tasks to be preempted and switched among in an arbitrary order. A single context can persist over multiple function calls. In C, contexts can be stored and recalled in userspace using the **setjmp()** and **longjmp()** functions.
- A *kernel-level thread*. A kernel-level thread is defined to be an entity that is known to the operating system and schedulable by the kernel scheduler. The relationship between a kernel-level thread and a context is typically one-to-one, but could be one-to-many, or many-to-many; in other words, contexts can be shared between kernel-level threads. A kernel thread may or may not have memory protection from other kernel threads in the system.
- A *process*. A process is defined to be a group of one or more kernel-level threads that have memory protection from all kernel-level threads that are not in the group. The relationship between a process and a kernel-level thread is typically one-to-one, but could be one-to-many. In many older operating systems, the distinction between kernel-level threads and processes did not exist; all kernel-schedulable entities had independent memory protection. In such cases, the term "process" was generally preferred over other terms.

Making Use of Task Mechanisms. A *user threading library* implements tasks using *user-level threads*, *i.e.*, threads which the kernel is not directly aware of or able to schedule (in contrast to kernel-level threads). User threading libraries typically implement user-level threads by means of contexts. There exist many such libraries; GNU Portable Threads ("Pth") [8] is one example. User-level threading libraries support concurrency (*i.e.*, multiple tasks whose execution is interleaved); however, they do not support parallelism, because they are assumed to execute from within only a single kernel thread.

POSIX Threads ("Pthreads") is a POSIX standard that defines an API for creating and managing threads; a POSIX-compliant threading implementation can make use of either user-level threads or kernel-level threads. Most generic RTOSs support Pthreads, possibly alongside a

4. This term is equivalent, for our purposes, to the term *continuation*, which arises in the study of programming languages and denotes an abstract representation of the processor context that can be stored and recalled by the programmer.

vendor-defined API; these implementations, in turn, rely on kernel-level threads (with or without memory protection), which are then scheduled to run according to a static-priority scheduling algorithm, as discussed previously.

The virtual scheduler implementation proposed in this paper uses both user-level threads *and* kernel-level threads. This approach, known as *hybrid threading* (as opposed to "user threading" or "kernel threading"), has been used before in the parallel processing community. The highly concurrent, non-real-time Erlang programming language uses this approach [6]; however, Erlang does not offer enough fine-grained control of tasks to support our needs. On the other hand, a hybrid scheduling approach known as *scheduler activations* [1]—like Erlang, originally devised for large-scale parallel processing—does support many features that are desirable for virtual scheduling. However, making use of scheduler activations requires specialized kernel support. To the best of our knowledge, no RTOS offers support for scheduler activations.[5]

2.3. Related Work

Virtual scheduling is not the only approach that proposes to enhance practical real-time scheduling capabilities through software that runs in userspace. Other approaches include the following.

- In industry, *operating system abstraction layers* (OSALs) are occasionally used to provide portability for applications between different RTOSs.
- The *Real-Time Ada* programming language provides for a userspace runtime library that is intended to simplify and standardize the development of real-time applications, and particularly, to improve their portability across different operating system platforms.
- The *Real-Time Java* programming language is similar (for our purposes), though it is implemented using a process virtual machine (a specialized JVM) instead of a runtime library.

Real-Time Ada and Real-Time Java do support certain kinds of processor synchronization. However, to the best of our knowledge, they currently only support the same scheduling services offered by the underlying kernel scheduler—*i.e.*, fixed-priority scheduling.[6] Because these two languages rely on userspace implementations, enhancing the facilities they currently offer with the addition of a virtual scheduler (*i.e.*, a scheduler that supports services *not* present in the underlying scheduler) would be a natural fit.

5. Support for scheduler activations is present in certain versions of the Solaris operating system, and (surprisingly) in Windows 7.

6. There has been some discussion of extending Ada to support earliest-deadline-first scheduling [11]. This is comparable to the overall topic of this paper, but has a different goal.

3. Constructing a Virtual Scheduler

In this section, we first determine the functionality that should be supported by our virtual scheduler. We then describe our proposed implementation. In describing this implementation, our intention is, first, two show that a generic, POSIX-compliant RTOS offers sufficient features to virtualize a broad variety of scheduling algorithms; and, second, to provide a starting point for investigating more refined virtual scheduler implementations. Thus, we attempt to avoid trivial optimizations in favor of a more generic and easily-understood model.

3.1. Functional Requirements

Earlier, we classified scheduling algorithms according to whether the priorities of tasks can change, and whether tasks can migrate. In effect, this creates four classes of algorithms, which (for convenience) we label as follows.

(P-SP) partitioned, static-priority algorithms
(P-DP) partitioned, dynamic-priority algorithms
(M-SP) migrating, static-priority algorithms
(M-DP) migrating, dynamic-priority algorithms

Any scheduler (either virtual or kernel-based) that supports arbitrary P-DP algorithms supports arbitrary P-SP algorithms. Any scheduler that supports arbitrary M-DP algorithms supports arbitrary M-SP algorithms. Finally, any scheduler that supports arbitrary M-DP algorithms supports arbitrary P-DP algorithms. Thus, any scheduler that can support arbitrary M-DP algorithms can support all four classes. Furthermore, in each of the classes, it is possible to define algorithms that support either preemptive or non-preemptive scheduling; preemptive scheduling is more general. To the best of our knowledge, these classifications cover all of the multiprocessor scheduling algorithms for sporadic real-time task systems that are currently of interest in the research community.

The virtual scheduler proposed in this paper supports one scheduling algorithm from the preemptive M-DP class. To aid in understanding of the proposed implementation, we did not attempt to support arbitrary algorithms from the preemptive M-DP class, which would (in turn) allow any scheduling algorithm from any of the given classes to be supported. However, we conjecture that extending our specific implementation to support arbitrary M-DP algorithms is possible. Thus, we hope to convince the reader that our proposed virtual scheduler can easily be extended to support all scheduling algorithms of interest.

In contrast to the algorithmic flexibility suggested by the virtual scheduler implementation proposed in this section, our proposed implementation supports only partial memory protection between tasks. Memory protection is a relatively new feature for some widely-used RTOSs; for example, the

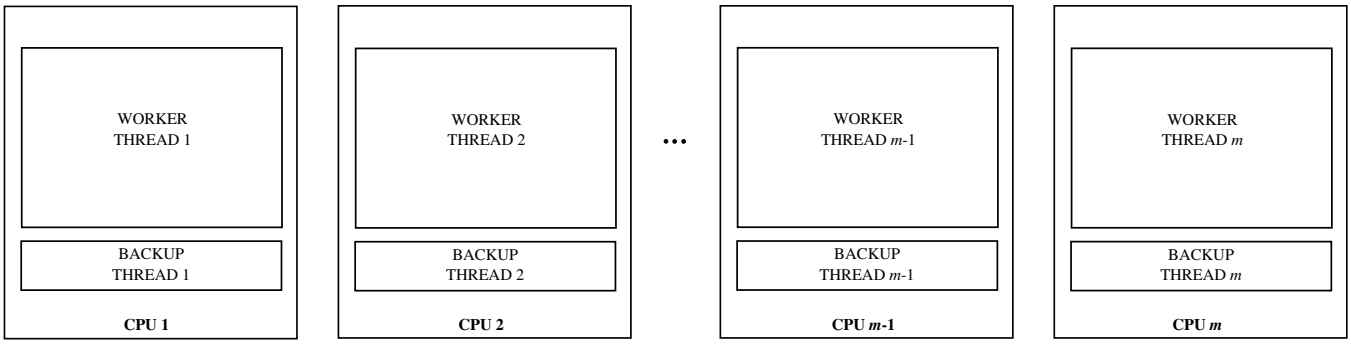

Figure 1: Assignment of kernel-level threads to CPUs.

first edition of VxWorks to support memory protection was the 6.0 series, which was released in 2004. Nonetheless, memory protection can be expected to play a larger role in real-time systems over time, as these systems grow more complex. These issues are explored in depth in Section 4, where we describe how our proposed implementation could be altered to support full memory protection between tasks, at the expense of increased runtime overhead.

3.2. Proposed Implementation

Our approach uses hybrid scheduling—that is, a mixture of user-level threads and kernel-level threads. For ease of explanation, the implementation provided here only supports the *global earliest-deadline-first* (G-EDF) algorithm. G-EDF is equivalent to C-EDF with cluster size m. As mentioned previously, we conjecture that our implementation can be extended to support arbitrary scheduling algorithms in the M-DP class.

Task mechanism. Our implementation supports real-time tasks as contexts. This choice is preferable over more heavy-weight task mechanisms because user-level context switching is generally considered to have at least an order-of-magnitude speed advantage over switching between kernel threads [1]. On the other hand, a more light-weight mechanism—supporting tasks in a manner that maps multiple tasks to one context—would not allow for switching between tasks in arbitrary order, as explained in Section 2. This rules out using mapping the n tasks to fewer than n separate contexts. Nonetheless, each task *is* encapsulated inside a function call; when this function call is executed, a job of the task begins executing, and when a job completes, the function returns. Our implementation mechanisms will ensure that each task receives its own context. These mechanisms are implemented as a runtime library that merely needs to be included with the tasks' code as a header file.

Task initialization. A special *initialization thread* is responsible for setting up all state needed by the virtual scheduler for the real-time workload to begin executing. This includes initializing the real-time tasks. To initialize a task, the initialization thread calls the **setjmp()** function immediately before a conditional call to the task's function, storing the result—the newly-initialized context—in a variable. The conditional call, starting the execution of the task, only executes when **setjmp()** returns after a context switch—hence, *not* during the initialization phase.

Data structure initialization. After initializing the tasks, the initialization thread creates the following key data structures.

- The release queue, a priority queue that holds the contexts for tasks that have no currently eligible job, but will experience a job release at a known time in the future.
- The ready queue, a priority queue that holds the contexts for tasks that are currently eligible, but not running.
- The lowest-priority indicator. During runtime, this variable indicates the CPU of the lowest-priority currently-running task. This will allow for the proper task to be interrupted when a new task that has sufficient priority to run becomes eligible. Ties are broken arbitrarily.

Additional data structures created by the initialization thread are described later in the paper, when they can be understood more easily.

Kernel-level thread initialization. Finally, the initialization thread creates a number of kernel-level threads. This includes n *worker threads* and m *backup threads*. Each of the m cores has exactly one of the worker threads statically assigned to it; these are called *active* worker threads. The non-active worker threads and the backup threads are only relevant in a specialized case, which is explained later. Until then, they can be safely ignored. See Figure 1 for a

diagram showing the described kernel threads.

Overview of runtime scheduling. The active worker threads perform the role of switching between tasks and executing them. Under our G-EDF example, at any time, the m highest-priority eligible tasks are executed by the m active worker threads; if there are less than m eligible tasks, then one or more of the active worker threads is idle.

Task switching. An active worker thread can switch tasks by saving the current task context using **setjmp()** and appending it to a queue (or other shared data structure) where it can be retrieved later, and executing **longjmp()** on a task context that it has obtained from a queue (or other shared data structure) and is to begin executing. Whenever task switching occurs, the lowest-priority indicator variable is updated.

In summary, the basic technique used by the virtual scheduler is to execute the n tasks on m active worker threads, switching execution between tasks as appropriate using **setjmp()** and **longjmp()**. Now, we can move on to an explanation of how job completions, job releases, and task synchronization are handled.

Job completions. When a job completes, the active worker thread executing the task switches to a new task drawn from the ready queue (if any is available). If the completed task is released periodically—*i.e.*, at a known offset from its previous release— it is added to either the release queue or ready queue, depending on whether the release time of the next job has already been reached. Note that adding a task to the release queue implies updating the release timer under certain circumstances, as explained below. On the other hand, if the task is released in response to some stimulus, it is instead saved in a data structure reserved for tasks of this kind.

Responding to asynchronous stimuli. Tasks that are released in response to external stimuli can be accommodated easily. The virtual scheduler implementation merely needs to define a signal handler for signals indicating that a task of this kind needs to be added to the ready queue. Such a signal could originate from a task that receives network packets, for example, or directly from a device driver.

Aside: POSIX timers and signals. The rest of our explanation requires a basic understanding of POSIX timers and signals. A POSIX timer is armed with an associated expiry time and notification thread. (The notification thread must be one created using the POSIX API). When the expiry time is reached, the timer fires, sending a POSIX timer-expired signal to the notification thread.

Maintaining the release queue timer. Throughout task system execution, a POSIX timer is used to mark the time of the next job release in the system. In our virtual scheduler, this timer is set (or reset) whenever a task is added to the release queue, if no task with an earlier release time exists in the queue. The low-priority indicator variable is used as a basis for selecting the proper notification thread (*i.e.*, one of the worker threads).

Responding to a timer expiry signal. When a worker thread receives the timer expiry signal, its execution jumps to an associated signal handler (provided in our virtual scheduler's runtime library). In this signal handler, the worker thread moves the next-to-be-released task to the ready queue, and updates the release queue timer. If the moved task is of sufficiently high priority, some worker thread will begin executing it. This could be the same worker thread that just released it; otherwise, a signal is sent to the appropriate worker thread by the thread that released it. The appropriate thread is determined by the current status of the lowest-priority indicator variable.

Synchronization protocols. The virtual scheduler runtime library must provide an API that can be used by a task to request access to a shared resource. If the task is to be allowed to acquire the resource without waiting, the API returns immediately. Otherwise, the task's context is saved and stored in a data structure used specifically to hold tasks blocked in such a case. When a task is finished with the resource, it must perform another API call to release it. At this point, any task held in the data structure referenced above that is now eligible (*i.e.*, can obtain the resource) is moved onto the ready queue. If the newly-unblocked task has sufficient priority, a signal is then sent to the appropriate worker thread (as indicated via the lowest-priority indicator variable) so that the task will be removed from the ready queue and executed.

Blocking in the kernel. We have laid most of the essential groundwork for our virtual scheduler. Only one key issue remains: What if a worker thread blocks while running in kernelspace? Such a scenario could arise due to a system call or a page fault. We solve this problem using the backup threads that were mentioned earlier. Recall that there are exactly m backup threads, each affixed to one of the m processors. The backup threads are assigned a lower priority than the active worker threads; thus, a backup thread only runs on a core if the worker thread on that core blocks in the kernel. In such a case, the backup thread causes one of the non-active workers to become the new active worker thread on that core. (The non-active workers are kept at a priority below the backup threads, so that none of them will execute until it has been selected to become an active worker, has been migrated to the proper core, and has had its priority adjusted.) Just before adjusting the

priority of the new active worker and giving up its control of the processor, the backup thread boosts the priority of the blocked worker thread to be above that of all worker threads, and sends it a signal to indicate that it has blocked in the kernel. As soon as the blocked worker finishes blocking in the kernel, it will receive the signal indicating that it has blocked. This worker adds its task to the release queue, sends a signal to the worker thread indicated by the lowest-priority indicator variable (to potentially trigger a context switch to the task that blocked in the kernel), and becomes a non-active worker. Note that this process happens immediately when the task finishes blocking in the kernel, since the priority of the blocking thread has been boosted above that of the active worker threads.

4. Supporting Memory Protection

One hazard of any multitasking computer system is the possibility of one task corrupting the memory of another task. As real-time systems have become more complex, this problem has become a growing concern. In this section, we discuss the degree of memory protection provided by the virtual scheduler implementation described in Section 3, and also explain how full memory protection between tasks can be provided, at the expense of additional runtime overhead.

Note that, in this paper, we are *not* concerned with the difficult problem of preventing code created with malicious intent from damaging the system, which is typically only a concern in a narrow segment of highly-critical real-time systems (such as military weapon systems and nuclear power plants). Rather, we are interested in preventing widespread failures caused by programmer error and in the absence of attacks by adversaries.

4.1. Existing Properties

The implementation given in Section 3 can be made relatively robust to these kinds of errors with just a little bit of effort. Memory protection for kernel-level threads is now a common feature, even in RTOS kernels. If each kernel-level worker thread has memory protection from all other threads (*i.e.*, is treated as a process), the only possibility for one task to corrupt another is by corrupting the data structures shared between worker threads (such as the release queue and the ready queue).

Furthermore, if this kind of corruption does not occur, the implementation is already robust to process failures (for example, segfaults). Under such a failure, the worker thread and the real-time task being executed inside it would be lost; the backup thread running on that core would then run and activate a non-active worker thread in its place, in accordance with the specification given in Section 3.

4.2. Achieving Complete Memory Protection

Nonetheless, the implementation given in Section 3 leaves critical shared scheduling data structures vulnerable to corruption. If one of these data structures were to become corrupted, it could cause the failure of all tasks in the system. The only way around this problem is to prevent any task application code from being able to write to these shared scheduling data structures. (We assume that the virtual scheduler runtime library code is trusted not to cause corruption; ultimately, some code in the system must be trusted to update scheduling data structures.) Below, we outline how the implementation from Section 3 can be modified to achieve this property. (There are many specific implementation tradeoffs that may be worth investigating in future work. Here, our concern is simply to show that our virtual scheduler mechanism can be modified to enable memory protection.)

Modifications for full memory protection. Rather than using contexts as the task mechanism, kernel-level threads are used. When the system is initialized, n such threads are created, all with a priority lower than that of the backup threads. In this scheme, the backup threads play a more important role than before. The backup threads share access to scheduling data structures. They select tasks to run on each core, and cause them to do so by forcing them to migrate to the appropriate core and setting their priority to be higher than that of the backup threads. Task completion is carried out when a task sets its priority back to the lower setting, returning control to the backup thread on that core. Timer-driven signals for releasing tasks are set up in a way that causes them to be delivered to the task running on the relevant core. When a task receives such a signal, it returns control to the backup thread on that core by lowering its priority.

5. Future Work

Initially, we would like to make a more elaborate examination of the likelihood of the virtual scheduling approach to yield useful results. Such an examination would most likely focus on measuring the performance of several virtual scheduler implementations and comparing their performance to that achieved by native RTOS schedulers.

Given the success of such an examination, we would ultimately like to conduct a more comprehensive study of virtual schedulers. Such a study would first catalog relevant classes of real-time systems, as distinguished by characteristics like hardware platform (including number of cores and caching hierarchy); workload properties (such

as the number of tasks and the maximum per-task utilization); performance requirements (such as types of timing constraints or requirements for adaptivity); and robustness requirements (such as fault isolation and security protection). Then, the study would determine, as best as possible, the most effective virtual scheduling implementation for each class, in terms of characteristics like runtime overhead. Such a study would allow industry practitioners to choose intelligently between native RTOS scheduling and virtual scheduling for applications of interest, and would (hopefully) open up avenues for entirely new classes of real-time systems to be deployed.

In this study, we do not wish to rule out implementation approaches necessitating basic RTOS kernel modifications, in return for a significant boost in the capabilities of the virtual scheduler. Such approaches would be of interest if the needed modifications were generic and straightforward enough that one could reasonably hope for commercial RTOS vendors to eventually adopt them.

If the virtual scheduling approach proves viable, we are eager to apply it to novel topics currently being studied by the real-time systems research community, such as adaptive systems, mixed-criticality systems, hierarchical scheduling systems, and secure real-time systems. We believe virtual scheduling may enable significant practical advances in these areas.

6. Conclusion

In this paper, we proposed a new approach for the runtime scheduling of real-time workloads, *virtual scheduling*, which decouples real-time scheduling from the underlying real-time operating system (RTOS) kernel. This decoupling provides for the use of scheduling algorithms on an RTOS platform that does not support them natively. If it proves to be viable, this approach will allow new scheduling functionality to be offered to industry practitioners without sacrificing the ideal of the stable, predictable, time-tested, and (mostly) bug-free RTOS kernel. However, the best way to go about exploring this concept is far from obvious. We are eager to receive feedback from real-time kernel developers and academic researchers on how to make the most of this approach, considering the intricacies of real-time operating systems and the complexity of real-time scheduling algorithms.

References

[1] T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Scheduler activations: Effective kernel support for user-level management of parallelism. In *ACM Transactions on Computer Systems, vol. 10, no. 1*, pages 53–79, 1992.

[2] T. P. Baker. Stack-based scheduling of real-time processes. *The Journal Of Real-Time Systems*, 3(1):67–99, 1991.

[3] A. Bastoni, B. Brandenburg, and J. Anderson. An empirical comparison of global, partitioned, and clustered multiprocessor edf schedulers. In *Proceedings of the 2010 31st IEEE Real-Time Systems Symposium*, pages 14–24, 2010.

[4] A. Block, B. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *Proceedings of the 13th IEEE Conference on Embedded and Real-Time Computing Systems and Applications*, pages 47–57, 2007.

[5] B. Brandenburg and J. Anderson. Optimality results for multiprocessor real-time locking. In *Proceedings of the 2010 31st IEEE Real-Time Systems Symposium*, pages 49–60, 2010.

[6] Erlang Web site. http://www.erlang.org/.

[7] Evidence Web site. http://www.evidence.eu.com.

[8] GNU Portable Threads Web site. http://www.gnu.org/software/pth.

[9] R. Rajkumar. Real-time synchronization protocols for shared-memory multiprocessors. In *Proceedings of the 10th International Conferece on Distributed Computing Systems*, pages 116–123, 1990.

[10] R. Rajkumar, L. Sha, and J. Lehockzy. Real-time synchronization protocols for multiprocessors. In *Proceedings of the 9th Real-Time Systems Symposium*, pages 259–269, 1988.

[11] A. Wellings and A. Burns. Generalizing the edf scheduling support in ada 2005. In *Ada Letters, vol. 30*, pages 116–124, 2010.

[12] Wind River Web site. http://www.windriver.com.