Federated Scheduling for Parallel Real-time Tasks

Jing Li, David Ferry, Kevin Kieselbach, Abusayeed Saifullah,
Kunal Agrawal, Christopher Gill, and Chenyang Lu
Department of Computer Science and Engineering
Washington University in St. Louis
{li.jing, dferry, kevin.kieselbach, saifullah}@go.wustl.edu, {kunal, cdgill, lu}@cse.wustl.edu

***Abstract*—We present a novel *federated scheduling algorithm* for parallel real-time tasks. This algorithm provides a *capacity augmentation bound* of $\frac{3+\sqrt{5}}{2} \approx 2.618$, the best such bound known so far for parallel real-time tasks. In addition, we present a portable platform design FSS based on a federated scheduling service, and investigate the real-time performance of a simple implementation that supports parallel programs written in Cilk Plus and OpenMP, both of which are widely used parallel languages. This platform allows us to run existing parallel applications written in these languages with minimal modifications. Moreover, we measured and incorporated system overheads due to preemption and parallel scheduling into the federated scheduling test. Our experiments indicate that using this improved algorithm, FSS provides good real-time performance and easily outperforms an existing scheduling service for real-time programs.**

***Index Terms*—real-time scheduling, parallel scheduling, federated scheduling, capacity augmentation bound**

I. Introduction

Multi-core processors are changing the computing landscape as the number of cores per chip continues to increase. To harness the power of these cores to perform more computationally intensive tasks in real-time, we must develop new approaches that can support parallel real-time task sets.

There has been extensive prior research on ***multiprocessor scheduling***, which support ***inter-task parallelism***; multiple tasks can take advantage of multiple cores by running concurrently with each other, but each task is sequential and can use only one core at a time. In contrast, in ***parallel scheduling***, we also allow ***intra-task parallelism*** where individual tasks are themselves parallel programs. Therefore, unlike multiprocessor systems, individual tasks can run faster as the number of cores increases. Parallel real-time systems are particularly well suited to handle tasks that process large quantities of data, such as in hybrid real-time testing [1]. One recent example [2] showed that parallel execution provides demonstrably better system performance for autonomous vehicle route planning.

There has been some recent theoretical work on scheduling parallel real-time tasks [2–8]. These results generally fall into two categories. Some provide scheduling strategies where each parallel task is decomposed into a set of sequential subtasks and these subtasks are then executed using existing multiprocessor scheduling algorithms [2–4]. Others analyze the performance of global EDF for parallel tasks under various assumptions [5–8]. With respect to system design and implementation, to our knowledge, only two systems [2, 9] have been implemented for executing parallel real-time programs. Both are based on task decomposition strategies, which usually require substantial modification to application programs or compilers and exact task structure with details of each subtask in a task.

In this work, we describe a novel ***federated scheduling strategy*** for parallel real-time tasks. The theoretical contributions of this paper are:

1) We propose a ***federated scheduling algorithm*** for parallel sporadic task sets with implicit deadlines where tasks are represented by ***directed acyclic graphs (DAGs)***. Each high-utilization task (utilization > 1) is allocated a dedicated cluster (set) of cores (number of cores depends on its utilization). A multiprocessor scheduling algorithm is used to schedule all low-utilization tasks sequentially on a shared cluster composed of the remaining cores. This strategy does not require any decomposition.
2) For this scheduler, we prove a ***capacity augmentation bound*** of $\frac{3+\sqrt{5}}{2} \approx 2.618$ for parallel DAG tasks. This bound implies that, given a task set τ and a machine with m cores, our scheduler can schedule the task set if (1) the total utilization of the tasks in τ is at most $m/2.618$ and (2) the critical-path length of each task is at most $1/2.618$ of its deadline. The best previous capacity augmentation bound is 3.42 for parallel synchronous tasks (a more restricted class than general DAG tasks) using task decomposition [4]. The best result for general DAGs uses global EDF and provides a capacity augmentation bound of 4 [8].
3) A capacity augmentation bound provides a natural schedulability test, since we can check if the task set satisfies the above two conditions. However, the scheduling algorithm itself is also a schedulability test (if the algorithm returns a valid core allocation for all tasks, then the task set is schedulable, otherwise it is not) that, in practice, often admits task sets with utilization much greater than $m/2.618 \approx 38\%m$.

In addition, we built a scheduling service for the federated scheduling strategy, and integrated it into a platform for real-time parallel tasks, called **FSS**. This platform has the following features:

1) Application developers can write their tasks using either Cilk Plus [10] or OpenMP [11]. Both are well-known and feature-rich languages for writing parallel programs. Existing parallel OpenMP or Cilk programs can be run on our platform with real-time semantics by making only minimal modifications.
2) For low-utilization tasks, FSS uses a rate-monotonic scheduling policy. For high-utilization tasks, FSS simply

uses the scheduler of the respective parallel runtime system (e.g. for Cilk Plus or OpenMP). Through experiments, we find that FSS provides reasonable real-time performance even with schedulers of such out-of-box runtime systems.
3) We demonstrate the generality and portability of FSS by instantiating it using two mainstream parallel languages, Cilk Plus and OpenMP, without significant modifications to their runtime systems.
4) We measure and characterize the run time overheads (due to preemption and parallel scheduling) of FSS using micro-benchmarks and incorporate them into the scheduling algorithm and test. The achieved scheduling algorithm effectively enhances the predictiveness and real-time performance of our platform.

We evaluated the federated schduling platform FSS using Cilk Plus and OpenMP programs on both a 14-core Intel machine and a 36-core AMD machine. The goal is to evaluate the kind of real-time performance we can achieve using generic schedulers not initially designed for real-time performance. On our set of synthetic benchmarks, our improved scheduling algorithm performs well even on tasks with extremely tight deadlines (as small as 1ms). In addition, the schedulability test that takes overheads into account is predictive — when the algorithm says a task set is schedulable, the platform can schedule it —for all of our task sets in 12, 14 and 36 core experiments. Moreover, FSS easily outperforms prior decomposition-based parallel real-time platform [9]. From the comparison between OpenMP and Cilk Plus, we find that they have comparable performances in most cases, and the performances of Cilk Plus is slightly better with large number of cores and highly parallel tasks.

The outline of the paper is as follows: Section II discusses related work. Section III provides our scheduling algorithm and proves the augmentation bound. We discuss our platform design in Section IV, while Section V describes how the measured overhead can be incorporated into our federated scheduling algorithm. Section VI shows experimental results and we conclude in Section VII.

II. Related Work

In this section, we review closely related work on both theoretical and system-based multiprocessor and parallel real-time scheduling. Real-time multiprocessor scheduling has been studied extensively (see [12, 13] for survey) Here, we focus on some relevant work on implicit deadline tasks. For these task sets, many scheduling strategies such as EDF, both global and partitioned, and rate monotonic scheduling provide a utilization bound of about 50% [14–17]. In particular, for our platform, we implemented a multiprocessor partitioned rate monotonic scheduling approach called ***mPRM*** [17] to schedule low-utilization tasks on a subset of cores. We used a strategy similar to [18] to pin sequential tasks on subsets of cores via affinity mask in Linux.

The CONFIG_PREEMPT_RT [19] patch on Linux with a push-pull scheduler can be used for multiprocessor fixed-priority scheduling; we use this patch in our platform. Systems such as LITMUS$^{\text{RT}}$ [20, 21] provide operating system level support for multiprocessor overhead-aware scheduling, and middleware [22] exists to provide it in user space. Others [23] have performed empirical comparisons between RM and EDF under global, partitioned, and clustering schemes. However, these systems do not support tasks with intra-task parallelism.

For parallel real-time tasks, most early work considered intra-task parallelism of limited task models [24–27]. Researchers have since considered more realistic synchronous tasks and general directed acyclic graphs (DAGs) that more naturally represent programs written in common parallel languages. Some prior work relies on a task decomposition (or transformation) strategy. Lakshmanan et al. [4] proved an augmentation bound of 3.42 for a restricted synchronous task model. For general synchronous tasks, Saifullah et al. [3] proved a capacity augmentation bound of 4 when decomposed sequential tasks are scheduled by GEDF and 5 by deadline monotonic scheduler. With different improved decomposition strategies, Kim et al. [2] achieved a capacity augmentation bound to 3.73 and [28] a resource augmentation bound of 2 for general synchronous tasks. A schedulability test is provided for synchronous tasks under EDF in [29].

Scheduling without decomposition has been studied for the more general task model — directed acyclic graphs (DAGs). For soft real-time scheduling, Liu and Anderson [5] provide a response time analysis for GEDF, and Nogueira et al. [30] investigate a work-stealing scheduler. A resource augmentation bound of $2 - \frac{1}{m}$ under GEDF was proved for: a staged DAG model [31]; a single DAG with arbitrary deadlines [6]; and for multiple DAGs [7, 8]. A capacity augmentation bound of $4 - \frac{2}{m}$ was proved in [8] for implicit deadlines.

We are aware of two systems [2, 9] that support parallel real-time tasks; both are based on different strategies for task decompositions. Kim et al. [2] used a reservation-based OS to implement a system that can run parallel real-time programs for an autonomous vehicle application, demonstrating that parallelism can enhance performance for complex tasks. Ferry et al. [9] developed a parallel real-time scheduling service on standard Linux. However, since both systems adopted task decomposition approaches, therefore they require users to provide exact task structures and subtask execution time details in order to decompose tasks correctly. The system of [9] also requires modifications to the compiler and runtime system to decompose, dispatch and execute parallel applications. Our federated scheduling strategy, in contrast, allows mostly unmodified parallel programs to be run using mostly unmodified Cilk Plus and OpenMP, but still provides good real-time performance.

Finally, there has been significant work on purely parallel systems. Many parallel languages and runtime systems have been devised, such as the Cilk family [10], OpenMP [11], and Intel's Thread Building Blocks [32]. Most of these systems are built to execute single parallel programs on pre-allocated cores to maximize throughput. While multiple tasks on a single platform have been considered in the context of fairness in resource allocation [33], none of this work considers real-time

constraints.

III. Federated Scheduling Algorithm

In this section, we present our federated scheduling algorithm for parallel task sets and prove its capacity augmentation bound of $\frac{3+\sqrt{5}}{2} \approx 2.618$. The key idea is simple: each high-utilization task is executed on its own set of dedicated cores, and can be scheduled using any greedy scheduler; all tasks with low utilization are scheduled on the remaining cores using an existing multiprocessor scheduling technique.

A. Task Model

A parallel task can be represented as a directed acyclic graph (DAG), such that each node is a segment of computational work (also called a subtask) and each edge is a dependence between two subtasks. Each node in the DAG is sequential subtask and edges represent dependences between subtasks. A node is ***ready*** to execute only when all of its predecessors have finished. A task set τ consists of n tasks $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$, each of which can be a parallel task. A real-time task τ_i has a period P_i and deadline D_i. In this paper, we consider sporadic task sets with implicit deadlines, i.e. $P_i = D_i$. The total ***work*** C_i of a task τ_i is its ***worst case execution time*** when it is running on a single core; therefore, it is the sum of the worst case execution times of all the nodes (subtasks). When running (hypothetically) on an infinite number of cores, the worst case execution time of the task τ_i is called the ***critical-path length*** L_i. Let $u_i = \frac{C_i}{P_i}$ be the ***utilization*** of task τ_i. We want to schedule the task set τ on a multicore machine with m cores.

B. Federated Scheduling Algorithm

Given a task set τ_i, the federated scheduling algorithm works as follows. First, tasks are divided into two disjoint sets: τ_{high} contains all ***high-utilization tasks*** — task with utilization greater than one ($u_i > 1$), and τ_{low} contains all the rest ***low-utilization tasks***.

Second, we assign a dedicated cluster of cores to each high-utilization task. For each such task having deadline (and period) D_i, critical-path length L_i, and worst case execution time C_i, we assign n_i cores to it, where

$$n_i = \left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil$$

Then, a greedy parallel scheduler can be used to schedule τ_i on these n_i cores. Informally, a greedy scheduler is one that never keeps a core idle if some work is ready to execute. (In Section III-D Theorem 2, we prove n_i cores are sufficient to schedule τ_i using any greedy scheduler.) We denote n_{high} as the total number of cores assigned to high utilization tasks.

Finally, we assign the remaining cores to all low-utilization tasks, denoted as $n_{\text{low}} = m - n_{\text{high}}$. We observe that parallel execution is not required to meet the deadlines of low utilization tasks since $C_i \leq D_i$. Hence, we consider these tasks to be sequential and use a multiprocessor scheduling algorithm for sequential tasks. Any multiprocessor scheduling algorithm that provides a utilization bound of at least 38% can be used, such as global EDF [15], partitioned EDF [16], or various rate-monotonic schedulers [17].

C. Capacity Augmentation Bound

We first define *capacity augmentation bound*, which is the property we prove about the federated scheduling algorithm.

Definition 1. *A scheduling algorithm $\mathcal{S}$ provides a **capacity augmentation bound** of b if, given any implicit deadline task set τ with total utilization $\sum u_i$, it can schedule this task set on m processors if the following **conditions** are satisfied by the task set:*

$$\text{Total available cores, } m \geq b \sum u_i \tag{1}$$

$$\text{For each task, } D_i \geq bL_i \tag{2}$$

We should note that a commonly used ***resource augmentation bound*** of b implies that $\mathcal{S}$ can schedule any task on m cores of speed b if an optimal scheduler can schedule it on m cores of speed 1. The capacity augmentation bound is stronger than it, since any scheduler with a capacity augmentation bound of b automatically has a resource augmentation bound of b.

Theorem 1. *The federated scheduling algorithm has a capacity augmentation bound of $\frac{3+\sqrt{5}}{2}$.*

To prove Theorem 1, we consider any task set that satisfies Conditions 1 and 2 for $b = \frac{3+\sqrt{5}}{2}$. Then, (1) state the relatively obvious Lemma 1; (2) prove the schedulablity of each high-utilization task; (3) calculate the lower bound on the size of the remaining cores for low-utilization tasks; and (4) use existing multiprocessor scheduling theory to prove the schedulability of low-utilization tasks. These four steps complete the proof.

Lemma 1. *A task set τ is classified into disjoint subsets $s_1, s_2, ..., s_k$, and each subset is assigned a dedicated cluster of cores with size $n_1, n_2, ..., n_k$ respectively, such that $\sum_i n_i \leq m$. If each subset s_j is schedulable on its n_i cores using some scheduling algorithm $\mathcal{S}_i$ (possibly different for each subset), then the whole task set is schedulable on m cores.*

D. High-Utilization Tasks Are Schedulable

Here, we prove that any greedy scheduler can schedule a high utilization task τ_i on a dedicated cluster of n_i cores, where $n_i = \left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil$. Assume that a machine's execution time is divided into discrete quanta called ***steps***. During each step a core can be either idle or performing one unit of work. We say a step is ***complete*** if no core is idle during that step, and otherwise we say it is ***incomplete***. We say a scheduler is ***greedy*** if processors never idle whenever there is ready work available. Then, for a greedy scheduler on n_i cores, the following two straightforward lemmas are derived in [8].

Lemma 2. *Consider a greedy scheduler running on n_i cores for t time steps. If the total number of incomplete steps during this period is t^*, the total work F^t done during these time steps is at least $F^t \geq n_i t - (n_i - 1)t^*$.*

Lemma 3. *If a task τ_i is executed by a greedy scheduler, then every incomplete step reduces the remaining critical-path length of τ_i by 1.*

From Lemmas 2 and 3, we can establish Theorem 2.

Theorem 2. *If a parallel task with implicit deadline is assigned $n_i = \left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil$ dedicated cores, then it will meet all its deadlines if scheduled using any greedy scheduler.*

Proof: For contradiction, assume that some job of a high-utilization task τ_i misses its deadline when scheduled on n_i cores with a greedy scheduler. Therefore, during the D_i time steps between the release of this job and its deadline, there are fewer than L_i incomplete steps; otherwise, by Lemma 3, the job would have completed. Therefore, by Lemma 2, the scheduler must have finished at least $n_i D_i - (n_i - 1)L_i$ work.

$$\begin{aligned} & n_i D_i - (n_i - 1)L_i = n_i(D_i - L_i) + L_i \\ = & \left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil (D_i - L_i) + L_i \\ \geq & \frac{C_i - L_i}{D_i - L_i}(D_i - L_i) + L_i = C_i \end{aligned}$$

Since the task has only C_i total work to do, it must have finished within these D_i steps, leading to a contradiction. ■

E. Lower Bound on Number of Remaining Cores n_{low}

Recall that we consider any task set that satisfies Conditions 1 and 2 for $b = \frac{3+\sqrt{5}}{2}$. We first calculate the upper bound on n_{high}, the number of total cores assigned to high-utilization tasks.

Lemma 4. *For a high-utilization task τ_i, if $D_i \geq \frac{3+\sqrt{5}}{2} L_i$ (Condition 2), then number of assigned cores $n_i < \frac{3+\sqrt{5}}{2} u_i$.*

Proof: By the definition of high-utilization task τ_i, we have $1 < u_i$. Hence

$$\begin{aligned} 1 < u_i \quad & \Rightarrow \quad \left\lceil \frac{b}{b-1} u_i \right\rceil - \frac{b}{b-1} u_i < 1 < u_i \\ & \Rightarrow \quad \left\lceil \frac{b}{b-1} u_i \right\rceil < \frac{2b-1}{b-1} u_i \end{aligned}$$

Also since $bL_i \leq D_i$. Hence $D_i - L_i \geq \frac{b-1}{b} D_i$

$$\begin{aligned} n_i \quad = & \quad \left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil \leq \left\lceil \frac{C_i - L_i}{\frac{b-1}{b} D_i} \right\rceil < \left\lceil \frac{C_i}{\frac{b-1}{b} D_i} \right\rceil \\ = & \quad \left\lceil \frac{b}{b-1} u_i \right\rceil < \frac{2b-1}{b-1} u_i = \frac{3+\sqrt{5}}{2} u_i \end{aligned}$$

So $n_i < \frac{3+\sqrt{5}}{2} u_i = b u_i$. ■

Therefore, the total number of cores assigned to high utilization tasks is $n_{\text{high}} = \sum_{\text{high}} n_i < b \sum_{\text{high}} u_i$.

By Condition 1, the lower bound on the remaining number of cores in the low-utilization task cluster is: $n_{\text{low}} = m - n_{\text{high}} > b \sum_{\text{all}} u_i - b \sum_{\text{high}} u_i = b \sum_{\text{low}} u_i = \frac{3+\sqrt{5}}{2} \sum_{\text{low}} u_i$.

F. Low-Utilization Tasks Are Schedulable

Therefore, the total utilization of the cluster of low-utilization tasks is less than $m/b \approx 0.38m$. As mentioned in Section II, many multiprocessor scheduling algorithms have a utilization bound of 50% ($> 38\%$). Hence, in principle, we can use any of these scheduling algorithms to schedule the low-utilization tasks while maintaining the overall bound of $\frac{3+\sqrt{5}}{2}$. In our implementation we chose to use a multiprocessor partitioned rate monotonic scheduling algorithm (see Appendix A).

In conclusion, since each high-utilization task and the set of low-utilization tasks are all schedulable given Conditions 1 and 2, the entire task set is schedulable using our federated scheduling algorithm, thus proving Theorem 1.

G. Schedulability Analysis

The capacity augmentation bound of $\frac{3+\sqrt{5}}{2}$ functions as a simple schedulability test, since we can safely admit task sets that satisfy $\sum u_i \leq m/2.618$ and $L_i \leq D_i/2.618$ for each task τ_i, but this test is often pessimistic. Note, however, that the federated scheduling algorithm described in Section III-B can also be directly used as a (polynomial-time) schedulabily test: given a task set, after assigning cores to each high-utilization task using our algorithm, if the remain cores are sufficient for all low-utilization tasks, then the task set is schedulable and we can admit it. For now, this test ignores all system overheads. In Section V, we show how to incorporate overheads to get better real-time predictability. As we will see from experiments in Section VI, this improved test still admits many task sets with utilization greater than 38%.

IV. FSS Design

In this section, we describe the design of a scheduling service based on federated scheduling algorithm presented in Section III and how we integrated this service into a simple platform, **FSS**. This design has several benefits: (1) It allows us to use existing parallel languages and runtime systems (though not designed for real-time programs) to explore the degree of real-time performance one can achieve without implementing an entirely new parallel runtime. (2) It requires little modification to existing parallel applications. (3) It can be extended to many different parallel languages, since little modification is required to either the compiler or the runtime system of the parallel language. We instantiated it using Cilk Plus and OpenMP and performed experiments in those languages. (4) While FSS does not explicitly consider cache overheads, the scheduling policy has an inherent advantage with respect to cache locality, since no tasks migrate and each high-utilization task is assigned to a dedicated set of cores; therefore, there is no interference from other tasks.

A. Application Programming Interface (API)

FSS API makes it easy to convert existing parallel programs into parallel real-time programs. Tasks are C or C++ programs that include a header file (**`task.h`**) and conform to a simple structure: instead of a **`main`** function, a programmer specifies a **`run`** function, which is executed when a job of the task is invoked. Tasks can also specify optional **`initialize`** and **`finalize`** functions, each of which (if defined) will be called once, before the first and after the last call to the run function, respectively. These optional functions let tasks set up and clean up resources as needed.

Additionally, configuration file must be specified for the task set set, specifying runtime parameters (such as program name and arguments) and real-time parameters (such as worst case execution time, critical-path length, and period) for each task in the task set. This separate specification makes tasks flexible and easy to maintain; e.g., we do not have to recompile tasks in order to change timing constraints. More details about task and task-set specification are in Appendix B.

B. Platform Structure and Operation

FSS provides two essential capabilities — reliable real-time performance and efficient parallel execution. For low-utilization tasks, we have to worry about real-time deadlines, but we do not consider their parallel performance since they are executed sequentially. On the other hand, we only have to worry about parallel performance of high-utilization tasks, since they are scheduled on dedicated cores and do not suffer from interference from other tasks.

FSS's scheduling service first runs the federated scheduling algorithm (Section III-B), which assigns n_i-sized clusters to each high-utilization task τ_i and the remaining n_{low} cores to low-utilization tasks. Then it uses mPRM (described in Appendix A) to partition and prioritize low-utilization tasks on the cluster of n_{low} cores. The main function (provided by the scheduling service) binds each task to its assigned cores (by changing the CPU affinity mask) and sets each task's priority to the assigned real-time priority (using standard Linux system calls). It then executes the (optional) user-specified **`initialize`** functions. All tasks in the system perform an initial synchronization and start execution at their relative release times by calling **`run`** for each invocation.

The federated scheduling service separates the scheduling of different subsets of tasks using different scheduling policies, and relies on different mechanisms. In particular, low-utilization tasks are executed sequentially at specified priorities so the Linux scheduler (enhanced by CONFIG_PREEMPT_RT patch) handles scheduling and preemption automatically. High-utilization tasks execute on dedicated cores; since no other tasks interfere with them, their scheduler does not need to be deadline- or priority-aware and can use any greedy strategy. Many parallel runtime systems use greedy (e.g. OpenMP [11]) or nearly-greedy (e.g. Cilk Plus [10]) scheduling. In principle, FSS design allows programmers to specify their tasks in any such language or library and FSS simply uses that language's native runtime system. In this paper, we present instantiations using GNU OpenMP and Cilk. It would be straightforward to use other runtime systems, such as Threading Building Blocks [32]; one could even use different parallel languages' runtimes for different tasks within a single task set.

As a caveat, these runtime systems are not designed for real-time execution; therefore, theoretically, FSS does not provide hard real-time guarantees. In order to convert this platform into a system with hard-real time guarantees, we would have to implement a greedy scheduler that is specially designed for hard-real time performance for tasks written in these languages, which is beyond the scope of this paper.

C. Instantiation With OpenMP and Cilk Plus

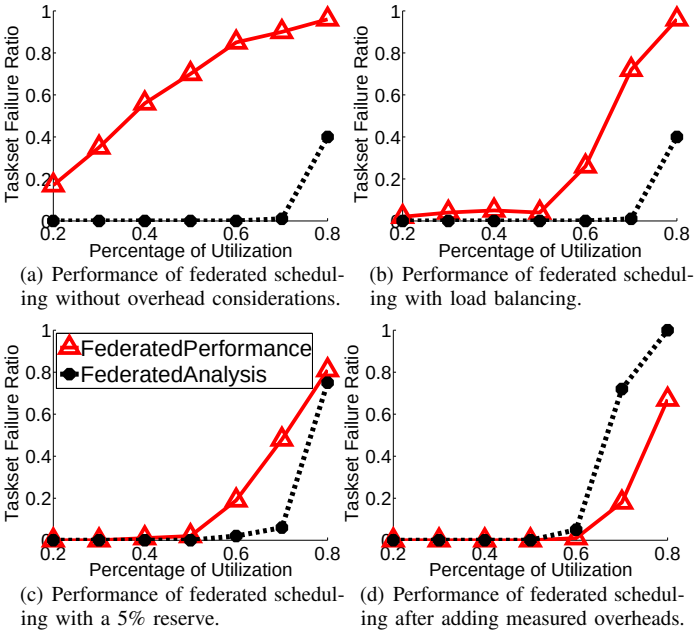
For a high-utilization task τ_i, assigned n_i cores by the 'algorithm, FSS configures Cilk Plus or OpenMP to create n_i threads for τ_i and binds one thread per assigned core.

For OpenMP, we use ***static*** thread management — where the number of threads is fixed at compile time – and allocate n_i threads to the task. OpenMP supports various scheduling policies for each of its parallelism constructs; we use the ***dynamic 1*** strategy with a centralized queue that holds all available work. When a core finishes its assigned work, it grabs another piece of work from the centralized ready queue. This strategy is a greedy strategy since a thread is never idle if ready work is available in the queue. See Appendix C for more details about OpenMP.

Cilk Plus uses a distributed scheduling policy, called ***randomized work-stealing***. Each thread maintains a local queue of ready work and takes work from this queue as needed. If a thread's local queue is empty, it randomly picks another thread (running on another core) and ***steals*** some work from its queue. Work-stealing is not a strictly greedy strategy. However, it provide strong probabilistic guarantees of linear speedup ("near-greediness") [34]. In practice, it provides very good performance [35] and variants widely used in many parallel runtimes such as Intel's TBB, IBM's X10, Microsoft's TPL. Our experiments also indicate that it scales better than OpenMP's purely greedy scheduler at large core-counts. See Appendix D for more details about Cilk Plus.

V. Overhead-Aware Federated Scheduling

The federated scheduling algorithm described in Section III does not consider any overhead (such as preemption, synchronization between parallel threads of a task and parallel scheduling overhead) of an actual system implementation. In this section, we describe how we incorporate overheads into the scheduling algorithm to get better performance and predictiveness. Figures 1 evaluates the progressive improvements of the scheduling algorithm with successive refinements.



(a) Performance of federated scheduling without overhead considerations.

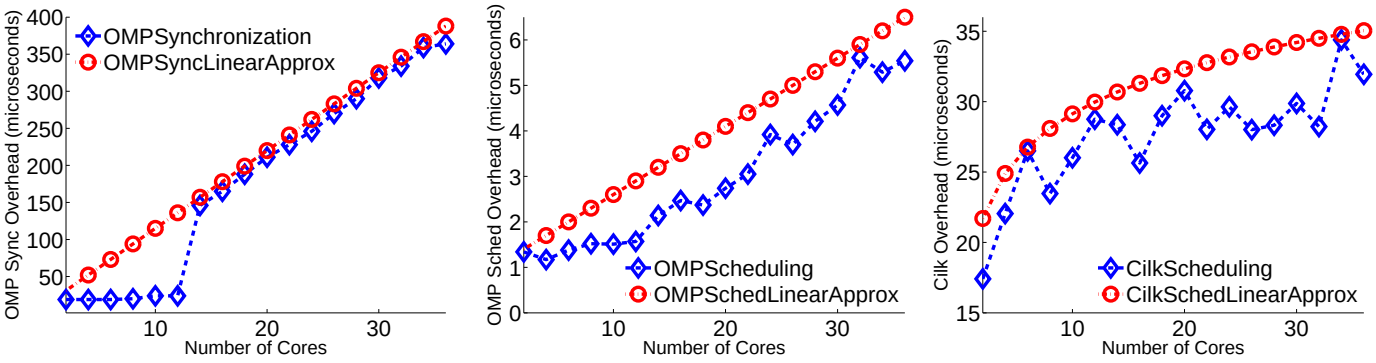
(b) Performance of federated scheduling with load balancing.

(c) Performance of federated scheduling with a 5% reserve.

(d) Performance of federated scheduling after adding measured overheads.

Fig. 1: Task set utilization vs. failure ratio on 12-core (*T120* set) as we incorporate overheads. Periods are in the range 2ms-64ms.

Figure 1(a) shows the discrepancy between the schedulability analysis and the actual performance of the platform FSS on a set of synthetic task sets (see task sets description in Section VI-B). when the algorithm does not consider any overheads. The y-axis shows the failure ratio (the number



(a) OMP Synchronization Overhead. (b) OMP Scheduling Overhead. (c) Cilk Overall Overhead.

Fig. 2: Overhead Measurement and Estimation.

of task sets that have any missed deadlines divided by the total number of task sets). Note that the theoretical analysis is very optimistic; it predicts that almost all task sets even at 70% total utilization are schedulable using the federated algorithm. However, when they are executed on FSS, many of them have deadline misses due to overheads.

Many deadline misses are caused due to an artifact of mPRM bin-packing, which overloads some cores while it under-loads others. Intuitively, a **load-balancing heuristic** among low-utilization task sets may be applied, resulting in Figure 1(b). Note that this heuristic does not change the results of the theoretical schedulability test (the dashed black line remains unchanged), but does improve the performance of FSS (the solid red line).

An important source of inaccuracy in the analysis is that it assumes that 100% of the processing power can be used for task execution, which is unrealistic on a real platform. Therefore, in Figure 1(c), **a reserve of 5% utilization** for each core is applied, so that each core can only be utilized up to 95%. In this case, the results of the scheduling algorithm (and schedulability test) change. For example, tasks with utilization over 95% are now considered high-utilization tasks, and are allocated 2 cores instead of 1. Therefore, fewer task sets are considered schedulable compared to Figure 1(a).

Note that as the scheduling algorithm includes more overheads, we see both the ***accuracy*** and ***performance*** of scheduling improve. That is, not only does the analysis (dashed black line in the graph) become more realistic, but more interestingly, the real-time performance of FSS (solid red lines in the graph) improves as well. However, load-balancing and 5% reserve heuristics are still insufficient, since, in Figure 1(c), many task sets that the algorithm considers schedulable still miss deadlines. Therefore, we must develop more sophisticated overhead estimates and incorporate them into our scheduling algorithm to enhance real-time performance, as we now discuss in Sections V-A and V-B. Figure 1(d) indicates that the resulted algorithm has high predictability and better performance since it schedules more task sets than the previous attempts.

A. Precise Overhead Measurements

From now on, we will focus on ***parallel synchronous tasks*** — tasks that consists of a sequence of parallel-for loops. While our theory holds for general DAG tasks (parallel synchronous tasks are a subset of DAGs), DAGs have a more complex structure and a larger number of scheduling uncertainties; therefore characterizing the overheads for general DAG tasks is a more difficult problem. We defer the (brief) discussion the overhead measurement of DAGs briefly to Appendix G. Restricting our attention to synchronous tasks allows us to do a more precise characterization of overheads to improve the scheduling algorithm. In addition, parallel synchronous model represents a large subset of useful parallel programs.

All of our measurements are made on two machines, namely *Austen* — a 4-socket 48-core AMD machine (we use 36 of the cores for experiments), and *Dickens* — a 2-socket 16-core Intel machine (we use 14 cores for experiments). More details about the machines is provided in Section VI. There are two main sources of overhead in the platform: preemption overhead and overhead of parallel runtime systems.

Preemption overhead: We used an open-source tool, Cyclictest [36], to measure the preemption latency. Over 10,000 iterations the maximum preemption overheads, denoted as p, on Dickens and Austen were 27ms and 164ms respectively.

Parallel Runtime Overheads: Recall that we use Cilk Plus or OpenMP to schedule high-utilization tasks. Their overheads generally increase with the number of cores allocated to a task. Therefore, if a task is schedulable on 4 cores, it is not necessarily schedulable on 5 cores since the increase in overheads may eat up the gain from one more core. Hence, it is necessary to understand the relationship between the overheads and the number of cores on each individual platform. In addition, OpenMP and Cilk Plus have different scheduling policies, so that the effect of these overheads is different.

The main sources of overhead in OpenMP are: (1) ***parallel scheduling overhead*** due to OpenMP's `dynamic 1` scheduling — after each iteration, a thread must perform a synchronized access to the job's global work queue to get more work; and (2) ***barrier synchronization overhead*** due to implicit synchronization at the end of each parallel-for loop. We used an open-source micro-benchmarks (EPCC [37]) to measure these overheads. Figures 2(a) and 2(b) show the maximum overheads of each scheduling and synchronization operation on our AMD machine *Austen* (on *Dickens*, the values are different, but the trends are similar). The top line shows a linear approximation of the measured trend. For scalability, instead of

hard coding each value, we linearly approximate them by the formula $b_0 + b_1 * n_i$, where b_0 and b_1 are constants obtained from the linear approximation. Similarly, we can denote the OpenMP scheduling overhead as $s_0 + s_1 * n_i$. These constants are different for different machines; in order to conduct the schedulability test on a different machine, we would simply run the microbenchmark on that machine, measure the overheads, and again derive these constants using a linear approximation. A brief explanation of why the overheads increase linearly is provided in Appendix E.

We also modified EPCC to measure the overhead of Cilk Plus. Due to the very different operation of Cilk Plus scheduler, we do not distinguish between scheduling and synchronization overheads and only conduct an overall measurement, the results of which are shown in Figure 2(c). Note that both theoretically and practically, for Cilk the scheduling overhead has logarithmic growth in the number of cores. Again, we fit a curve of the form $c_0 + c_1 \lg n_i$ to the measured overhead and use this in the scheduling algorithm. An explanation for this logarithmic relationship is given in the Appendix E.

B. Overhead Calculation

We now describe how the overhead estimates from the previous section are incorporated into the scheduling algorithm.

Low-utilization tasks: Recall that tasks with lower priority will be preempted by high priority ones. The maximum number of preemptions that can happen while a task τ_i is executing is $q_i = 2\sum_j \left\lceil \frac{D_i}{D_j} \right\rceil$, where task τ_j is any task that has higher priority than τ_i. Therefore, the maximum preemption overhead for τ_i is pq_i, where p is the measured preemption overhead. We simply add this quantity to the worst case execution time of τ_i, in addition to the 5% reserve.

High-utilization tasks: Whether a high-utilization task τ_i can meet its deadline is determined by the slack S_i. If the slack is large enough to absorb the overhead O_i, then the task is schedulable, and otherwise is not. Hence, in the scheduling algorithm, we add four steps:(1) calculate slack S_i; (2) calculate overhead O_i using the estimates from the above measurements; (3) check if $S_i > O_i$ (if so, then the task is schedulable, and otherwise is not); and (4) if the slack $S_i \leq O_i$, then we increase the number of cores assigned to the task until the slack is smaller than the overheads. A task is considered unschedulable if we can not find any such core count.

The **slack** is the ideal remaining time between a parallel task's finish time and its deadline, formally defined as $S_i = D_i - L_i - (C_i - L_i)/(n_i * 0.95)$ (the 0.95 term is due to the 5% reserve). Theoretically, as long as the slack is not negative, the task is schedulable under a greedy schedular. However, overhead will occupy some additional time. Unless the slack is more than the sum of all overheads, it is possible that in the worst case the task will miss deadlines. Now we can calculate the overhead O_i, by adding the contributions from the following components:

1) Overhead for invocation of each run of a task by the OS, which is equal to the preemption overhead p.
2) For a synchronous task (written in OpenMP), the total number of synchronization operations equals to the number of segments (for-loops) g_i in the task; so the total barrier synchronization overhead for each task is $(b_0 + b_1 * n_i) * g_i$.
3) The number of scheduling operations (for both Cilk Plus and OpenMP) is the total number of iterations of all segments divided by n_i. We assume that we do not know the task structure in this much detail, and approximate the number of scheduling operations as $g_i * (C_i/L_i/n_i)$, where C_i/L_i represents the degree of parallelism. Therefore, for OpenMP, the total scheduling overhead is $(s_0 + s_1 * n_i) * g_i * (C_i/L_i/n_i)$, and for Cilk, it is $(c_0 + c_1 * \lg n_i) * g_i * (C_i/L_i/n_i)$.

VI. Evaluation

In this section, we describe our empirical evaluation of FSS using randomly generated tasks in both Cilk Plus and OpenMP. We find that the federated scheduling algorithm provides good real-time performance. In addition, we find that FSS outperforms the only prior openly available parallel real-time platform [9].

A. Experimental Machines

Experiments were conducted on a 48-core machine (called ***Austen***) composed of four AMD Opteron 6168 processors and a 16-core machine (called ***Dickens***) composed of two Intel Xeon E5-2687W processors. On *Austen*, we reserved one processor for system tasks, leaving 36 experimental processing cores. We ran single socket 12-core experiments and multi-socket 36-core experiments on *Austen*. On *Dickens*, we reserved two cores, leaving 14 experimental cores. Linux applied with CONFIG_PREEMPT_RT was the underlying RTOS.

B. Task Set Generation

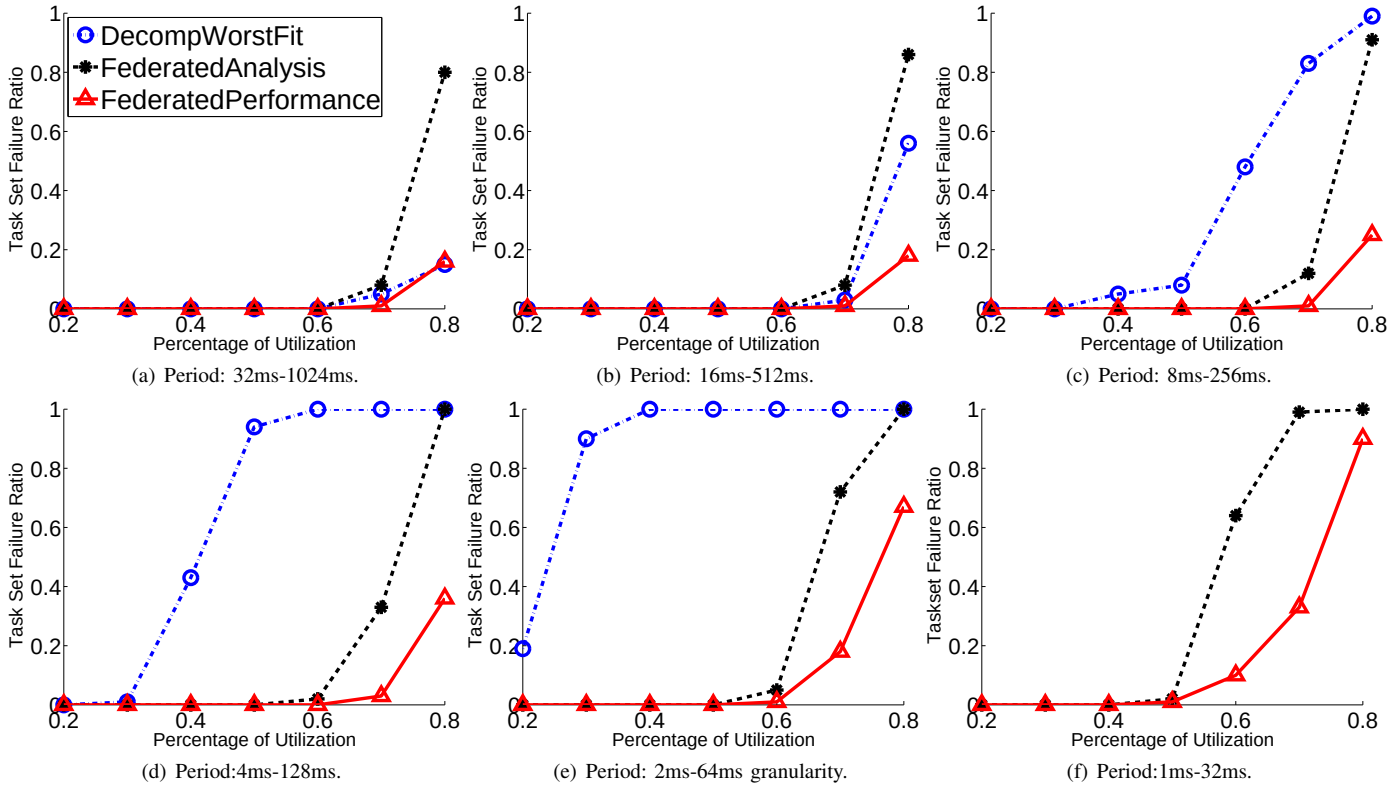
We ran our experiments on synchronous tasks consisting of sequences of parallel for-loops of varying lengths and numbers of iterations implemented as regular OpenMP and CilkPlus programs. We ran 4 categories of task sets (shown in Table I), 1 using 12 cores, 1 using 14 cores and 2 using 36 cores. *T12O* and *T36O* are 12 and 36-core benchmarks used in [9],[1] which allows for direct comparison between this and previous work. Since we used exactly the same task sets, we only provide a brief description in Appendix F. The only important detail is that for each task τ_i, its period is confined by constraint $D_i \geq 5L_i$ in *T12O* and *T36O*, which was required by the prior system for theoretical schedulability.

Name	#Total Cores	Average Task Utilization	Average #Tasks per Taskset	Machine
T12O	12	0.35	16.72	Austen
T14N	14	1.76	3.74	Dickens
T36O	36	2.47	7.04	Austen
T36N	36	5.41	3.32	Austen

TABLE I: Taskset Characteristics

In addition, we wanted to stress-test our system using tasks with higher parallelism. As can be seen from Table I, if we used the same settings for (previous) tasksets *T12O* and *T36O*, then most task sets would have no task with utilization more than 4. Therefore, we used a different set of parameters to

[1]The previous system cannot run OpenMP programs directly, so the API used to write tasks in the two systems is different, but the task structure and the parameters are identical.



(a) Period: 32ms-1024ms. (b) Period: 16ms-512ms. (c) Period: 8ms-256ms.

(d) Period:4ms-128ms. (e) Period: 2ms-64ms granularity. (f) Period:1ms-32ms.

Fig. 3: Task Set Utilization vs. Failure Rate (both in percentages) for 12-Core *T12O* Task Sets with Different Task Granularities. At the largest granularity (lowest frequency) decomposition-based scheduling and FSS perform comparably. As the frequency increases, while both get worse, but FSS's performance degrades much more slowly. At the second highest frequency, `DecompWorstFit` can not schedule most task sets, while FSS can still schedule most task sets up to 60% utilization.

generate tasks with greater parallelism. Therefore, for new 14- and 36-core task sets, called *T14N* and *T36N*, we changed two characteristics: (1) *increased parallelism* by increasing the mean number of iterations per parallel-for loop; and (2) *decreased slack* by relaxing the constraint, so that the deadline (and period) satisfies $D_i \geq 2L_i$.

C. Experimental Settings

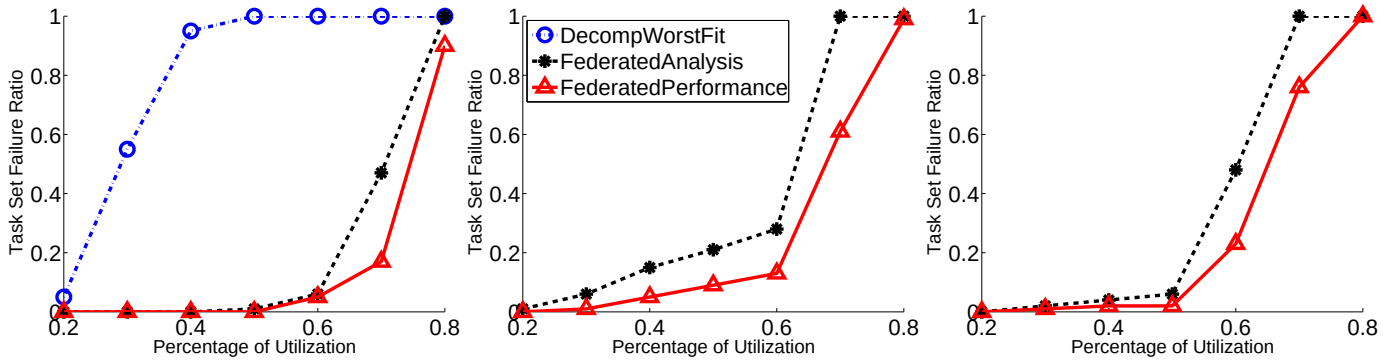
We wished to understand the effects of both ***total utilization*** and ***frequencies***. Total utilization of a task set indicates how loaded the machine is when running that task set. For each category, we generate 100 task sets at utilizations of $20, 30, ..., 80$ percent of m. For example, the total utilization of a 50% task set on a 36 core machine is approximately 18. For all experiments, each task set was run for 1,000 hyper-periods.

Frequency (or granularity) dictates how often a task has to run. In many applications such as hybrid testing [1] or route planning [2], it is important to enable high-frequency execution, since it leads to better accuracy and/or stability. However, high frequency tasks are more difficult to schedule in practice, since the frequency dictates the amount of slack available. For instance, consider two tasks with identical structures: τ_1 has a period of 1 ms, critical path length of 0.5 ms and total work (worst case execution requirement) of 2 ms, and τ_2 has a period of 10 ms, critical path length of 0.5 ms and total work of 20 ms. In theory, these tasks are equivalent; if one is schedulable, so is the other. On a real platform, however, τ_1 is much harder to schedule, since it has a smaller slack to absorb overheads. In order to evaluate the effect of frequencies on a practical system, we scale the task sets by multiplying or dividing the execution time and period of each task by a factor of 2 (i.e., halving or doubling the frequency).

We first run the scheduling algorithm (which is also a schedulability test) on each task set. The output is whether or not the task set is schedulable under the test and if so, it returns a valid schedule. Even if a schedulability test deem a task set unschedulable, we still run it on FSS; however, in this case, the schedule we use does not provide theoretical schedulability guarantees. ***Failure ratio*** is the number of failed task sets over the total number of task sets. A task set ***fails*** schedulability analysis if the algorithm deems that task set unschedulable. A task set ***fails*** on FSS if during execution of the task set, *any job* misses any deadline. Failure ratio from schedulability analysis as well as the measured execution results on FSS are presented.

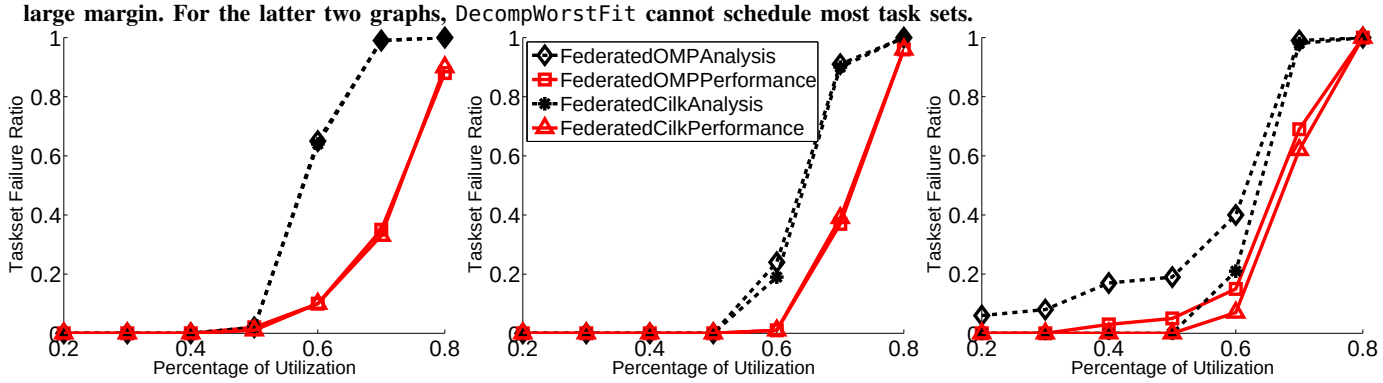
D. Federated vs. Decomposition-based Scheduling

To provide a direct comparison between this and previous work, we used exactly the same task sets, run on the same machine (*Austen*) with same system setup. Since Cilk Plus and OpenMP have nearly identical performance on *T12O* task sets (see Section VI-E) we only present the Cilk Plus results. The failure ratio of the analysis is indicated with the line `FederatedAnalysis` and the failure ratio of FSS is shown with line `FederatedPerformance`. We compare these against our baseline — previous results from using task



(a) Period: 8ms-256ms on *T36O*. (b) Period: 2ms-64ms on *T36O*. (c) Period: 2ms-64ms *T36N*.

Fig. 4: Task Set Utilization vs. Failure Rate (both in percentages) for 36-Core Task Sets. FSS outperforms `DecompWorstFit` by a large margin. For the latter two graphs, `DecompWorstFit` cannot schedule most task sets.



(a) Period: 1ms-32ms on *T12O*. (b) Period: 1ms-32ms on *T14N*. (c) Period: 4ms-128ms on *T36N*.

Fig. 5: Task Set Utilization vs. Failure Rate for Comparison between Cilk Plus and OpenMP Task Sets. On 12 and 14 core experiments, they perform comparably. On 36 core experiments, Cilk Plus performs better.

decomposition scheduling with worst-fit bin-packing from [9] (marked with `DecompWorstFit`).[2] We only present 12 and 36 cores experiments, but 14-core ones show similar trends.

1) Experimental Results on 12 Cores: The results of running tasks of various utilizations at different granularities are shown in Figures 3. The first observation is that the `Federated-Performance` always dominated the previous platform using task decomposition. Moreover, as the granularities decrease (frequencies increase), the performance difference between `FederatedPerformance` and `DecompWorstFit` increases dramatically. Even at high frequencies of 1000 hertz, FSS schedules most tasks at 60% utilization and almost all at 50% utilization, while decomposition based scheduling has an unacceptably high failure ratio even at 500 hertz frequency.

One major reason of such performance differences is that in our federated scheduling algorithm, tasks are given the minimum number of cores for it to be schedulable, while in baseline system for each task there will be a generated thread working on each core. Hence synchronization and scheduling overheads are much smaller on our platform.

As expected, due to decreasing slack, FSS's failure ratio increases with increasing frequencies. However, the analysis successfully captures this effect and deems fewer task sets schedulable, demonstrating its predictability. Even though we use existing parallel runtimes that are not designed for real-time execution, our platform nevertheless provides good real-time performance.

2) Experimental Results on 36 Cores: Experimental results on 36 cores using both *T36O* and *T36N* are shown in Figure 4. The dotted line for decomposition based scheduling is missing from the latter figures because these task sets are completely unschedulable (even in theory) using the decomposition based scheduler. Again, we see that the federated scheduling analysis is predictive of the performance on the real system. The difference between decomposition-based scheduling and federated scheduling is even more pronounced, as seen in Figure 4(a); while baseline platform cannot even schedule most task sets at 30% utilization, our platfrom schedules majority of task sets up to 70% utilization. Results are similar at other granularities, which we do not show due to space constraints. (3) Finally, Figures 4(a) and 4(b) illustrates that increasing frequencies makes task sets more difficult to schedule.

We have a couple of additional observations. (1) If we compare the experiments of the same frequency on 12 and 36 cores (shown in Figures 4(a) and 3(c)) on the same machine, we see that while both provide worst performance with increasing m, federated scheduling is more scalable than decomposition based scheduling. Note that increasing the number of cores also increases total utilization; a 50% utilization task set has a total utilization of 6 on 12 cores and 18 on 36 cores. (2) In order to stress test our system, we ran task sets from the category *T36N*, which have higher parallelism. Figures 4(b)

[2]We do not show the results from their first-fit bin packing heuristic, since worst-fit was always better in the previous experiments.

and 4(c) show results at the same granularity using low and high-parallelism task sets. Surprisingly, the high parallelism task sets perform slightly better. One possible reason is that we are using a very efficient and scalable runtime system from Cilk Plus that is targetted at high-parallelism tasks.

E. Cilk Plus vs. OpenMP

Figure 5 shows the comparison between performances of Clik Plus and OpenMP programs within FSS. We use *T12O*, *T14N* and *T36N* for the three different processor configurations, respectively. For 12 and 14 core experiments, the two runtime systems behave comparably. However, Cilk Plus performs better on 36 core experiments, both in terms of the analysis and the real performance on FSS. This is not surprising since *T36N* task sets have much higher degree of parallelism and therefore individual tasks are assigned larger numbers of cores. We saw in Section V that while the overhead of OpenMP grows linearly with the number of cores, Cilk Plus overhead only has logarithmic growth. These experiments indicate that, even though in theory the work stealing scheduler in Cilk Plus is not entirely a gready scheduler and the overhead estimation is imprecise due to this reason, in real execution, the Cilk Plus runtime can actually perform as well as or better than OpenMP.

VII. Conclusions

We have presented a novel federated approach to schedule parallel real-time tasks. This approach provides the best-known theoretical capacity augmentation bound of $\frac{3+\sqrt{5}}{2}$and naturally leads to a simple implementation using existing parallel languages such as OpenMP and Cilk Plus. We also augmented the theoretical scheduling with overhead estimation to get an algorithm with better real-time performance and predictive power. There are several avenues of future work. First, our current overhead measurements do not consider the effect of cache contention, and incorporating these measurements enhance the practicality of the platform. Second, while our theory admits DAG tasks, our overhead estimation and experiments are primarily on parallel synchronous tasks. Current overhead estimation for DAGs are imprecise and pessimistic. Finally, implementing a greedy scheduler for real-time execution would allow us to guarantee hard real-time platform.

References

[1] H.-M. Huang, T. Tidwell, C. Gill, C. Lu, X. Gao, and S. Dyke, "Cyber-physical systems for real-time hybrid structural testing: a case study," in *ICCPS '10*.

[2] J. Kim, H. Kim, K. Lakashmanan, and R. Rajkumar, "Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car," in *ICCPS '13*.

[3] A. Saifullah, K. Agrawal, C. Lu, and C. Gill, "Multi-core real-time scheduling for generalized parallel task models," in *RTSS '11*.

[4] K. Lakshmanan, S. Kato, and R. R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *RTSS '10*.

[5] C. Liu and J. Anderson, "Supporting soft real-time parallel applications on multicore processors," in *RTCSA '12*.

[6] S. Baruah, V. Bonifaciy, A. Marchetti-Spaccamelaz, L. Stougiex, and A. Wiese, "A generalized parallel task model for recurrent real-time processes," in *RTSS '12*.

[7] L. Becchetti, M. Dirnberger, A. Karrenbauer, and K. Mehlhorn, "Feasibility analysis in the sporadic dag task model," in *ECRTS '13*.

[8] J. Li, K. Agrawal, C.Lu, and C. Gill, "Analysis of global edf for parallel tasks," in *ECRTS '13*.

[9] D. Ferry, J. Li, M. Mahadevan, K. Agrawal, C. Gill, and C. Lu, "A real-time scheduling service for parallel tasks," in *RTAS '13*.

[10] "Intel CilkPlus." http://software.intel.com/en-us/articles/intel-cilk-plus.

[11] "OpenMP Application Program Interface v3.1," July 2011. http://www.openmp.org/mp-documents/OpenMP3.1.pdf.

[12] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comp. Surv.*, vol. 43, pp. 35:1–44, 2011.

[13] M. Bertogna and S. Baruah, "Tests for global edf schedulability analysis," *J. Syst. Archit.*, vol. 57, no. 5, pp. 487–497, 2011.

[14] B. Andersson, S. Baruah, and J. Jonsson, "Static-priority scheduling on multiprocessors," in *RTSS '2001*, pp. 193–202, dec. 2001.

[15] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, and S. Stiller, "Improved multiprocessor global schedulability analysis," *Real-Time Syst.*, vol. 46, pp. 3–24, Sept. 2010.

[16] J. M. López, J. L. Díaz, and D. F. García, "Utilization bounds for edf scheduling on real-time multiprocessor systems," *Real-Time Syst.*, vol. 28, pp. 39–68, Oct. 2004.

[17] B. Andersson and J. Jonsson, "The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%," in *Real-Time Systems, 2003. Proceedings. 15th Euromicro Conference on*, pp. 33–40, 2003.

[18] A. Gujarati, F. Cerqueira, and B. Brandenburg, "Schedulability analysis of the linux push and pull scheduler with arbitrary processor affinities," in *ECRTS '13*.

[19] "CONFIG_PREEMPT_RT Patch." https://rt.wiki.kernel.org/index.php/Main_Page.

[20] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson, "LitmusRT: A testbed for empirically comparing real-time multiprocessor schedulers," in *RTSS '06*.

[21] B. B. Brandenburg, *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2011.

[22] M. S. Mollison and J. H. Anderson, "Bringing theory into practice: A userspace library for multicore real-time scheduling," in *RTAS '13*.

[23] J. Lelli, D. Faggioli, T. Cucinotta, and G. Lipari, "An experimental comparison of different real-time schedulers on multicore systems," *J. Syst. Softw.*, vol. 85, pp. 2405–2416, Oct. 2012.

[24] W. Y. Lee and H. Lee, "Optimal scheduling for real-time parallel tasks," *IEICE Trans. Inf. Syst.*, vol. E89-D, no. 6, pp. 1962–1966, 2006.

[25] S. Collette, L. Cucu, and J. Goossens, "Integrating job parallelism in real-time scheduling theory," *Inf. Process. Lett.*, vol. 106, no. 5, pp. 180–187, 2008.

[26] G. Manimaran, C. S. R. Murthy, and K. Ramamritham, "A new approach for scheduling of parallelizable tasks in real-time multiprocessor systems," *Real-Time Syst.*, vol. 15, no. 1, pp. 39–60, 1998.

[27] S. Kato and Y. Ishikawa, "Gang EDF scheduling of parallel task systems," in *RTSS '09*.

[28] G. Nelissen, V. Berten, J. Goossens, and D. Milojevic, "Techniques optimizing the number of processors to schedule multi-threaded tasks," in *ECRTS '12*.

[29] H. S. Chwa, J. Lee, K.-M. Phan, A. Easwaran, and I. Shin, "Global edf schedulability analysis for synchronous parallel tasks on multicore platforms," in *ECRTS '13*.

[30] L. Nogueira and L. M. Pinho, "Server-based scheduling of parallel real-time tasks," in *International Conference on Embedded Software*, 2012.

[31] B. Andersson and D. de Niz, "Analyzing global-edf for multiprocessor scheduling of parallel tasks," in *Principles of Distributed Systems*, pp. 16–30, 2012.

[32] J. Reinders, *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, 2010.

[33] K. Agrawal, C. E. Leiserson, Y. He, and W. J. Hsu, "Adaptive work-stealing with parallelism feedback," *ACM Trans. Comput. Syst.*, vol. 26, pp. 112–120, September 2008.

[34] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, 1999.

[35] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," (Santa Barbara, California), pp. 207–216, July 1995.

[36] "Real-time linux wiki. cyclictest - RTwiki." https://rt.wiki.kernel.org/index.php/Cyclictest.

[37] J. M. Bull, "Measuring synchronisation and scheduling overheads in openmp," in *Proceedings of First European Workshop on OpenMP*, vol. 8, p. 49, 1999.

APPENDIX

A. Scheduling Low Utilization Tasks

As mentioned in Section III, we can use any multiprocessor scheduling algorithm with utilization bound more than $1/2.618$ to schedule the low-utilization tasks. Here we briefly review the multiprocessor partitioned rate monotonic (mPRM) approach used in FSS. Note that other multiprocessor schedulers such as mGEDF and mPEDF can be used as well. We used the schedulability test called R-BOUND-MP-NFR from [17], which employs a per-core rate monotonic schedulability test.

Theorem 3. *(From [17]) Let* $B(r_q, t_p) = t_p(r_p^{1/t_p} - 1) + 2/r_p - 1$*, where* r_p *is the ratio between the maximum and minimum period among all tasks assigned to core* p*, and* t_p *is the number of tasks on core* p*. If* $\sum_{i=1}^{t_p} C_i/P_i \leq B(r_q, t_p)$ *and rate monotonic scheduling is used, then all deadlines are met on core* p*.*

We can use the R-BOUND-MP-NFR scheduling algorithm and schedulability test to assign low-utilization tasks on n_{low} cores. First, we sort tasks according their periods from shortest to longest, and then use Theorem 3 as a schedulability test on each processor, assigning tasks with the next-fit bin-packing algorithm. If we run out of cores and some tasks are still unassigned, then declare the task set unschedulable. This schedulability test guarantees that as long as the sum of the utilizations of all low-utilization tasks is at most $n_{\text{low}}/2$, then all low-utilization tasks will be successfully scheduled.

B. Specification of Programming Interface (API)

```
#include <omp.h>
#include "task.h"
int init(int argc, char *argv[]) {
        //Initialize the task
}
int run(int argc, char *argv[]) {
        //Arbitrary parallel code
}
int finalize(int argc, char *argv[]) {
        //Clean up after the task
}
task_t task = { init, run, finalize };
```

Fig. 6: Task Program Format.

```
SystemFirstCore  SystemLastCore
Task1ProgramName  Task1Arg1  Task1Arg2 ...
Task1: WorstCaseExecutionTime CriticalPathLength
      Period RelativeReleaseTime NumIterations
...
TasknProgramName  TasknArg1  TasknArg2 ...
Taskn: WorstCaseExecutionTime CriticalPathLength
      Period RelativeReleaseTime NumIterations
```

Fig. 7: Format of the Configuration File

C. OpenMP Overview

OpenMP is an application programming interface (API) standard [11] (for C, C++, and FORTRAN) that is published by the OpenMP Architecture Review Board, a group of industrial organizations that create OpenMP products. The API is widely supported by many compilers. Like other parallel environments, OpenMP allows a programmer to specify where parallelism can be exploited and actual parallel scheduling and execution is done by the OpenMP runtime library. Parallelism is specified through compiler **`pragma`** statements. For example, in OpenMP, a for-loop is specified by a parallel for-loop statement with **`#pragma omp parallel for`**. In the **`parallel-for`** loop each iteration can be executed concurrently, if processing capacity is available. An OpenMP program where only this idiom is used generates a parallel synchronous program. OpenMP also supports other types of parallel idioms, allowing more general parallel programs (such as DAG tasks).

In OpenMP each parallel section is assigned a team of threads, that is responsible for executing that parallel section. OpenMP can do ***static*** thread management, where the number of threads available to each parallel region is fixed at compile time, or dynamic thread management where they are not. OpenMP supports various scheduling policies for each of its parallelism constructs. Here we focus on explaining the policies used for the parallel-for construct, since that is the one used in our experiments. In ***dynamic N*** scheduling (where N can be any integer), each thread (and core in our case, since we create one thread for each allocated core) is initially assigned N iterations of a loop, and it grabs the next N unclaimed iterations when finished. In ***static N*** scheduling, all iterations are divided among the threads at the start of a loop, in chunks of size N. In ***guided N*** scheduling, the size of each chunk is proportional to the number of unclaimed iterations divided by the number of threads, for a minimum chunk size of N. Static scheduling has the lower overhead than dynamic scheduling. In OpenMP, *dynamic 1* is the only greedy scheduler, though it has highest and potentially unbounded overhead. In contrast, other policies are not greedy, since they might leave a core idle even if there is available work.

D. Cilk Plus Overview

Cilk Plus, as part of the Cilk family of parallel languages, is an extension to C++ for parallel programs. Its most relevant feature is its work-stealing scheduler, in which work dispatching (or load-balancing) is executed dynamically, rather than statically. Scheduling are done in a distirbuted manner, which results in scalability and lower overhead.

If a job is assigned n cores, the Cilk Plus runtime system creates n worker threads for the task. Each worker thread maintains a local double-ended queue, called its ***deque***. When a worker generates new work (enables a ready node from the job's DAG), it pushes the work to the bottom of its local queue. When a worker finishes its current node, it pops a ready node from the bottom of its local queue. But if the local queue is empty, the worker thread becomes a ***thief*** and picks a ***victim*** thread uniformly at random among the other threads $n - 1$ working on the same task and tries to steal work from the of the victim's deque. Note that most of the time, workers work off their own queues and don't need to communicate with each other at all. Therefore, this ***work-stealing strategy*** is very effective in practice and the amount of scheduling and synchronization overhead is small.

We now briefly explain how parallel-for loops are executed in Cilk Plus using work-stealing. Within the compiler, a parallel-for loop is transformed so that the iterations of the loop with k are "spawned" in a balanced binary tree with k leaves. The leaves of the tree represent the actual work within each

iteration. The internal nodes serve as dummy nodes purely for control purposes. The root of the tree represents all k iterations. Thereafter, each child represents half the iterations of its parent node. When a parallel-for loop begins execution, one worker has the entire tree and executes the root node. All other workers are trying to steal. During execution, when a core executes an internal node, it puts the right child on its deque and continues to execute the left child. This right child is available for stealing from other workers.

Theoretically, Blumofe and Leiserson [34] proved that given a job with work C_i and critical-path length L_i, work-stealing will complete this job in time $C_i/n_i + O(L_i + \lg n_i + \lg 1/\epsilon)$ time on n_i cores with probability $1-\epsilon$ for any $0 < \epsilon < 1$. Hence, work-stealing is provably efficient with high-probability. However, in very rare cases, it can perform badly. Therefore, if we use work-stealing in the real-time context, we can not expect hard real-time guarantees; nevertheless, due to the high-probability guarantees, the bad case occurs extremely rarely.

E. OpenMP and Cilk Overheads

We now give brief explanations for some basic reasoning behind the overhead measurement and estimation methodology explained in Section V. Recall that FSS allocates n_i threads to task τ_i, where n_i is the number of allocated cores. We explain why OpenMP overheads increase linearly with increasing n_i while Cilk Plus overheads increase logarithmically.

Linear increase in OpenMP scheduling and barrier synchronization overheads: OpenMP scheduling overheads are due to the fact that there is a centralized queue of available work and all threads must get their work from this queue. In the worst case, each thread accesses this queue at the same time. In this case, getting each piece of work can take up to $\Theta(n_i)$ time, leading to linear dependence on n_i, the number of cores assigned to the task. OpenMP barrier synchronization overheads are due to an implicit synchronization at the end of each parallel-for loop. With a global barrier synchronization, threads need to access the barrier sequentially. Therefore, again, in the worst case, all of the threads arrive at the same time, so one thread will have to wait for all the other threads finish their operation on the barrier. Therefore, each thread has the overhead of $\Theta(n_i)$ per synchronization operation.

Logarithmic increase in Cilk Plus scheduling overheads: Note that for Cilk Plus, steals and steal-attempts are the major source of overhead. If we look at how parallel-for loops are executed in Cilk, it is not difficult to see that a loop with k iterations is expected to generate $\Theta(\min\{n_i, k\} \lg n_i)$ steal attempts. Therefore, the number of steal attempts per iteration is $\Theta(\lg n_i)$ in the worst case.

The number of scheduling and synchronization operations: OpenMP has an implicit barrier at the end of each parallel-for loop; therefore the number of those operations is g_i, the number of segments in the program. Therefore, to calculate the total synchronization overhead for task τ_i, we multiply the estimated synchronization overhead with g_i. The scheduling overhead for both Cilk Plus and OpenMP is calculated per-iteration. We estimate the average number of iterations per loop as C_i/L_i, the average parallelism of the task. Therefore, the total number of iterations is $g_i C_i/L_i$. Therefore, to get the amount of time spent on scheduling operations, we multiply the estimated scheduling overhead with $g_i C_i/L_i$. However, since this time is divided over n_i cores, this quantity is further divided by n_i to get the scheduling overhead of task τ_i.

F. Experimental Task Set Generation

Here, we describe how we randomly generate task sets for our empirical evaluation. For comparison with previous system, we use the same task sets *T12O* and *T36O* (see description in Table I). The generation steps are: for each task, we first randomly select its period (and deadline) so that the period was at least five times the critical-path length.[3]; then for each for-loop, the number of iterations was chosen from a log-normal distribution with a mean of 4; for-loops are added to the task until adding another for-loop would make its critical-path length longer than desired. For each utilization level, starting from zero task instead of previous task sets, tasks are successively added into task sets until the utilization is within 2% of a desired utilization level. For new task sets with higher parallelism tasks, we changed two parameters: now the period only needs to be twice the critical-path length and the mean number of subtasks (iterations) per segment is 40 instead of 4.

G. Overhead Estimation for DAG Tasks

The number of synchronization and scheduling instances for DAG tasks is hard to calculated without pessimism. In principle, we could count the number of all possible instance. However, it requires the exact task structure and a deterministic online execution, which are hard to get. Alternatively, to measure overhead, one can run each task numerous times using different numbers of cores. Per task overhead can be measured, which is feasible for each individual task set, but becomes task profiling and is out of the scope of our paper. For scalability, we still want to estimate overhead using the formular $s_0 + s_1 n_i$ (linear to the number of cores), so we run all tasks with different n_i and fed the measured overhead into the formula and get the constants s_0 and s_1. Since synchronous task model is one type of general DAGs, for which we know a more precise overhead estimation, so we run experiments to compare their pessimism in Figure 8. Unlike using fine grain overhead estimates, using coarse grain estimates is indeed more pessimistic,

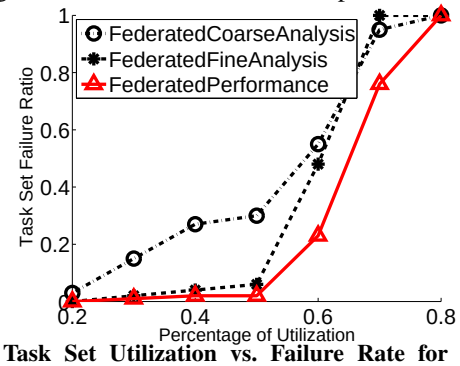


Fig. 8: Task Set Utilization vs. Failure Rate for Comparison between Coarse and Fine Grain Overhead Estimation, Period: 2ms-64ms *T36N*.

[3]This is the condition that was required for deriving the capacity augmentation bound of 5 for the decomposition-based scheduling in [9].