A Real-Time Scheduling Service for Parallel Tasks

David Ferry, Jing Li, Mahesh Mahadevan, Kunal Agrawal, Christopher Gill, and Chenyang Lu
Department of Computer Science and Engineering
Washington University in St. Louis
{dferry, li.jing}@go.wustl.edu, mmahadevan@wustl.edu, {kunal, cdgill, lu}@cse.wustl.edu

***Abstract*—The multi-core revolution presents both opportunities and challenges for real-time systems. Parallel computing can yield significant speedup for individual tasks (enabling shorter deadlines, or more computation within the same deadline), but unless managed carefully may add complexity and overhead that could potentially wreck real-time performance. There is little experience to date with the design and implementation of real-time systems that allow parallel tasks, yet the state of the art cannot progress without the construction of such systems. In this work we describe the design and implementation of a scheduler and runtime dispatcher for a new concurrency platform, *RT-OpenMP*, whose goal is the execution of real-time workloads with intra-task parallelism.**

I. Introduction

In the last 15 years, the increase in processor clock speeds has slowed, but this has been replaced by significant increases in the number of processing cores per chip. Sequential program performance no longer improves with newer processors, so real-time application developers either must content themselves with stagnating execution speeds, or must be willing to take the plunge into multi-core and parallel programming. Such a shift requires extensions throughout real-time systems, from theoretical foundations to the design and implementation of real-time software.

For many years, real-time systems researchers have developed models, theory, and software to support ***inter-task parallelism***, where workloads consist of a collection of independent sequential tasks and increasing the number of processors or cores allows us to execute more such tasks at once. While these systems allow many tasks to execute on the same multi-core host, they do not allow an individual task to run any faster on a multi-core machine than on a single-core one. We call this form of real-time processing ***multiprocessing***.

In this paper we concentrate on ***parallel processing*** systems, where real-time tasks can have ***intra-task parallelism*** in addition to inter-task parallelism. In these systems, workloads consist of a collection of independent parallel tasks, but each individual parallel real-time task is allowed to execute on multiple (potentially overlapping) cores. This capability allows parallel processing systems to execute a strictly larger class of programs than multiprocessing systems. In particular, when the opportunity for parallel execution exists, it allows for the execution of individual tasks with tighter timing constraints or higher computational loads within a given constraint. This can lead to improved execution of computation-heavy real-time systems such as those for video surveillance, computer vision, radar tracking, and hybrid real-time structural testing [**?**], whose stringent timing constraints can be difficult to meet through traditional multiprocessing. Many of these applications are highly parallelizable, and supporting intra-task parallelism can allow real-time systems to run more demanding programs.

In this paper, we describe a prototype scheduling service for ***RT-OpenMP***, a *real-time concurrency platform* we are developing based on OpenMP [**?**], which supports real-time semantics, provides a true parallel programming interface, performs automatic scheduling of parallel tasks [**?**], and is based on theoretical results [**?**] in parallel real-time scheduling. As far as we are aware, this is the first platform for the execution of real-time parallel tasks that provides automatic scheduling with respect to a theoretical schedulability bound.

The *RT-OpenMP* concurrency platform is designed to provide tools and high-level abstractions for parallel real-time execution. In this paper, we present the following contributions:

- We present the design and implementation of a scheduler and runtime dispatcher capable of automatically scheduling and executing a collection of parallel real-time tasks that conform to the parallel synchronous task model introduced in [**?**].
- We use a set of synthetic workloads to evaluate the performance of the platform under various partitioned scheduling strategies and utilizations. We show that the platform provides good performance for a significant class of potential applications.

In Section **??** we describe the parallel synchronous task model. Section **??** provides background information about OpenMP. In Section **??** we describe the design of our scheduler and dispatcher. In Section **??** we present an empirical evaluation of our scheduling service through full system tests with synthetic parallel tasks, as well as through micro-benchmarks. We describe related work in Section **??** and conclude in Section **??** with plans for extending this work.

II. Parallel Synchronous Task Model

In this paper, we focus on ***synchronous tasks*** — tasks described by a sequence of segments where each segment consists of one or more parallel ***strands***[1] of execution, as

[1]The nomenclature we use here is purposely different from the preferred term in [**?**], *threads*. In the context of our scheduling service it reduces confusion to use the term *strands* to refer to fundamental units of executable code and reserve the term *threads* for the operating system's *persistent threads* that are responsible for executing those units.

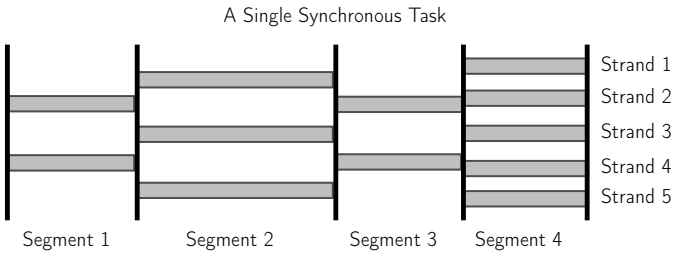


Fig. 1: A parallel-synchronous task with four segments.

shown in Figure **??**. The end of each segment serves as a synchronization point: no strand from the next segment may begin executing before all strands from the current segment have completed. The deadline of a synchronous task is the time by which all strands of the last segment must finish executing.

This is not the most general model for describing parallel programs. We use this model for two reasons. First, recent work [**?**] allows us to provide schedulability assurances. Second, the high-level **`parallel for`** programming construct naturally maps to this particular task model, which as future work will allow *RT-OpenMP* to support this very common parallel programming idiom. This construct is of primary importance for many parallel programs as it very nicely captures the SIMD (single instruction multiple data) paradigm of many parallel applications.

To describe the model more formally, a task set τ consists of n parallel synchronous tasks $\{\tau_1, \tau_2, \cdots, \tau_n\}$. Each task τ_i is a sequence of k_i segments, and a segment may not execute until the previous segment is entirely finished. The j^{th} segment in the i^{th} task τ_i is denoted $\langle e_{i,j}, m_{i,j} \rangle$, where $e_{i,j}$ is the worst-case execution requirement of all strands in segment j and $m_{i,j}$ is the total number of strands. Since there is a synchronization point at the end of each segment a task τ_i can be alternately described as the sequence $(\langle e_{i,1}, m_{i,1} \rangle, \langle e_{i,2}, m_{i,2} \rangle, \cdots, \langle e_{i,k_i}, m_{i,k_i} \rangle)$, where k_i is the number of segments of task i. We assume sporadic implicit deadline tasks with the deadline D_i of each task equal to its period T_i. Later, for the purpose of scheduling, we will refer to the release time $r_{i,j}$ and deadline $d_{i,j}$ of a strand, which are the times by which a strand may start and must finish execution in order to assure that the overall task deadline D_i is met.

Other Definitions: Intrinsic to each task are several quantities of practical importance. The worst case execution time of a task τ_i on a single processor, called its ***work***, is denoted by C_i. The task's execution time on an infinite number of cores, called its ***critical-path length***, is denoted by P_i. By definition, the worst case execution time is $C_i = \sum_{j=1}^{k_i} m_{i,j} \cdot e_{i,j}$ and the critical path length is $P_i = \sum_{j=1}^{k_i} e_{i,j}$. Intuitively, the work is the total amount of computation in a task (all strands from all segments), while the critical-path is the longest chain of sequential computation (the longest strand from each segment). The ***utilization*** $U_i = C_i/T_i$ of a task τ_i is the ratio of total work to the task period, while the utilization of a task set is simply the sum of the utilization of each task in the set. Note that, unlike sequential tasks, it is possible for a parallel task to have utilization greater than 1.

The ***augmentation bound*** is a property of a scheduling algorithm which provides a schedulability test. In our case where all processors execute at the same speed (as in many common multiprocessor systems), an augmentation bound of b implies that under the given scheduling algorithm a system with p processors can execute any task set with total utilization equal to or less than p/b. This is a sufficient but not necessary test, meaning that task sets with greater utilization may still be schedulable.

III. Overview of OpenMP

OpenMP is an Application Programming Interface (API) specification that defines a standardized model for parallel programming on shared-memory multiprocessors. The specification is governed by the OpenMP Architecture Review Board (ARB), which is primarily composed of representatives from companies engaged in the production of hardware and software used in parallel and high-performance computing. The OpenMP API [**?**] is defined for the languages C, C++, and FORTRAN, and has been implemented on many different architectures and major compilers. Importantly, for the purposes of our research, there exists an open source version within the GNU Compiler Collection (GCC).

OpenMP provides programming support through library routines and directives. Library routines include auxiliary functions that allow a program to query and modify OpenMP runtime parameters (such as the number of threads or thread scheduling policy), as well as locking and timing routines. OpenMP directives are compiler pragma statements that indicate where and how parallelization can occur within a program. For example, one such directive converts a regular **`for`** loop to a **`parallel-for`** loop, by prefacing the loop with **`#pragma omp parallel for`**.

However, the current OpenMP implementation does not support real-time execution. First, the specification lacks any notion of real-time deadline and period semantics. More fundamentally, the current OpenMP platforms, and particularly their schedulers, are ill-suited for real-time performance. When invoking a parallel directive in OpenMP there is no expectation of how, where, and when parallel execution will take place. These directives merely point out the available program parallelism, and the compiler and runtime system make very few guarantees about how the program actually executes. For general parallel execution this may be desirable, as it allows the system to load-balance flexibly and effectively, allowing OpenMP to run correctly on a sequential machine or on different parallel machines with varying numbers of processors. Unfortunately, such flexibility is not good for real-time computing, where correctness is also a function of execution time. Thus, it is necessary for a real-time concurrency platform to provide the stronger assurance that the deadlines of a given parallel workload are feasible on a specific execution target. It is also necessary that the runtime system supports such assurances through robust deadline-aware scheduling and dispatching.

IV. Real-Time Scheduling Service Design

The role of our scheduling service for *RT-OpenMP* is to schedule parallel-synchronous applications while providing real-time assurances to application developers. There are two objectives. First, *RT-OpenMP* must ensure that dependences between segments are respected. Second, it must execute tasks so that they meet their deadlines.

We use two systems to enforce this behavior, a ***scheduler*** and a ***dispatcher***. The scheduler decomposes and annotates tasks prior to execution time, and the dispatcher uses that information to dispatch strands of execution at runtime. In our current system, the scheduling phase occurs before execution begins and the dispatching phase occurs at runtime.

A. Scheduler

Our scheduler consists of two components: a ***decomposition algorithm*** (from [**?**]) that decomposes a parallel task into a set of sequential strands, each with its own release time and deadline; and a ***priority assignment and partitioning algorithm*** (from [**?**]) that sets priorities for each sequential strand and assigns each of them to a particular core given a p-core processor. The theoretical result from [**?**] provides an augmentation bound of 5 for this method. This yields the following schedulability test: if the total utilization of a task set is $p/5$ (20% of the maximum utilization allowed) and the critical path length P_i of each task is at most $1/5$th of its deadline T_i, then the theory guarantees this task set as schedulable.

Decomposition Algorithm: The decomposition in [**?**] performs the following operations. First, each task in the task set is decomposed into a set of independent strands, where each strand has its own individual release time and deadline. These strands are analyzed collectively, and the total computational time is divided in a way that provides enough capacity for each. This assignment is reflected in an intermediate release time and deadline for each individual strand. Release times are also chosen to satisfy dependences between segments (the actual mechanism used to enforce this in the system is barrier synchronization, but the dependency timing constraints are used to ensure the correctness of the decomposition).

For this work, we must perform the following adaptation: the above decomposition provides an augmentation bound of 5: this means that if an ideal scheduler can schedule a task set on m cores of speed 1, then a decomposed task set can be scheduled on m cores of speed 5. Due to the derivation in [**?**], it is important that the parameters T_i, C_i and P_i, are measured on an ideal unit-speed machine and the decomposition is done at speed 2. In this formulation both the speed-1 and speed-2 machines are hypothetical, while the task set actually runs on what are considered speed-5 processors. Therefore, we must compute the decomposition for machines that are 2.5 times *slower* than our machines, since decomposition occurs at speed-2.

Hence, this gives rise to a constant value of 2.5 in the following equations. In practice we measure C_i, T_i, and P_i on real machines and then multiply those quantities by 2.5 to simulate a 2-speed machine. The following process of decomposition is otherwise exactly the same as in [**?**], but modified to reflect the inversion we have described.

The decomposition works as follows. The total slack of a task is the extra time it has to finish computation, if it was given an infinite number of processors as soon as it was released. Then the slack on the (hypothetical) speed-2 processors is denoted as

$$L_i = T_i - 2.5P_i$$

The idea behind decomposition is to divide this total slack among all the segments "equitably." For this purpose, we classify segments into ***heavy*** and ***light*** segments. Intuitively, heavy segments are those with many strands, and therefore, a larger computational requirement. Our classification is based on a threshold: a segment is classified as heavy if the number of strands in the segment is more than the total computational requirement divided by the slack (on the 2-speed processor), that is,

$$m_{i,j} > \frac{2.5C_i}{T_i - 2.5P_i}$$

otherwise it is a light segment. The total slack is distributed among heavy segments, giving them more time to finish and therefore reducing the maximum workload density of the task.

If there are any heavy segments in a task, we compute their segment slack as

$$l^h_{i,j} = \frac{m_{i,j}(T_i - 2.5P^\ell_i)}{2.5(C_i - C^\ell_i)} - 1$$

where P^ℓ_i is the portion of the critical path contributed by light segments and C^ℓ_i is the portion of the worst case execution time contributed by light segments. In the case where heavy segments exist, no slack will be given to light segments: that is, $l^\ell_{i,j} = 0$. The relative deadline for all strands in a segment is

$$d_{i,j} = 2.5e_{i,j}(1 + l_{i,j})$$

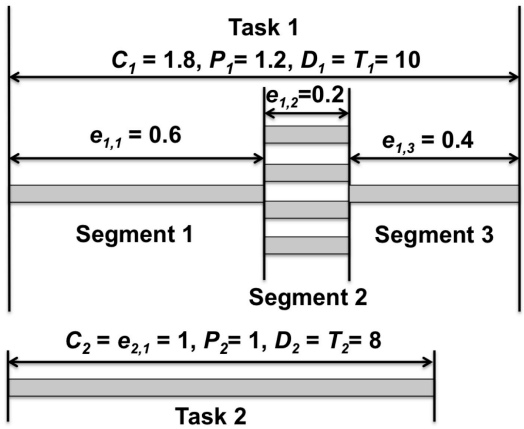
Note that even when light segments are not given any slack on (the hypothetical) speed-2 processors, when we run them on the (real) speed-5 processors, they do have slack.

If all segments are light, each segment will receive an equal portion of the slack and the relative deadline of all strands in a segment is

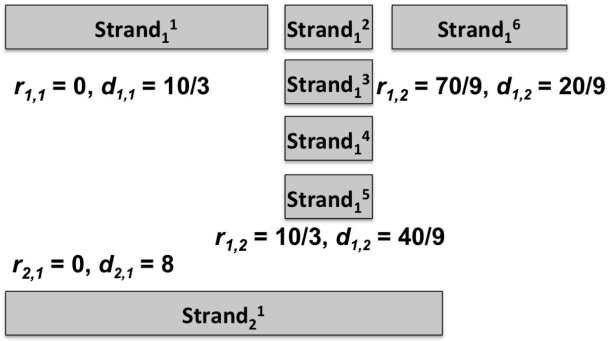
$$d_{i,j} = \frac{e_{i,j}T_i}{P_i}$$

In either case, the release time of each strand is the deadline of the preceding segment.

We give an example to show the action of our scheduler in Figure **??**. The example task set has two synchronous tasks whose parameters are shown in Figure **??** and are executed on a machine with three cores. In task 1, all segments are calculated as heavy segments, since $m_{1,j}$ should be larger than



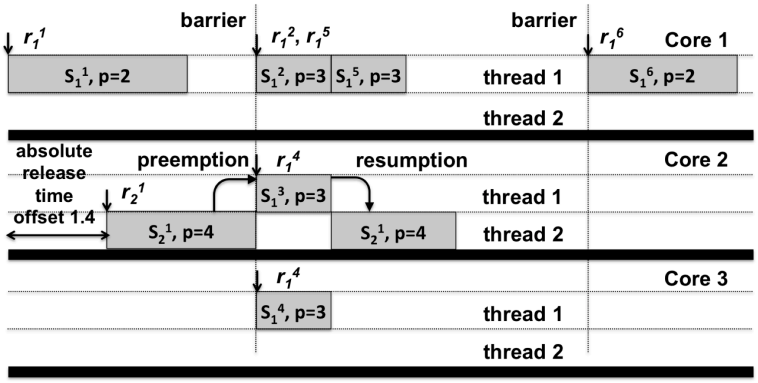
(a) Task set consists of two tasks.



(b) The decomposed tasks on 3 processors. Each strand has its own release time and deadline.

strand	deadline	priority	core
s_1^1	10/3	2	1
s_1^2	40/9	3	1
s_1^3	40/9	3	2
s_1^4	40/9	3	3
s_1^5	40/9	3	1
s_1^6	20/9	1	1
s_2^1	8	4	2

(c) Priority Assignment and Partitioning



(d) Execution trace of the strands execution

Fig. 2: An example of the execution trace provided by our scheduling service

0.643. Thus, segment 1, 2 and 3 get extra slack of 1.22, 7.88 and 1.22 respectively and hence have relative deadline of 10/3, 40/9 and 20/9 respectively. Similarly, the only segment of task 2 is a heavy segment and gets all the slack of 3.2 time units. So the deadline for segment 1 is $3.2*2.5*1=8$, the same as task 2's deadline. Figure **??** shows the decomposed sequential strands with individual release times and deadlines.

Partitioning and Priority Assignment: As indicated by [**?**], we use FBB-FFD [**?**] (Fischer Baruah Baker First-Fit Decreasing bin packing) to assign strands to cores.

First, strands are sorted according to their relative deadlines. Since we are using a segment-level fixed-priority scheduler, the segments with the smallest relative deadlines have the highest priority. Note that all strands in a segment have the same priority and the same relative release time, though strands from same segment may be placed on different cores. As shown in Figure **??**, the priorities are assigned according to each strand's relative deadline, where priority 1 is the highest priority.

Starting with the highest priority strands, the scheduler then tries to place each strand on a core. To do so, the FBB-FFD [**?**] algorithm defines a RBF (request-bound function) as

$$RBF(\tau_i,t) = e_i + u_i t$$

The RBF is the maximum amount of computation required by task i over time of length t on the system. The above original RBF is tight for sequential tasks and represents the upper bound of computational requirement. However, for decomposed parallel tasks, strands from different segments of the same task will never be released and executed simultaneously. Hence when calculating the total RBF of a task, directly summing the RBF of every strand will be pessimistic.

The offset-aware FBB-FFD algorithm replaces the original RBF with RBF^*, which takes release offsets into account. It calculates all possible interference from other strands of other tasks, as well as strands from the same segment on a given core.

When segment $\tau_{j,l}$ is the first interfering segment, the interference of task τ_j on segment $\tau_{i,k}$ with relative deadline $d_{i,k}$ is defined as

$$RBF_q^*(\tau_{j,l}, d_{i,k}) = \sum_{(r_{j,p}+T_j-r_{j,l}) \bmod T_j \le d_{i,k}} e_{j,p}.m_{j,p,q} + \sum_{j,p} u_{j,p}.m_{j,p,q}.d_{i,k}.$$

This offset-aware RBF is different in the first item by only summing the interference from strands that can be released within the deadline $d_{i,k}$, considering the start segment $\tau_{j,l}$ and all the offsets of subsequent segments.

Then the maximum interference of task τ_j is

$$RBF_q^*(\tau_j^{decom}, d_{i,k}) = \max\left\{RBF_q^*(\tau_{j,l}, d_{i,k}) \middle| 1 \le l \le k_i\right\}$$

A detailed explanation can be found in [**?**].

If the RBF^* of a strand on a given core q satisfies the condition that $d_{i,k} - \sum_j RBF_q^*(\tau_j^{decom}, d_{i,k}) \ge e_{i,k}$ (load condition), then the strand can be assigned to this core. In this case, the strand is guaranteed to not miss its deadline.

For each strand, there may be more than one core that satisfies the load condition. Therefore, there is a choice of assignment algorithms used to place strands on cores. This choice results in different scheduling strategies and potentially different execution results. One contribution of this paper is to experiment with the following two heuristics.

A *first-fit heuristic* will scan cores in some canonical order (from core 1 to core p) and place the strand on the first core that satisfies the load condition. This is the standard method for FBB-FFD bin-packing algorithms. However, it does not provide any load-balancing, and the first few cores become heavily loaded, while the last cores may be entirely unused.

A *worst-fit heuristic* on the other hand, will scan all the cores and find the core with the least RBF^* value (the least loaded core) and will assign the strand to that core. In principle, the worst-fit heuristic should exhibit better load balancing than the first-fit heuristic by spreading computational work across as many cores as possible. However, it takes longer to run the scheduler since each assignment step must scan all cores. An example assignment using worst-fit assignment is shown in Figure **??**. Note that if using first-fit, all strands in this example will be assigned to core 1, simply because the sum of the worst case execution times of both tasks is much smaller than either task's periods.

B. Dispatcher

The dispatcher is responsible for enforcing previously generated schedules and providing synchronization at the end of each segment during runtime. This requires the dispatcher to support scheduling priorities, runtime preemption, and synchronization, which we accomplish through facilities provided by Linux. We use real-time priorities to enforce the schedule and enable task-level preemption. We accomplish segment (barrier) synchronization through *futex* (fast user-space mutex) system calls.

Recall from the previous section: prior to runtime, the scheduler decomposes a task set into individual strands and encodes this in a static schedule. Thus, when the dispatcher begins, it has the program structure of each task, and each strand is annotated with a processor assignment, priority, and relative release time. An example of such an assignment table is shown in Figure **??**. In order to enforce that schedule, the dispatcher must be able to run strands on cores to which they are assigned at the proper release time, and if a high priority strand is released while a low-priority strand is running on its assigned core, then the high-priority strand must preempt the low priority strand.

We describe the operation of the dispatcher in two phases: *initialization* and *runtime operation*.

Initialization: The ability to assign specific strands to processors is accomplished by creating a *team* of threads for each task in the system. Each team has exactly as many threads as available cores. Therefore, if there are n tasks and p processors, there are a total of $n * p$ threads in the system: one thread from each team pinned to each core. The number of tasks and processors are known a priori, so all teams are created and pinned during initialization. This simplifies the dynamic operation of the system, as once these threads are pinned during initialization they never again migrate. In the system shown in Figure **??**, there are two tasks and three processors. Each task has a dedicated team of threads, and the team is distributed so that every task has one thread per processor.

Runtime Operation: A thread from task i's team, pinned on processor j has the following function: it executes all the strands from task i that are assigned to processor j and only those strands. Therefore, given a strand-to-core assignment computed by the scheduler, we have an automatic strand to thread assignment; once a strand is assigned to a processor, there is one corresponding thread that is responsible for executing it. This can be seen in the example in Figure **??**.

The execution of our dispatching system occurs in a completely distributed manner. Each thread is individually responsible for finding the right work to do, and doing it at the right time. This distributed approach has the advantage that we can use all the cores for useful processing, rather than wasting a core for dedicated dispatching, and we avoid overhead due to centralized coordination. The pseudo-code for how dispatching occurs is given in Algorithm **??**.

Algorithm **??** realizes a team of threads synchronously stepping through a task segment by segment, and is executed by all threads at runtime. At the start of each segment each thread will look at the schedule to determine whether some strand is assigned to it from the current segment. Note that more than one strand from a segment may be assigned to the same thread. The thread looks to see if any strands of the segment are assigned to it. If there are, it performs the work of those strands. Once it finishes this work (or if it has no strands assigned to it) it skips to the barrier (line 11) and waits for the rest of its team to finish the segment. Threads wait at the barrier until all threads in the team have reached it.

There is one additional issue. Each thread is responsible for dispatching itself, but each thread is running concurrently with many other threads, some of which are executing real-time workloads. The two dangers are thus: dispatching actions done at a high priority will interfere with currently executing jobs, while dispatching actions done at a low priority could lead to a situation in which a moderately high priority job blocks the dispatch of a higher priority job (which results in priority inversion).

We address this by reserving the maximum real-time priority for dispatching. The choice to have threads spend so much time at the highest system priority might seem counter-

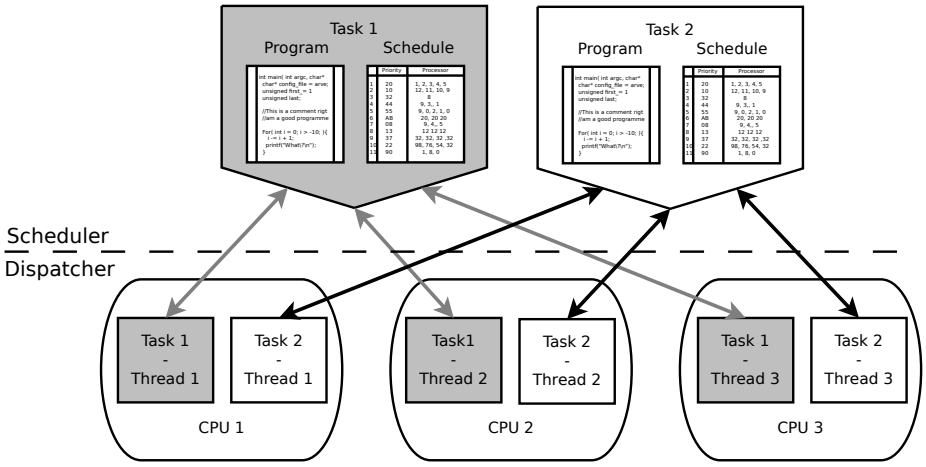


Fig. 3: A system consisting of two tasks and three processors. Each task has a team of threads that are created at runtime, and consists of exactly one thread per processor. The black dashed line represents the division between compile time and runtime: the schedule is generated prior to runtime, but the dispatcher must refer to the generated schedule frequently during execution.

Algorithm 1 Distributed Dispatching

```
1: Raise priority to system maximum
2: while new_task_iteration do
3:     while more_segments_remain do
4:         Wait until segment relative release
5:         Check for any strands from this segment are assigned to me
6:         while has_assigned_strands do
7:             Lower priority to segment priority
8:             Perform the work of the strand
9:             Raise priority to system maximum
10:        end while
11:        Team barrier synchronization
12:    end while
13:    Sleep until next task iteration
14: end while
```

intuitive, as this means that all dispatching actions will always block the real-time execution of any currently working thread (even if those dispatching actions are for a lower priority thread). However, each thread is only ever performing one of three actions while dispatching: checking for work, modifying its own priority, or barrier waiting. These three actions can be made brief enough that they do not significantly disrupt the operation of other threads in the system. In essence, we have traded long and unpredictable priority inversion (a long-running low priority thread blocking the dispatch of a high priority thread) for very brief and very predictable priority inversion (the brief but frequent dispatching actions of every thread).

Preemption: Note that preemption occurs correctly and automatically in this design. Each thread executes a strand assigned to it at the priority of the strand itself. Therefore, when a high-priority strand is released on processor p, the thread that is responsible for that strand inherits this high priority. If another thread is executing a lower-priority task on processor p, that thread has the lower priority. As a result, the high-priority thread will preempt the lower priority thread through the normal Linux thread scheduling mechanism. As can be seen in Figure **??** strand s_1^3 has higher priority than strand s_2^1. Therefore, when s_1^3 is released, the thread responsible for it immediately preempts the thread executing s_2^1. When s_1^3 completes, s_2^1 resumes its execution.

Synchronization Mechanism: Finally, the dispatcher must ensure that no thread executing a parallel-synchronous task can race ahead and begin executing a future segment before it's predecessors have finished. We ensure this in our system through barrier synchronization, and we now describe how this barrier is implemented efficiently.

Recall that barriers have two operations (*wait* and *wake*) that allow a team of threads to synchronize with one another. When a thread reaches a barrier it waits there until all other threads arrive. Once this happens all threads are awoken and allowed to proceed. This is the precise behavior desired for segment synchronization, and prevents any one thread from racing ahead and starting on a new segment while other threads are still working on the previous.

In our system the wait and wake operations are performed at the system's maximum real-time priority (to address the same priority inversion problem as with dispatching). This necessitates a barrier implementation that is as non-interfering as possible. To achieve this we use the *futex* (fast userspace mutex) system call within Linux.

Futexes are single atomic counters in shared memory, used to support efficient mutual exclusion. There are two system calls that allow the kernel to arbitrate between processes that are contending on the same futex: **`futex_wait`** and **`futex_wake`**. When a thread waits on a futex it yields the processor and is put to sleep by the kernel. Later, some other thread wakes the futex, which revives some or all of the threads that were previously waiting.

We use **`futex_wait`** to implement our barrier wait, and **`futex_wake`** to implement our barrier wake. This is especially advantageous for our system: in our design the threads spend almost all their time either working or barrier

waiting at the maximum system priority. Many waiting threads at such high priority might contribute significant overhead. However, with the futex implementation we invoke the kernel to put the threads to sleep, and they consume no resources in this wait state. This allows us to have many threads idling at the system's highest real-time priority without incurring substantial overhead.

V. Design Space Considerations

In the design of our real-time scheduling service, we made choices among many alternatives. In this section, we describe some of those alternatives, and discuss the pros and cons of our choices with respect to these alternatives. We discuss three aspects of our system: the scheduling strategy, the preemption mechanism, and the synchronization mechanism.

Scheduling: We chose to use partitioned DM (Deadline Monotonic) scheduling. The alternative would have been to use global EDF (Earliest Deadline First), which, in fact, provides a better augmentation bound of 4 (instead of 5 provided by partitioned DM). We chose to implement partitioned DM for our first prototype for multiple reasons. First, partitioned DM is easier to implement on a multi-core system by leveraging thread priorities. Dynamic priority schedulers are more difficult to implement using OS priorities (and a user-space scheduler that implements preemption also would have been difficult, as discussed subsequently). Second, partitioned scheduling has lower overheads for several reasons: (1) Scheduling occurs statically, so there are no overheads of computing the schedule at run time. (2) Strands do not migrate from one core to another during execution. (3) Preemption occurs more rarely and predictably since strands can only preempt other strands that are assigned to the same core. For these reasons, we chose to start by implementing this strategy. In the future, we plan to explore global dynamic priority scheduling strategies in our concurrency platform.

Preemption: There are two ways to implement preemption between threads. One can either rely on the *operating system mechanisms* to provide preemption (as we do), or implement user-space preemption by *voluntary yielding*. For user-space preemption, each thread must periodically check if it has been preempted. If it has, it should save its current state and yield the core to the preempting thread. This has the advantage that it doesn't involve expensive system calls. In addition, this is often safer for programs that use mechanisms such as locks, since threads can make sure that they are at a safe point before they yield. On the other hand, this method has a few disadvantages as well. The user (or compiler) must provide the mechanism for periodic polling and checkpointing. Moreover, a high-priority thread may have to wait for a long time before a low-priority thread decides to yield its processor. Due to this priority inversion, it is difficult to provide real-time performance unless there are bounds on how long it can last and how often it can happen. In this paper, we are interested in validating the real-time performance provided by the theory presented in [**?**], which assumes instantaneous preemption. In addition, our tasks do not have any locks, so there is no danger of threads being preempted while holding locks. Therefore, we chose to implement the OS interrupt strategy. In the future, we plan to explore yielding mechanisms for our real-time scheduling service (which might be used with a user-space scheduler design).

Synchronization: In our system design, each thread must wait on a barrier for other threads of its team to finish executing the current segment. There are two ways we could have implemented this waiting. One method is to use *sleeping* (as we do), and the other is to use *polling*. Both methods involve waiting for a specific condition to change, but their implementation differs. Sleeping generally means that a thread is removed from consideration of the scheduler (through elimination from the runqueue) and hence does not consume any processor time, even indirectly, until it is woken. The downside of this is that the operating system must become involved both to suspend and resume the thread. The other approach, polling, involves spinning until the condition becomes true. Unlike a sleeping thread, a polling thread continuously occupies the processor. The benefit is that polling generally yields better latency; a thread can get past the barrier faster once the condition becomes true. Hence, polling is the preferred strategy if it is known that wait times will be very short. In our system, the number of threads is much larger than the number of cores, and many threads spend a large amount of time waiting on a barrier. Therefore, sleeping is the right choice for our mechanism, and is implemented using the `futex_wait` system call.

VI. Evaluation

In this section, we present an experimental evaluation of our system. We present two types of experiments: (A) full-system evaluation using synthetic parallel tasks to see if the scheduling service meets task deadlines, and (B) micro-benchmarks in order to understand the overheads of the mechanisms used by our system.

A. System Evaluation: Synthetic Parallel Tasks

One of the goals of this evaluation is to determine whether our system agrees with the theoretical augmentation bound of 5. Even though that bound holds in theory, the overheads of a practical implementation might invalidate it on a real system. We generate theoretically schedulable task sets such that (1) utilization by the task set is at most $m/5$ (20% of maximum allowed utilization), and (2) each task's critical path length is at most $1/5$ its deadline (again, 20% of maximum). We call these 20% *utilization* tests for brevity.

To evaluate the practical applicability of our approach more broadly, we considered several parameters, as follows.

Utilization Level: The theoretical results hold only for 20% utilization task sets. Since theoretical results are often pessimistic, we also evaluate our system on higher utilization task sets.

Task Frequency: Each periodic iteration of a task costs a (relatively) fixed amount of overhead. Hence, a task that

executes at 1000Hz (1000 times per second) will incur approximately ten times more overhead than a task that executes at 100Hz, and this additional overhead may wreck performance. We explore several timescales to quantify this effect.

Bin-Packing Heuristic: As we described in Section **??**, the heuristic changes how work is assigned to processors: the first-fit heuristic heavily loads as few processors as necessary and leaves the rest underutilized, while the worst-fit heuristic attempts to minimize the maximum load across all processors. Theoretically both heuristics should guarantee schedulability for 20% utilization task sets; but we expect their performance may differ for higher utilization task sets.

Number of Processors: We are also interested in how well our approach scales as we increase the number of processors involved. The size of each thread team increases by one for every processor used in the system, which increases the synchronization overhead at the end of every segment. In addition, each processor chip in our machine has 12 cores. When we use more than 12 cores, the teams have to synchronize across multiple chips, potentially leading to communication overheads.

Test Platform: We tested our runtime system with a 48-core symmetric multiprocessor, a 1U AMD Quad with four Opteron 6168 processors. We used standard Linux kernel version 3.4.4 with RT_PREEMPT patch version r14 as our underlying RTOS. We left processors 0-11 in their default configuration (to handle normal Linux activities and interrupts), and processors 12-47 were optimized for real-time performance. This was done by isolating them from the Linux scheduler and load balancer with the boot parameter `isolcpus` and preventing them from servicing all maskable interrupts. This gave us 36 processors on which to run real-time task sets.

Task Set Generation: Task set generation is straightforward; tasks were randomly generated and included in the task set until the total utilization of the whole set was as desired. Given a certain number of processors, the goal is to generate a parallel synchronous task set to within 2% of the desired utilization (i.e. if the desired utilization level is 50% then the actual utilization will be between 48% and 50%). Task periods and strand lengths are unitless and can be scaled at runtime to achieve a desired task frequency.

First, the period of the task is chosen to be 2^i, where $i \in \{11, 12 \ldots 16\}$. To conform to the scheduling theory, the critical path length of the task was chosen to be 8%, 10%, 14% and 20% of the period, with probability of 0.4, 0.3, 0.2 and 0.1 respectively, to yield tasks that have varying levels of slack. As indicated above, the maximum allowable critical path length is 100% of the period of speed-$1/5$ ideal processors, or 20% of the period on our processors. This methodology gives us critical path lengths of 40%, 50% 70% and 100% of the maximum allowable critical path length.

Given these parameters, the task is generated segment by segment to get a series of segments such that the critical path length of the task is equal to the chosen critical path length. To do so, we generate each segment in turn and randomly choose its execution time from a log normal distribution. This allows us to control the distribution mean while still allowing for occasional large and small values. The average segment length was 400, and the minimum segment length was 100. The number of strands in each segment also was chosen from a log-normal distribution with mean 4 and minimum value 1.

Minimum Task Period	Maximum Task Period	Minimum Segment Length	Average Segment Length
2048ms	2^{16}ms	100ms	400ms
32ms	1024ms	1563μs	6250μs
16ms	512ms	781μs	3125μs
8ms	256ms	391μs	1563μs
4ms	128ms	195μs	781μs
2ms	64ms	98μs	391μs

TABLE I: Several timescales allow us to validate the system for a variety of potential application domains. The top three timescales demonstrate the limits of the design under 36-core operation, and the bottom three timescales demonstrate the limits for 12-core operation.

Methodology: We ran experiments with $m = 12$ and $m = 36$, where m is the number of cores. For both values of m, we generated task sets with utilizations between 20% and 80%. For $m = 12$ we generated 100 task sets, and for $m = 36$ we generated 20 task sets. Each task set was then scheduled with both the first-fit and worst-fit heuristics. Each task set was run for five minutes of wall-clock time under various timescalings. The absolute values derived from each timescaling can be seen in Table **??**.

For each experiment, we calculate the ***failure rate***. A task set is said to have failed if any task misses a deadline. The failure rate is the ratio of the number of task sets that failed to the total number of task sets. Before presenting these results we first describe a series of system overhead measurements.

B. System Overhead Measurements

As described earlier, one goal of our system design is to minimize the overhead due to contention. There are two primary sources of contention within the system, preemption and segment (barrier) synchronization. We evaluate these mechanisms with micro-benchmarks: short programs designed to expose a specific facet of system performance.

Preemption Overhead: Our first micro-benchmark is designed to measure the effect of preemption on scheduling latency, which we define to be the difference in time from when a job may start executing (its release time when it has higher priority than any other job that is currently executing) to when it actually starts. We use two jobs to accomplish this: the first is a low-priority job that executes for a long time on twelve cores simultaneously. The second is a 12-core high-priority job whose release time is a fixed interval after the start of the low-priority job. There are two salient features: the second job should always preempt the first job immediately upon its release, and we always know the precise time of that release. Hence, we can measure the difference in the second task's release time and the time it actually starts executing code. This always involves preempting task one, and thus we

	Scheduling Latency (μs)	12-Core Barrier Latency (μs)	36-Core Barrier Latency (μs)
25^{th} Pct.	9.2	11.7	3277.5
50^{th} Pct.	9.9	12.2	3284.9
75^{th} Pct.	10.8	15.6	3290.1
95^{th} Pct.	12.5	18.1	3296.6
Max	27.5	76.8	6503.5

TABLE II: The 25^{th}, 50^{th}, 75^{th}, and 95^{th} percentiles for the scheduling and barrier latency micro-benchmarks, as well as the worst-case observed latency.

consider this to be a practical measure of overhead due to preemption.

Barrier Latency Delay: The second micro-benchmark addresses the segment synchronization delay. Whenever a team of threads moves through a barrier there will be many threads waiting on the barrier and only one thread to wake them. This could introduce a significant delay for whichever thread happens to be woken last. This micro-benchmark is straightforward: a team of threads goes through a barrier and they timestamp immediately before and after. The last pre-barrier timestamp is the time that the last thread entered the barrier, so it is the time that all threads become eligible to proceed. The last post-barrier timestamp is the time that the last thread left the barrier, which is the thread that suffered the most delay. Thus, we can compare these two timestamps to determine the true total delay of the barrier operation— the total amount of time it took for a thread to actually leave the barrier once it was semantically allowed to.

We perform the barrier latency micro-benchmark twice: once with a team of 12 threads on 12 cores, and once with a team of 36 threads on 36 cores. As previously discussed, we expect the 36-core version to incur greater overhead both because of the greater number of threads in the team as well as the cost of communicating across multiple processor chips.

All micro-benchmark results are presented numerically in Table **??**. We see that the overhead is generally small for teams of 12-cores: less than 30 μs overhead for preemption and less than 80 μs delay for barriers. However, the overhead of barrier synchronization for large teams is several orders of magnitude greater, requiring more than 3ms in all cases.

C. Empirical Results

Figures **??** through **??** present the results of our experiments. We now evaluate those results.

Our experiments show that all 12-core task sets are schedulable at 20% utilization once the minimum task period is 4ms or greater. This validates the theoretical augmentation bound results for those timescales, and demonstrates the suitability of the system to handle applications that require task frequencies of 250Hz. It is difficult to state whether the 2ms boundary is due entirely to preemption and barrier overhead or an additional factor, because the total incurred overhead is dependent on the exact task set configuration (how many preemptions and barrier synchronizations occur).

As we run less demanding tasks we're able to execute higher utilization task sets. Between figures **??** and **??** the achieved utilization grows from 20% to 30% at the expense of doubling the shortest task period. For 12-core operation we are able to achieve full schedulability at reasonably high utilization (greater than 50%) when the shortest task periods are greater than or equal to 16ms.

The large 36-core task sets are much more difficult to schedule, which is expected given the much higher barrier synchronization overhead. We only successfully schedule all 20% utilization tasks once the minimum task period is 16ms. We can achieve reasonably high utilization if we double the minimum task period to 32ms. This is not fast enough to run real-time applications that require extremely short timescales, but does demonstrate the suitability of the system for applications that have slightly longer periods but require much more processing power, for example processing video in real-time at 25 frames per second.

The barrier overhead appears to be the primary limiting factor in how fast we can run large teams of threads. Minimizing or avoiding multi-chip communication would appear to be necessary for any real-time systems that require sub-millisecond operation.

Packing Heuristic: One very clear result concerns the performance of the first-fit and worst-fit bin packing heuristics. The worst-fit heuristic dominates the first-fit heuristic, meaning that there was not a single task set where the first-fit heuristic was successful but the worst-fit heuristic was not. The worst-fit heuristic also scales better at longer timescales, which actually is not due to the timescale: at high utilizations the first-fit heuristic tends to over-utilize individual processors, resulting in a task set that is unschedulable under any scaling.

This provides strong evidence for the source of task set failures. If task failures were primarily due to contention and synchronization overhead, then worst-fit would be worse, since it potentially spreads a task across many cores, while first-fit clusters the tasks on a small number of cores. This result seems to suggest that the primary danger in our system is over-utilization of individual cores, rather than the contention overhead due to the cooperation of many cores.

This seems to be confirmed by the 2048ms timescale experiment, a timescale so large that it is extremely unlikely that any deadline misses arise due to overheads. At the 70% utilization level the first-fit heuristic begins to perform extremely poorly, while worst-fit has only a small increase in the number of unschedulable task sets. From a system design perspective the worst-fit heuristic appears to be a better choice, as it seems to offer a much larger margin of safety.

VII. Related Work

Most work on scheduling parallel tasks has been done in the parallel computing community. There has been extensive theoretical work on how to improve load-balance [**?**], number of cache-misses [**?**], resource allocation between different jobs [**?**], etc. Many parallel processing languages and runtime systems have been designed [**?**], [**?**]. The schedulers of these runtime systems are often based on provably good scheduling techniques such as ***work-stealing***. None of these systems

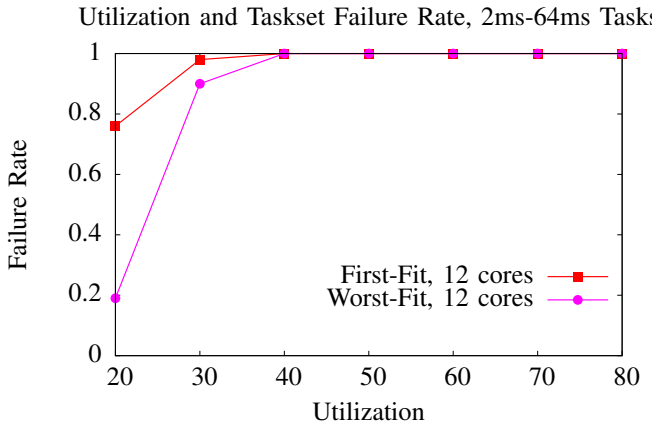


Fig. 4: Task set utilization vs. failure rate (both in percentages) for the 2ms timescale. Both the worst-fit and first-fit task sets failed at 20% utilization. This shows that at the 2ms timescale the scheduler's theoretical assurance fails due to system overheads. All 36-core task sets failed, and their results are not shown.

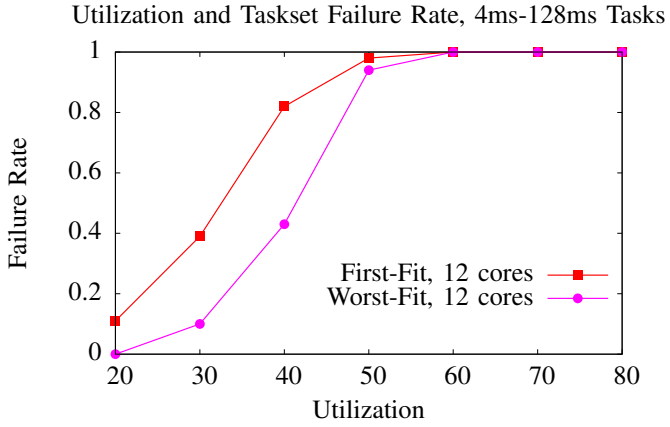


Fig. 5: Task set utilization vs. failure rate (both in percentages) for the 4ms timescale. The worst-fit heuristic succeeded at 20% utilization, but failed otherwise. Most 36-core task sets failed, and their results are not shown.

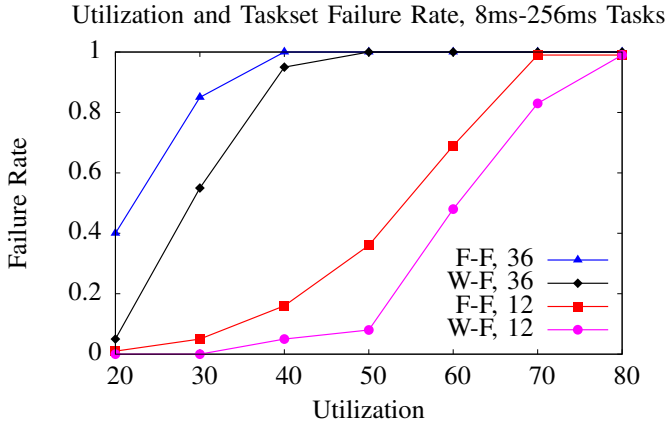


Fig. 6: Task set utilization vs. failure rate (both in percentages) for the 8ms timescale. Failure means at least one periodic deadline miss. The first-fit task sets are never completely schedulable, while the 12-core worst-fit task set is schedulable up to 30%. Worst-fit (W-F) and first-fit (F-F) are abbreviated.

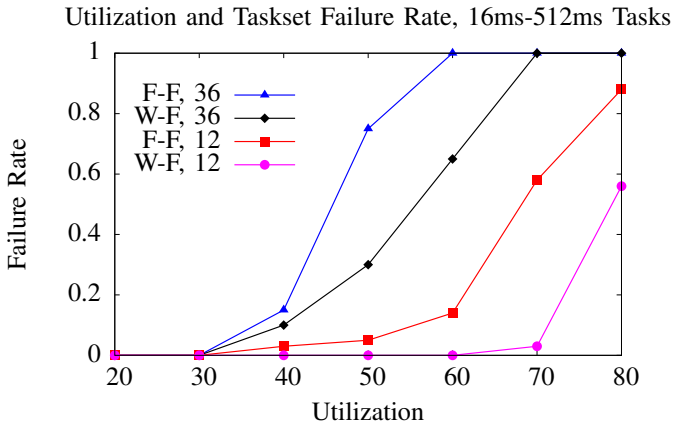


Fig. 7: Task set utilization vs. failure rate (both in percentages) for the 16ms timescale. Worst-fit (W-F) and first-fit (F-F) are abbreviated.

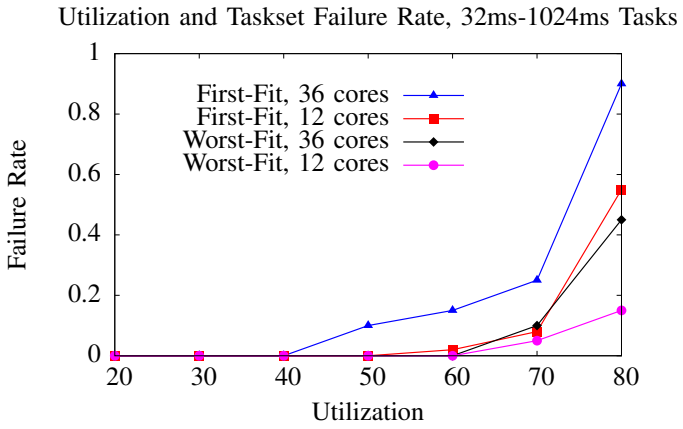


Fig. 8: Task set utilization vs. failure rate (both in percentages) for the 32ms timescale. This demonstrates how the worst-fit heuristic scales better than the first-fit heuristic, as the 36-core worst-fit task sets are approximately just as difficult to schedule as the 12-core first-fit task sets.

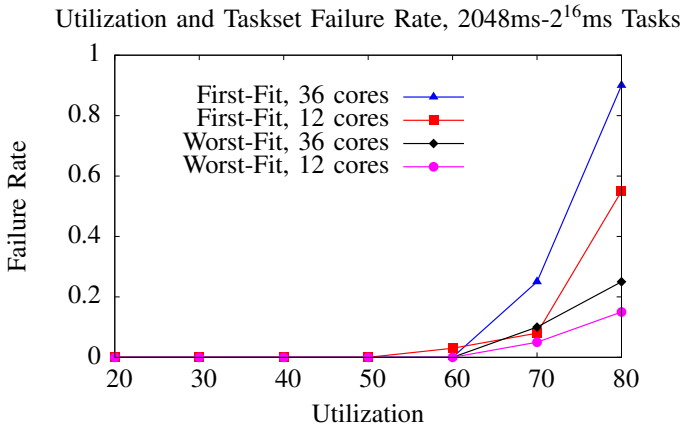


Fig. 9: Task set utilization vs. failure rate (both in percentages) for the 2048ms timescale. Failure means at least one periodic deadline miss.

provides a way to specify real-time semantics, nor do these scheduling techniques take deadlines into consideration. In the domain of multiprocessing real-time systems, LITMUS^{RT} [**?**] provides operating system support for a variety of schedulers for multiprocessing systems. In [**?**], the authors compared different multiprocessing scheduling strategies such as global EDF and deadline monotonic. Neither of these systems considers tasks with intra-task parallelism.

There has been recent theoretical work on scheduling real-time parallel tasks. For a limited subset of parallel synchronous tasks, Lakshmanan et al. [**?**] provide a decomposition algorithm to convert parallel tasks into sequential subtasks, which are then scheduled using the partitioned scheduler FBB-FDD [**?**], guaranteeing a resource augmentation bound of 3.42. Saifullah et al. [**?**] provided a different decomposition algorithm for more general parallel synchronous tasks and prove that this decomposition guarantees an augmentation bound of 4 using global EDF and 5 using partitioned DM. Our work is based on this latter work; we use this decomposition algorithm combined with partitioned deadline monotonic scheduling.

Researchers have also considered more general task models, particularly those where tasks are represented by general DAGs (Directed Acyclic Graphs). Saifullah et al. [**?**] extended their results for synchronous tasks to DAGs for both pre-emptive and non-preemptive scheduling. There has also been recent theoretical and simulation-based work on scheduling parallel tasks without decomposition. Nogueira et al. [**?**] present some theoretical results for hard-real time systems and simulation results for soft-real time tasks when using priority-based work-stealing algorithms. Their algorithm only provides hard-real time guarantees when each parallel task has utilization of at most 1. Researchers have also considered using global EDF without decomposition for soft-real time systems and analyzed the response time [**?**]. Baruah et al. [**?**] provide an augmentation bound of 2 when scheduling task sets consisting of a single task.

In contrast to all the theoretical and simulation-based work for scheduling parallel real-time tasks, we present, what is to our knowledge, the first scheduler and dispatcher design for a real Linux-based platform for intra-task parallelism. Our prototype currently only supports one scheduling strategy and only works for parallel synchronous tasks. In the future, we plan to extend it to support more general task models (such as DAGs) and other scheduling strategies (such as global EDF and work-stealing based strategies).

VIII. Conclusions

We have described the design, implementation and experimental evaluation of a prototype runtime system for parallel real-time programs. Our design consists of a static scheduler — which decomposes parallel tasks into sequential tasks, maps them on individual cores, and assigns priorities — and a runtime dispatcher — which executes these tasks while guaranteeing that they obey all dependence and priority constraints. Our experiments demonstrate suitable real-time performance for useful timescales, and moreover we have demonstrated that multi-chip parallel operation is feasible for applications that require large amounts of processing power.

There are many directions of future work for this system. First, we want to build a full system that takes OpenMP programs with real-time constraints, compiles them, schedules them, and dispatches them. Second, we want to experiment with other scheduling strategies. Our current system uses decomposition combined with partitioned deadline monotonic scheduling. We would like to add both global and dynamic scheduling strategies to our platform so we can compare them in practice. Finally, we want to run real task sets rather than synthetic benchmarks to evaluate the effects of other phenomena such as cache-misses, shared variables and I/O. We are particularly interested in real-time hybrid testing applications due to ongoing collaborations with researchers interested in testing earthquake resiliency of structural components. These applications do require shorter periods, and so mechanisms for low-overhead parallel coordination across multiple processor chips is a priority.

Acknowledgement

This research was supported in part by NSF grant CCF-1136073 (CPS).