2016 IEEE Real-Time Systems Symposium

Randomized Work Stealing for Large Scale Soft Real-time Systems

Jing Li, Son Dinh, Kevin Kieselbach,
Kunal Agrawal, Christopher Gill, and Chenyang Lu
Washington University in St. Louis
{*li.jing, sonndinh, kevin.kieselbach,kunal, cdgill, lu*}*@wustl.edu*

***Abstract*—Recent years have witnessed the convergence of two important trends in real-time systems: growing computational demand of applications and the adoption of processors with more cores. As real-time applications now need to exploit parallelism to meet their real-time requirements, they face a new challenge of scaling up computations on a large number of cores. Randomized work stealing has been adopted as a highly scalable scheduling approach for general-purpose computing. In work stealing, each core steals work from a randomly chosen core in a decentralized manner. Compared to centralized greedy schedulers, work stealing may seem unsuitable for real-time computing due to the non-predictable nature of random stealing. Surprisingly, our experiments with benchmark programs found that random work stealing (in Cilk Plus) delivers tighter distributions in task execution times than a centralized greedy scheduler (in GNU OpenMP).**

To support scalable soft real-time computing, we develop *Real-Time Work-Stealing platform (RTWS)*, a real-time extension to the widely used Cilk Plus concurrency platform. RTWS employs federated scheduling to allocate cores to multiple parallel real-time tasks offline, while leveraging the work stealing scheduler to schedule each task on its dedicated cores online. RTWS supports parallel programs written in Cilk Plus and requires only task parameters that can be readily measured using existing Cilk Plus tools. Experimental results show that RTWS outperforms Real-Time OpenMP in term of deadline miss ratio, relative response time and resource efficiency on a 32-core system.

I. Introduction

Parallel real-time scheduling has emerged as a promising scheduling paradigm for computationally intensive real-time applications on multicore systems. Unlike in traditional *multiprocessor scheduling* with only *inter-task parallelism* (where each sequential task can only utilize one core at a time), in *parallel scheduling* each task has *intra-task parallelism* and can run on multiple cores at the same time. As today's real-time applications are trying to provide increasingly complex functionalities and hence have higher computational demands, they ask for larger scale systems in order to provide the same real-time performance. For example, real-time hybrid structural testing for a large building may require executing complex structural models over many cores at the same frequency as the physical structure [1].

Despite recent results in parallel real-time scheduling, however, we still face significant challenges in deploying large-scale real-time applications on microprocessors with increasing numbers of cores. In order to guarantee desired parallel execution of a task to meet its deadline, theoretic analysis often assumes that it is executed by a greedy (work conserving) scheduler, which requires a centralized data structure for scheduling. On the other hand, for general-purpose parallel job scheduling it has been known that centralized scheduling approaches suffer considerable scheduling overhead and performance bottleneck as the number of cores increases. In contrast, a *randomized work stealing* approach is widely used in many parallel runtime systems, such as Cilk, Cilk Plus, TBB, X10, and TPL [2]–[6]. In work stealing, each core steals work from a randomly chosen core in a decentralized manner, thereby avoiding the overhead and bottleneck of centralized scheduling. However, unlike a centralized scheduler, due to the randomized and distributed scheduling decision making strategy, work stealing may not be suitable for hard real-time tasks.

In this paper, we explore using randomized work stealing to support large-scale soft real-time applications that have timing constraints but do not require hard guarantees. Despite the unpredictable nature of work stealing, our experiments with benchmark programs found that work stealing (in Cilk Plus) delivers smaller maximum response times than a centralized greedy scheduler (in GNU OpenMP) while exhibiting small variance. To leverage randomized work stealing for scalable real-time computing, we present *Real-Time Work Stealing (RTWS)*, a real-time extension to the widely used Cilk Plus concurrency platform. RTWS employs federated scheduling to decide static core assignment to parallel real-time tasks offline, while using the work stealing scheduler to execute each task on its dedicated cores online. RTWS supports parallel programs written in Cilk Plus with only minimal modifications, namely a single level of indirection of the program's entry point. Furthermore, RTWS requires only task parameters that can be readily measured using existing Cilk Plus tools.

This paper presents the following contributions:

1) Empirical study of the performance and variability of parallel tasks under randomized work stealing vs. centralized greedy scheduler.
2) Design and implementation of RTWS, which schedules multiple parallel real-time tasks through the integration of federating scheduling and work stealing.
3) Theoretical analysis to adapt federated scheduling to incorporate work stealing overhead.

DOI 10.1109/RTSS.2016.27

4) Evaluation of RTWS with benchmark applications on a 32-core testbed that demonstrates the significant advantages of RTWS in terms of deadline miss ratio, relative response time and required resource capacity when comparing with the integration of federated scheduling and centralized scheduler.

II. Task Model

We first describe the types of parallel tasks considered in this paper. Specifically, we are interested in parallel programs that can be generated using parallel languages and libraries, such as Cilk [2], Intel Cilk Plus [3], OpenMP [7], Microsoft's Task Parallel Library [6], IBM X10 [5], etc. In these languages, the programmer expresses algorithmic parallelism, through linguistic constructs such as "spawn" and "sync," "fork" and "join," or parallel-for loops.

These programs can be modeled using **directed acyclic graphs (DAGs)**. In a DAG task, computational work is represented by nodes (vertices), while dependencies between sub-computations are represented by edges. Each task is characterized using two parameters: work and critical-path length. The **work** C_i of a DAG task τ_i is defined as the total execution time of all the nodes in a DAG, which is the task execution time on 1 core. The **critical-path length** L_i is defined as the sum of the execution times of the nodes that are in the longest path of the DAG, which is also the task execution time on an infinite number of cores. Figure 1 shows an example of DAG task with the critical-path annotated following the dashed line.

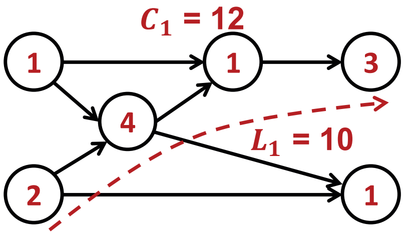


Figure 1: A directed acyclic graph (DAG) task τ_1 with six nodes. The execution time of each node is annotated in the center of the node. The total work C_1 is the sum of the execution times of all nodes, which is 12. The critical-path, i.e., the longest path in the DAG, is annotated using the dashed line. Hence, the critical-path length L_1 is 10.

In general, parallel programs can have arbitrary DAG structures. In real-time systems, researchers have given special consideration to a subset of DAG tasks, where the programs *only* use the parallel-for construct and do not nest these parallel-for loops. This restriction generates a special type of DAG, which we call **synchronous DAG**. Each parallel for-loop is represented by a **segment** — a segment contains a set of nodes (iterations) that can be executed in parallel with each other. The end of each segment is a synchronization point and the next segment can begin only after all iterations of the current segment complete. A sequential region of code is simply a segment with 1 iteration. Each synchronous task is a sequence of such segments. Synchronous tasks are also called as **Fork/Join** tasks in some publications. Figure 2 shows an example of a synchronous task with five segments; two of them are parallel segments, and the remaining three are sequential segments. This synchronous structure can be generated from a simple program shown in Figure 3, where parallel_for constructs can be Cilk Plus' cilk_for constructs or OpenMP's omp for directives.

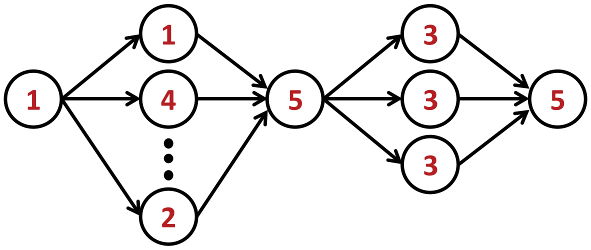


Figure 2: A synchronous task with two parallel-for loops. The execution time of each node is annotated in the center of the node. The second segment contains 20 nodes.

```
main()
{
    // Do some sequential work
    foo();
    // Do the first parallel segment
    parallel_for (i = 1; i <= 20; i++) {
        first_func();
    }
    // Other sequential work
    bar();
    // Do the second parallel segment
    parallel_for (i = 1; i <= 3; i++) {
        second_func();
    }
    // The last sequential work
    baz();
}
```

Figure 3: Example of a synchronous program.

Formally, a task set τ consists of n parallel tasks $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$, where each job of a task τ_i is a DAG program or a synchronous program — in principle, each job may have different internal structure. Each task τ_i has a period P_i and deadline D_i. We consider sporadic task sets with implicit deadlines, i.e. $P_i = D_i$. The *utilization* of a task τ_i is calculated as $u_i = C_i/P_i$, and the total utilization of the entire task set is $\sum_i u_i$. We want to schedule the task set τ on an m-core machine.

In this paper, we consider soft real-time tasks where a task is allowed to miss a few deadlines occasionally. Using the same resource capacity, a scheduling algorithm $\mathcal{S}$ has better performance if it schedules the same task set with a smaller **deadline miss ratio**, which is defined as the number of missed deadlines over the number of released jobs of the task set during a time interval.

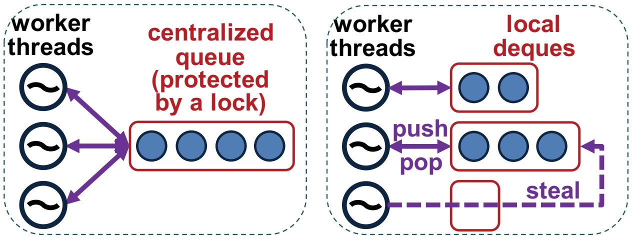
III. Scheduling Parallel Tasks

Most parallel languages and libraries, including those mentioned above, provide a runtime system that is responsible for scheduling the DAG on the available cores, i.e., dynamically dispatch the nodes of the DAG to these cores as the nodes become ready to execute. At a high-level, two types of scheduling strategies are often used: centralized scheduling and randomized work-stealing.

A. Centralized Schedulers

The system maintains a centralized data structure (such as a queue) of ready nodes that is shared by all the cores in a work sharing manner. There are a couple of possible instantiations of this strategy. In *push* schedulers, there is a master thread that dispatches work to other threads as they need this work. In *pull* schedulers, worker threads access this data structure themselves to grab work (ready nodes) as they need them. For example, the scheduler in the runtime system of GNU OpenMP is a pull scheduler, as in Figure 4(a).

Work-sharing schedulers have the nice property that they are **greedy** or **work-conserving** — as long as there are available ready nodes, no worker idles. However, these schedulers often have high overheads due to constant synchronizations. In particular, in a push scheduler, the master thread can only send work to cores one at a time. In a pull scheduler, the centralized queue must be protected by a lock and often incurs high overheads due to this.



(a) A pull scheduler (b) A work-stealing scheduler

Figure 4: Examples of centralized scheduling and work stealing

B. Randomized Work-Stealing Schedulers

In a randomized work-stealing scheduler, there is no centralized queue and the work dispatching is done in a distributed manner [2]. If a job is assigned n_i cores, the runtime system creates n_i worker threads for it. Each worker thread maintains a local double-ended queue (a *deque*), as shown in Figure 4(b). When a worker generates new work (enables a ready node from the job's DAG), it pushes the node onto the bottom of its deque. When a worker finishes its current node, it pops a ready node from the bottom of its deque. If the local deque is empty, the worker thread becomes a *thief* and randomly picks a *victim* thread among the other workers working on the same task and tries to steal work from the top of the victim's deque. For example, the third worker thread's deque is empty in Figure 4(b), so it randomly picks the second worker thread and steals work.

Randomized work-stealing is very efficient in practice and the amount of scheduling and synchronization overhead is small. In contrast to centralized schedulers where the threads synchronize frequently, very little synchronization is needed in work-stealing schedulers since (1) workers work off their own deques most of the time and don't need to communicate with each other at all and (2) even when a worker runs out of work and steals occur, the thief and the victim generally look at the opposite ends of the deque and don't conflict unless the deque has only 1 node on it.

However, because of this randomized and distributed characteristic, work-stealing is not strictly greedy (work conserving). In principle, workers can spend a large amount of time stealing, even if some other worker has a lot of ready nodes available on its deque. On the other hand, work-stealing provides strong probabilistic guarantee of linear speedup ("near-greediness") [8]. Moreover, it is much more efficient than centralized schedulers in practice. Therefore, variants of work stealing are the default strategies in many parallel runtime systems such as Cilk, Cilk Plus, TBB, X10, and TPL [2]–[6]. Thus, for soft real-time systems where occasional deadline misses are allowed, work stealing can be more resource efficient than a strictly greedy scheduler.

C. Specific Implementations of Centralized and Work-Stealing Schedulers

In this paper, we compare specific implementations of centralized and work-stealing schedulers: GNU OpenMP's centralized scheduler and GNU Cilk Plus's work-stealing scheduler. We choose these two implementations because OpenMP and CilkPlus are two of the most widely used parallel languages (and runtime systems) that have been developed by industry and the open source community over more than a decade; they are the only two parallel languages that are supported by both GCC and ICC.

OpenMP is a programming interface standard [7] for C, C++, and FORTRAN that allows a programmer to specify where parallelism can be exploited, and the GNU OpenMP runtime library in GCC is one of implementations of the OpenMP standard. OpenMP allows programmers to express parallelism using compiler directives. In particular, parallel for loops are expressed by `#pragma omp parallel for`, a parallel node in a DAG is expressed by `#pragma omp task` and synchronization between omp tasks is expressed by `#pragma omp taskwait`. While the details of scheduling are somewhat complex, and vary between omp parallel for loops and omp tasks, at a high level, GNU OpenMP provides an instantiation of a centralized pull scheduler. Available parallel work of a program is kept in a centralized queue protected by a global lock. Whenever a worker thread generates nodes of omp tasks or iterations in a parallel for loop, it has to get the global lock and places these nodes in the queue. When it finishes its current work, it again has to grab the lock to get more work from the queue.

Cilk Plus is a language extension to C++ for parallel programs and its runtime system schedules parallel programs using randomized work stealing. All Cilk Plus features are supported by GCC. Potential parallelism can be expressed using three keywords in the Cilk Plus language: a parallel node in a DAG is generated by cilk_spawn and the synchronization point is realized by cilk_sync; additionally, parallel for-loops are supported using a cilk_for programming construct. Note that in the underlying Cilk Plus runtime system, cilk_for is expanded into cilk_spawn and cilk_sync in a divide and conquer manner. Therefore, there is no fundamental difference between executing parallel DAGs or synchronous tasks in Cilk Plus. The Cilk Plus runtime system implements a version of randomized work stealing. When a function spawns another function, the child function is executed and the parent is placed on the bottom of the worker's deque. A worker always works off the bottom of its own deque. When its deque becomes empty, it picks a random victim and steals from the top of that victim's deque.

IV. The Case for Randomized Work Stealing for Soft Real-Time Tasks

In this section, we compare the performance of a work stealing scheduler in GNU Cilk Plus with a centralized scheduler in GNU OpenMP for highly scalable parallel programs. Our goal is to answer two questions: (1) Is it indeed the case that work stealing provides substantially better performance than centralized scheduler for parallel programs? Our experiments indicate that for many programs, including both synthetic tasks and real benchmark programs, work stealing provides much higher scalability. (2) Can work stealing be used for real-time systems? In particular, one might suspect that even if work stealing performs better than centralized scheduler *on average*, the randomization used in work stealing would make its performance too unpredictable to use even in soft real-time systems. Our experiments indicate that this is not the case — in fact, the variation in execution time using Cilk Plus' work-stealing scheduler is small and is comparable to or better than the variation seen in the deterministic centralized scheduler.

A. Scalability Comparison

We first compare the scalability of the OpenMP centralized scheduler with the Cilk Plus work-stealing scheduler. To do so, we use two types of programs: (1) three synthetic programs that are synchronous tasks; and (2) three real benchmark programs, namely Cholesky factorization, LU decomposition and Heat diffusion — none is synchronous and all have complex DAG dependences (of different types).

We implemented these programs in both Cilk Plus and OpenMP. It is important to note that the entire source code of each program is the same, except that the parallel directives are in either Cilk Plus or OpenMP. Both implementations are compiled by GCC, while linked to either Cilk Plus or OpenMP runtime libraries. Hence, the same program written in Cilk Plus and OpenMP has the same structure and therefore the same theoretical work and span.

No. cores	Synchronous Task - Type 1		
	OpenMP	Cilk Plus	Ratio
	(med., max, 99^{th} per.)	(med., max, 99^{th} per.)	
1	955.13, 958.10, 956.98	948.52, 953.41, 950.94	1.00
6	173.68, 174.31, 174.18	160.77, 161.46, 161.26	0.93
12	256.63, 259.19, 258.89	81.68, 82.56, 81.93	0.32
18	342.20, 365.99, 362.99	55.42, 59.22, 58.96	0.16
24	328.52, 331.11, 329.75	41.23, 45.22, 44.78	0.14
30	311.92, 330.00, 329.00	33.66, 35.02, 34.64	0.11
No. cores	Synchronous Task - Type 2		
	OpenMP	Cilk Plus	Ratio
	(med., max, 99^{th} per.)	(med., max, 99^{th} per.)	
1	1243.7, 1247.2, 1246.6	1237.2, 1239.9, 1239.2	0.99
6	210.22, 210.84, 210.74	213.77, 214.30, 214.19	1.02
12	111.58, 111.94, 111.87	107.90, 108.69, 108.11	0.97
18	95.55, 95.96, 95.92	73.45, 73.82, 73.62	0.77
24	85.97, 126.00, 123.01	58.95, 74.80, 69.18	0.59
30	86.74, 119.01, 86.96	45.07, 48.27, 47.33	0.41
No. cores	Synchronous Task - Type 3		
	OpenMP	Cilk Plus	Ratio
	(med., max, 99^{th} per.)	(med., max, 99^{th} per.)	
1	948.42, 950.39, 949.97	902.38, 903.29, 903.18	0.95
6	156.47, 156.94, 156.77	155.77, 156.06, 156.00	0.99
12	79.03, 79.34, 79.27	78.80, 79.46, 78.97	1.00
18	53.07, 53.49, 53.28	54.05, 54.41, 54.29	1.02
24	39.99, 69.95, 40.18	40.68, 44.35, 43.62	0.63
30	32.20, 33.18, 32.39	33.40, 37.12, 34.48	1.12

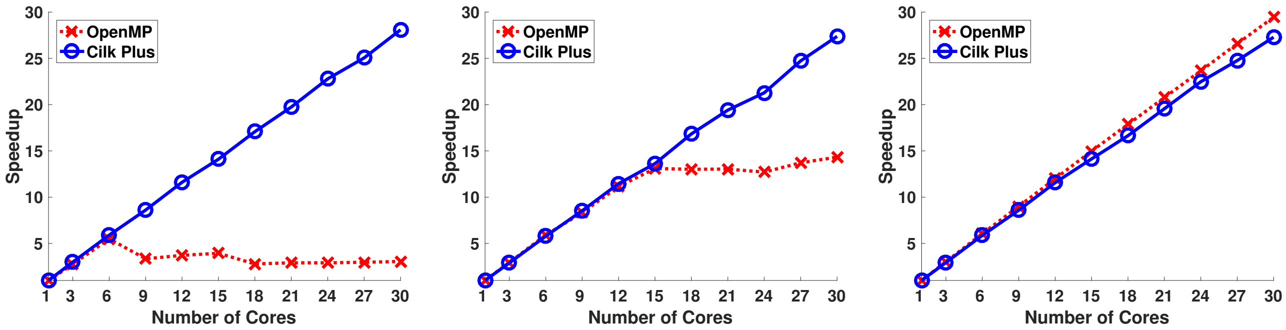
Table I: Median, maximum, and 99^{th} percentile execution times of synchronous tasks for OpenMP and Cilk Plus implementations (in milliseconds) and the ratios of the maximum execution times of Cilk Plus over OpenMP implementations.

Synthetic Synchronous Tasks: The synthetic synchronous tasks have different characteristics to compare the schedulers under different circumstances:

1) **Type 1** tasks have a large number of nodes per segment, but nodes has small execution times.
2) **Type 2** tasks have a moderate number of nodes per segment and moderate work per node.
3) **Type 3** tasks have a small number of nodes per segment, but nodes have large execution times.

The number of segments for all three types of synchronous tasks are generated from 10 to 20. For synchronous task type 1, we generate the number of nodes for each segment from $100,000$ to $200,000$ and the execution time per node from 5 to 10 nanoseconds; for task type 2, the number of nodes per segment varies from $10,000$ to $20,000$ and the execution time of each node from $2,000$ to $4,500$ nanoseconds; for task type 3, the number of nodes for each segment is from $1,000$ to $2,000$ and each node rans from $20,000$ to $50,000$ nanoseconds. The total work for synchronous tasks of different types was therefore similar. For each synchronous task generated, we ran it on varying numbers of cores with both Cilk Plus and OpenMP and we ran it 1000 times for each setting.

Table I shows the median, maximum, and 99^{th} percentile execution times of OpenMP and Cilk Plus tasks as well as



(a) Speedup of synchronous task type 1 (b) Speedup of synchronous task type 2 (c) Speedup of synchronous task type 3

Figure 5: Speedup of synchronous tasks in OpenMP and Cilk Plus implementations

the ratios of the maximum execution time of Cilk Plus over OpenMP implementations for the three types of synchronous tasks on varying numbers of cores. For most settings, Cilk Plus tasks obtain smaller maximum execution times than OpenMP tasks, as shown in the ratios. We also notice that for type 1 tasks the execution times of the OpenMP tasks even increase when the number of cores is high (e.g, for 18, 24, 30 cores) whereas Cilk Plus tasks keep a steady speedup.

Figure 5 shows the speedup of these synchronous tasks. For all three types of tasks, Cilk Plus provides steady and almost linear speedup as we scale up the number of cores. In contrast, for synchronous task type 1 in Figure 5(a) where the segment lengths are short and there are many nodes in each segment, OpenMP inevitably suffers high synchronization overhead due to the contention among threads that constantly access the global work queue. This overhead is mitigated when the number of nodes in each segment is smaller and the segment lengths are longer, as in Fig. 5(c). In this setting, OpenMP slightly outperforms Cilk Plus, though Cilk Plus still has comparable speedup to OpenMP. Figure 5(b) demonstrates the scalability of OpenMP and Cilk Plus with parameters generated in between.

Real DAG Benchmark Programs: To compare the performance between work stealing and centralized scheduler for programs with more complex DAG structures, we use three benchmark programs as described below.

(a) Cholesky factorization (Cholesky): Using divide and conquer, Cholesky program performs Cholesky factorization of a sparse symmetric positive definite matrix into the product of a lower triangular matrix and its transpose. The work and parallelism of Cholesky both increase when the matrix size increases. Note that because Cholesky is parallelized using divide and conquer method, it has lots of spawn and sync operations, forming a complex DAG structure.

(b) LU decomposition (LU): Similar to Cholesky, LU also performs matrix factorization, but the input matrix does not need to be positive definite and the output upper triangular matrix is not necessarily the transpose of the lower triangular matrix. LU also decomposes the matrix using divide and conquer and provides abundant parallelism.

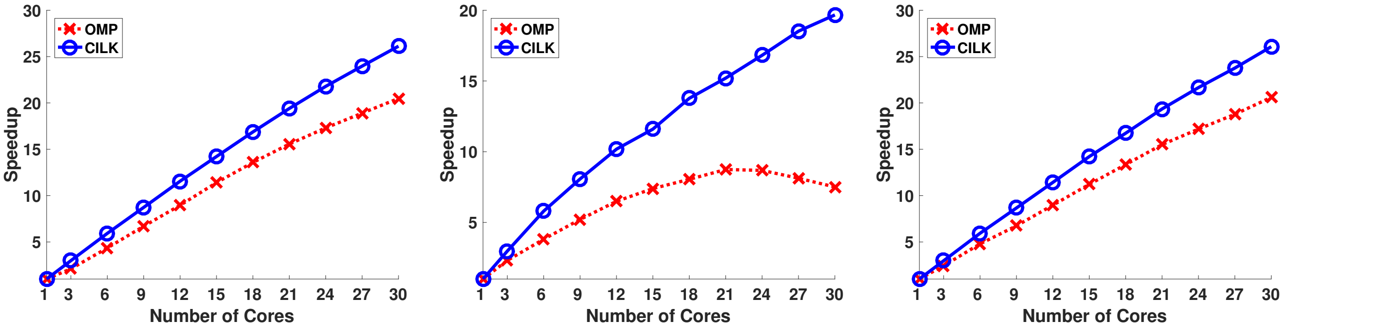
(c) Heat diffusion (Heat): This program uses the Jacobi iterative method to solve an approximation of a partial differential equation that models the heat diffusion problem. The input includes a 2-dimension grid with the numbers of rows and columns, and the number of time steps (or iterations) the computation is performed on that 2D grid. Within each time step, the computation is carried out in a divide and conquer manner.

No. cores	Cholesky Factorization		
	OpenMP (med., max, 99^{th} per.)	Cilk Plus (med., max, 99^{th} per.)	Ratio
1	32.12, 32.18, 32.17	32.31, 32.36, 32.35	1.01
6	7.39, 7.62, 7.61	5.44, 5.47, 5.47	0.72
12	3.58, 3.72, 3.71	2.79, 2.89, 2.87	0.78
18	2.36, 2.43, 2.43	1.91, 1.96, 1.95	0.81
24	1.85, 1.92, 1.92	1.48, 1.52, 1.51	0.79
30	1.56, 1.62, 1.61	1.23, 1.28, 1.28	0.79
No. cores	LU Decomposition		
	OpenMP (med., max, 99^{th} per.)	Cilk Plus (med., max, 99^{th} per.)	Ratio
1	16.98, 17.09, 17.07	16.76, 16.82, 16.82	0.98
6	3.53, 3.79, 3.79	2.82, 2.84, 2.83	0.75
12	1.89, 1.97, 1.97	1.44, 1.87, 1.78	0.95
18	1.27, 1.37, 1.35	0.99, 1.07, 1.06	0.78
24	0.99, 1.06, 1.05	0.76, 0.84, 0.83	0.79
30	0.82, 0.86, 0.86	0.64, 0.71, 0.69	0.82
No. cores	Heat Diffusion		
	OpenMP (med., max, 99^{th} per.)	Cilk Plus (med., max, 99^{th} per.)	Ratio
1	51.57, 52.04, 52.04	51.70, 52.11, 52.11	1.00
6	13.50, 13.83, 13.81	8.80, 9.28, 9.26	0.67
12	7.93, 8.41, 8.31	5.06, 5.82, 5.70	0.69
18	6.40, 6.73, 6.69	3.73, 3.96, 3.95	0.59
24	5.94, 6.10, 6.10	3.06, 4.06, 3.67	0.67
30	6.87, 7.20, 7.17	2.62, 2.73, 2.73	0.38

Table II: Median, maximum, and 99^{th} percentile execution times of Cholesky, LU, and Heat for OpenMP and Cilk Plus implementations (in seconds) and the ratio of the maximum execution times of Cilk Plus over OpenMP implementations.

The Cholesky program was run for a matrix of size 3000×3000. The LU program was run for a matrix of size 2048×2048. For both of them, the base case matrix had size of 32×32. The Heat program was run with a 2-dimensional input of size 4096×1024 and 800 time steps. For each setting, we ran the program 100 times.

For each program, we first compare its execution times under work stealing and centralized scheduler on varying numbers of cores, as shown in Table II. For all three



(a) Cholesky with matrix size 3000×3000 (b) Heat with input size 4096×1024 (c) LU with matrix size 2048×2048

Figure 6: Speedup of benchmark programs in OpenMP and Cilk Plus implementations

benchmarks, we notice that the execution times are tight which means both scheduling strategies have a decent predictability. However, Cilk Plus implementations have smaller maximum execution times which means that Cilk Plus tasks have higher chance of finishing by their deadlines.

Figure 6 shows the speedups of these programs in the same experiments. For matrix computation programs like Cholesky and LU, where there is abundant parallelism, OpenMP obtains good speedups but Cilk Plus obtaines even better speedups. The difference is more notable in the Heat diffusion program, where there is less parallelism to exploit. For this program, Cilk Plus still has reasonable speedup, while the speedup of OpenMP starts to degrade when the number of cores is more than 21.

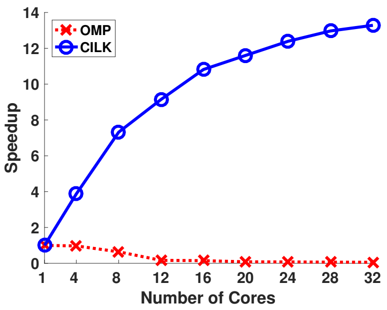


Figure 7: Cholesky with input size 1000×1000 **base case** 4×4

We also notice that for Cholesky and LU programs, the performances of OpenMP are quite sensitive to the base case sizes whereas Cilk Plus performed equally well regardless of the base case sizes. For demonstration, Figure 7 shows the experiment results of Cholesky with a base case matrix of size 4×4. Notably, no speedup was observed for OpenMP when the number of cores increases. Thus, one has to tune the base case size for OpenMP in order to get comparable performance with their Cilk Plus counterparts. This is again caused by the fact that the overhead of centralized scheduler adds up and outweighs the performance gain by running program in parallel, when the base case is small.

B. Tightness of Randomized Work Stealing in Practice

One might expect that even though a work-stealing scheduler may perform well on average due to low overheads, would not be suitable for real-time platforms due to high variability in its execution times due to randomness. However, this intuition turns out to be inaccurate. Theoretically, strong high probability bounds have been proven for the execution times for work stealing [8], [9]. Our experiments also suggest that the variation in execution time is small in practice. In our experiments, the difference between the mean execution time and the 99^{th} percentile execution time is less than 5% most of the times and the variation between the mean and the maximum execution time is also small.

More importantly, the variation shown by work stealing is never worse than (and is generally better than) that shown by the deterministic scheduler used by OpenMP. This indicates that work-stealing schedulers show promise for use in real-time systems, especially soft real-time systems which can tolerate some deadline misses, since they can potentially provide much better resource utilization than centralized schedulers for parallel tasks.

Discussion: One might wander whether a different centralized scheduler that builds on better synchronization primitives can outperform Cilk Plus's work-stealing scheduler. Our experiments in Figure 5(a), 5(b) and 7 indicate that the higher overhead of centralized scheduler mostly comes from the larger number of synchronization operations on the centralized global queue compared to lower contention on the distributed local queues. Therefore, even if synchronization primitives of the centralized scheduler is further optimized to reduce overheads, it is still unlikely to negate the inherent scalability advantages of randomized work-stealing, especially with increasing number of cores and workload complexity.

V. Adaptation to Federated Scheduling using Work Stealing

As demonstrated in Section IV, a centralized greedy scheduler incurs high overheads for parallel tasks and is less scalable compared to a randomized work-stealing scheduler. Therefore, when the task set allows occational deadline misses, using randomized work stealing can be more resource efficient and scalable. In order to leverage work stealing, while providing soft real-time performance to parallel task sets, we adapt federated scheduling to incorporate work stealing overhead. In this section, we first briefly introduce federated scheduling and then present how we can adapt it to incorporate work stealing overhead.

A. Federated Scheduling for Parallel Real-Time Tasks

The **federated scheduling** [10] is an existing scheduling paradigm for parallel real-time tasks. Given a task set τ, federated scheduling either *admits* a task set and outputs a **core assignment** for each task; or declares the task set to be unschedulable. In the core assignment, each high-utilization task (utilization > 1) is allocated n_i dedicated cores, where $n_i = \left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil$. During runtime, a *greedy* scheduler is required to execute each high-utilization task on its dedicated cores. All the low-utilization tasks are forced to execute sequentially on the remaining cores scheduled by a multiprocessor scheduling algorithm. Since low-utilization tasks do not need parallelism to meet deadlines, in this work we focus only on high-utilization tasks.

Federated scheduling has been proved to have a **capacity augmentation bound** of 2, meaning that given m cores federated scheduling can guarantee schedulability to any task set τ as long as it satisfies: (1) the total utilization of the task set is no more than half of the total available cores – $\sum u_i \leq m/2$; (2) for each task, the critical-path length is less than half of its deadline – $L_i < D_i/2$.

Discussion: Why do we choose to integrate work stealing into federated scheduling instead of other real-time schedulers, such as global EDF? Firstly, as discussed in related work, federated scheduling has the best capacity augmentation bound of 2, so it can schedule task sets with higher load than other scheduler strategies. More importantly, it has the benefit that the parallel scheduler used for executing parallel task does not need to be deadline- or priority-aware, since the task is assigned with dedicated cores. Otherwise, worker threads of the parallel scheduler have to be able to quickly switch to jobs with shorter deadlines or higher priorities when they arrive. However, recall that the advantage of work stealing is that a worker thread works off its own deque most of the time, which is against the requirement of fast switching between jobs. Because of this, implementing other parallel real-time scheduling strategies using work stealing can be difficult and involve high overheads.

B. Incorporating work stealing overhead into federated scheduling

When each parallel task is executed by a work-stealing scheduler on its dedicated cores, the core assignment of federated scheduling needs to incorporate work stealing overheads when calculating core assignment. In work stealing, there are two types of overheads that we need to consider: stealing overhead and randomization overhead.

Stealing overheads includes the explicit costs of bookkeeping parallel nodes and the implicit cost of cache misses due to the migration of stolen nodes. Since stealing can only occur at spawn and sync points in the DAG, given a specific DAG one can estimate the overheads due to scheduling events by counting these quantities. Based on this insight, *burdened DAG* is introduced to estimate stealing overheads for DAG task [11]. Using profiling tools, such as Cilk View and Cilkprof [12], the *burdened critical-path length* $\hat{L_i}$, with stealing overheads incorporated, can be measured. When calculating core assignment for tasks, we use the burdened critical-path length $\hat{L_i}$ to replace critical-path length L_i, so that stealing overhead is included.

Compared to a greedy scheduler, the randomness of work stealing introduces additional overhead. In particular, even if there is available work on another core, a core may still take some time to find the available work, because the random stealing may fail to find the available work. Thus, the execution time of a task under work stealing is a random variable. By extending the result of stochastic federated scheduling in [13], we incorporate the randomization overhead into core assignment and also analyze the expected tardiness bound for federated scheduling using work stealing.

To analyze the randomness overhead, we first state known results on work-stealing response time γ_i for task τ_i with total execution time C_i and critical path-length L_i [8].

Lemma 1: **[Tchi.13]** A work-stealing scheduler guarantees completion time γ_i on n_i dedicated cores where

$$\mathrm{E}\left[\gamma_i\right] \leq \frac{C_i}{n_i} + \delta L_i + 1 \tag{1}$$

$$\mathbb{P}\left\{\gamma_i \leq \frac{C_i}{n_i} + \delta(L_i + \log_2 \frac{1}{\epsilon}) + 1\right\} \geq 1 - \epsilon \tag{2}$$

Note that the δ in the above formula is the critical-path length coefficient. Theoretically it has been proven to be at most 3.65, while empirically it is set to 1.7 for measurement using Cilk View [11] and is set to 1.5 when using Cilkprof.

Consider a random variable X with CDF function

$$\mathbb{F}(x) = \mathbb{P}\{X \leq x\} = 1 - e^{-\lambda(x-\mu)}$$

where $\mu = \frac{C_i}{n_i} + \delta L_i + 1$ and $\lambda = \frac{\ln 2}{\delta}$. This is the CDF of a shifted exponential distribution with mean value $\mathrm{E}\left[X\right] = \mu + \frac{1}{\lambda} = \frac{C_i}{n_i} + \delta(L_i + \frac{1}{\ln 2}) + 1$ and variance λ^{-2}.

If we set $x = \mu + \delta \log_2 \frac{1}{\epsilon}$, then we get $\epsilon = e^{-\lambda(x-\mu)}$. Using Inequality (2), the above CDF can be rewritten as

$$\begin{aligned}\mathbb{F}(x) =& \mathbb{P}\left\{X \leq \frac{C_i}{n_i} + \delta(L_i + \log_2 \frac{1}{\epsilon}) + 1\right\} = 1 - \epsilon \\ \geq& \mathbb{P}\left\{\gamma_i \leq \frac{C_i}{n_i} + \delta(L_i + \log_2 \frac{1}{\epsilon}) + 1\right\}\end{aligned}$$

Therefore, the CDF of random variable X is the upper bound of the CDF of completion time γ_i. Every instance j drawn from the distribution of γ_i can be mapped to an instance in the distribution of X that is no smaller than j. In other words, X's probability density function of $f(x) = \lambda e^{-\lambda(x-\mu)}$ is the worst-case distribution of completion time γ_i of task τ_i under work stealing.

Now we can use a lemma from queueing theory [14] to calculate core assignment and bound the response time for federated scheduling incorporated with work stealing.

Lemma 2: **[KING70]** For a D/G/1 queue, customers arrive with minimum inter-arrival time Y, and the service time $\mathcal{X}$ is a distribution with mean $\mathrm{E}[\mathcal{X}]$ and variance $\delta^2_{\mathcal{X}}$. If $\mathrm{E}[\mathcal{X}] < Y$, then the queue is stable and the expected response time $\mathcal{R}$ is bounded by $\mathrm{E}[\mathcal{R}] \leq \mathrm{E}[\mathcal{X}] + \frac{\delta^2_{\mathcal{X}}}{2(Y - \mathrm{E}[\mathcal{X}])}$.

Inspired by the stochastic analyses in [15] and [13], Lemma 2 can be interpreted as follows: parallel jobs are customers; implicit deadline is the inter-arrival time $Y = D_i$; and the completion time on n_i dedicated cores using work stealing is the service time $\mathcal{X} = \gamma_i$. As discussed above, $f(x)$ is the worst-case distribution of γ_i with mean value $\frac{C_i}{n_i} + \delta(L_i + \frac{1}{\ln 2}) + 1$. Thus, Lemma 2 guarantees bounded response time for $n_i > \frac{C_i}{D_i - \delta(L_i + \frac{1}{\ln 2}) - 1}$, since

$$\mathrm{E}[\mathcal{X}] = \mathrm{E}[\gamma_i] \leq \frac{C_i}{n_i} + \delta(L_i + \frac{1}{\ln 2}) + 1 < D_i = Y$$

Therefore, after incorporating the stealing overhead and randomness overhead into federated scheduling, the number of cores assigned to a task is adapted as

$$n_i = \left\lfloor \frac{C_i + D_i - \delta\hat{L_i}}{D_i - \delta\hat{L_i}} \right\rfloor \tag{3}$$

Note that we omitted the terms $\frac{1}{\ln 2}$ and 1, because they are in unit time step, which is negligible compared with C_i and L_i in actual time. If $D_i \leq \delta\hat{L_i}$, the task is deem unschedulable.

From Lemma 2, we can also calculate the bound on task expected response time R_i. Again as $f(x)$ is the worst-case distribution of γ_i with variance $(\frac{\delta}{\ln 2})^2$, given n_i cores the expected response time of task τ_i is bounded by

$$\begin{aligned}\mathrm{E}[R_i] &\leq \mathrm{E}[\gamma_i] + \frac{\delta^2_{\gamma_i}}{2(D_i - \mathrm{E}[\gamma_i])}\\ &\leq \frac{C_i}{n_i} + \delta\hat{L_i} + \frac{(\frac{\delta}{\ln 2})^2}{2(D_i - \frac{C_i}{n_i} - \delta\hat{L_i})}\end{aligned}$$

VI. RTWS Platform

In this section, we describe the design of the **RTWS** platform, which provides federated scheduling service for parallel real-time tasks. RTWS has several benefits: (1) It separates the goals of efficient parallel performance and rigorous real-time execution. This separation of concerns allows programmers to re-purpose existing parallel applications to be run with real-time semantics with minimal modifications. (2) It allows the use of existing parallel languages and runtime systems (not designed for real-time programs) to explore the degree of real-time performance one can achieve without implementing an entirely new parallel runtime system. Therefore, we were able to evaluate the performance of centralized scheduler from OpenMP and the work stealing scheduler from Cilk Plus for real-time task sets. (3) While RTWS does not explicitly consider cache overheads, the scheduling policy has an inherent advantage with respect to cache locality, since parallel tasks are allocated dedicated cores and never migrate.

Application Programming Interface (API): The RTWS API makes it easy to convert existing parallel programs into 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 **`run`** function is specified, which is periodically executed when a job of the task is invoked. In addition, a configuration file must be provided for the task set, specifying runtime parameters (including program name and arguments) and real-time parameters (including period, work and burdened critical-path length) for each task.

Platform Structure and Operation: RTWS separates the functionalities of parallel scheduling and real-time scheduling. We use two components to enforce these two functionalities, an **real-time scheduler (RT-scheduler)** and a **parallel dispatcher (PL-dispatcher)**.

Specifically, the RT-scheduler provides the real-time performance of a task. Prior to execution, it reads tasks' real-time parameters from the configuration file and calculates a core assignment using the formula (3) in Section V during offline, which has incorporated work stealing overheads into federated scheduling. The main function (provided by RTWS) binds each task to its assigned cores (by changing the CPU affinity mask). This core assignment ensures that each task has sufficient number of dedicated cores to meet most of its deadline during execution. Moreover, because each parallel task is executed on dedicated cores and no other tasks can introduce CPU interference with it, the PL-dispatcher does not need to be deadline- or priority-aware.

During execution, the PL-dispatcher enforces the periodic invocation of each task and calls an individual GNU Cilk Plus (or OpenMP) runtime system to provide parallel execution of each task. Since there are multiple concurrent parallel runtime systems that are unaware of each other, we need to entirely isolate them from each other to minimize scheduling overheads and CPU interference. Therefore, for Cilk Plus we modified its runtime system, so that each Cilk Plus runtime only creates n_i workers, each of which is pinned to one of the n_i assigned cores. Similarly, for OpenMP we use *static* thread management and create exactly n_i threads to each task. In other words, there is only one worker thread per core and hence the worker assignment by PL-dispatcher is consistent with the core assignment of the RT-scheduler.

Profiling Tool: Since the work and critical-path length of each task must be specified to the platform (in the configuration file), we also provide a simple profiling utility to automatically measure these quantities for each task. The work of a task can be measured by running the profiling program on a single core. Measuring the critical-path length is more difficult. We adopt a profiling tool Cilkprof [12], which can automatically measure the work and the burdened critical-path length of a single job. In particular, Cilkprof uses compiler instrumentation to gather the execution time of every *call site* (i.e., a node in the DAG) and calculate the critical-path length in nanosecond. To be consistent with GNU Cilk Plus (and GNU OpenMP), we use a version of Cilkprof that instrumented the GCC compiler and incorporated the burdened DAG into the measurement. Intel provides another tool Cilkview [11] that can measure the number of instructions of burdened critical-path length using dynamic binary instrumentation.

Discussion: In addition to using the work-stealing scheduler of Cilk Plus in RTWS (where the Real-Time Work-Stealing (RTWS) comes from), the design of RTWS allows us to instantiate another version of federated scheduling service that uses the centralized scheduler of GNU OpenMP (which we name as Real-Time Centralized Greedy (RTCG) platform). As shown in Section IV, work stealing has better parallel performance than the centralized scheduler. Thus, RTWS using work stealing is a better candidate for parallel tasks with soft real-time constraints, as confirmed via the empirical comparison in Section VII. However, it may not be the best approach for other scenarios. First and foremost, the execution time of a parallel task using work stealing can be as slow as its sequential execution time in the worst case, even though the probability of the worst case happening can be extremely low in practice. Therefore, it can never be applied to hard real-time systems without modifying the work stealing protocol to provide some form of progress guarantee. In addition, for special purposed system where the structure of parallel task is static and well measured, a static scheduler that decides how to execute the parallel task prior to execution can effectively reduce scheduling overheads and may perform better than work stealing.

VII. Platform Evaluation

In this section, we evaluate the soft real-time performance provided by RTWS using a randomized work-stealing scheduler (**RTWS**) compared to the alternative implementation of federated scheduling using a centralized greedy scheduler (**RTCG**). We use three DAG applications written in both Cilk Plus and OpenMP (discussed in Section IV) to randomly generate task sets for emperical experiments. To the best of our knowledge, RTWS is the first real-time platform that supports general DAG tasks, such as these benchmark programs. Since other existing real-time systems do not support parallel DAG tasks, we do not compare against them.

Experiments were conducted on a 32-core machine composed of four Intel Xeon processors (each with 8 cores). When running experiments, we reserved two cores for operating system services, leaving 30 experimental cores. Linux with CONFIG_PREEMPT_RT patch version r14 applied was the underlying RTOS.

A. Benchmark Task Sets Generation

We now describe how we generate task sets composed of the three benchmark programs (Cholesky, Heat and LU) with the general DAG structures. We generate 4 sets of task sets and evaluate their performances. The first 3 sets are composed with tasks running the same application, denoted as **Cholesky**, **Head** and **LU task sets**. The last set comprises a mix of all benchmarks, denoted as **Mixed task sets**.

We profile Cholesky, Heat and LU programs using 14, 6 and 3 different input sizes, respectively. For each program with each input size, we measure its work and burdened critical-path length using Cilkprof. Then we generate different tasks (from one benchmark with one input size) and assign it with a randomly generated utilization. To see the effect of scalability of large parallel tasks (i.e., spanning many cores), we intentionally create 5 types of tasks: tasks with mean utilization from $\{1, 3, 6, 12, 15\}$. When assigning utilization to a task, we always try to pick the largest mean utilization that does not make the task set utilization exceed the total utilization that we desire. After deciding a mean utilization, we will then randomly generate the utilization of the task using the mean value. A task's period is calculated using its work over utilization. We keep adding tasks into the task set, until it reaches the desired total utilization. For each setting, we randomly generate 10 task sets.

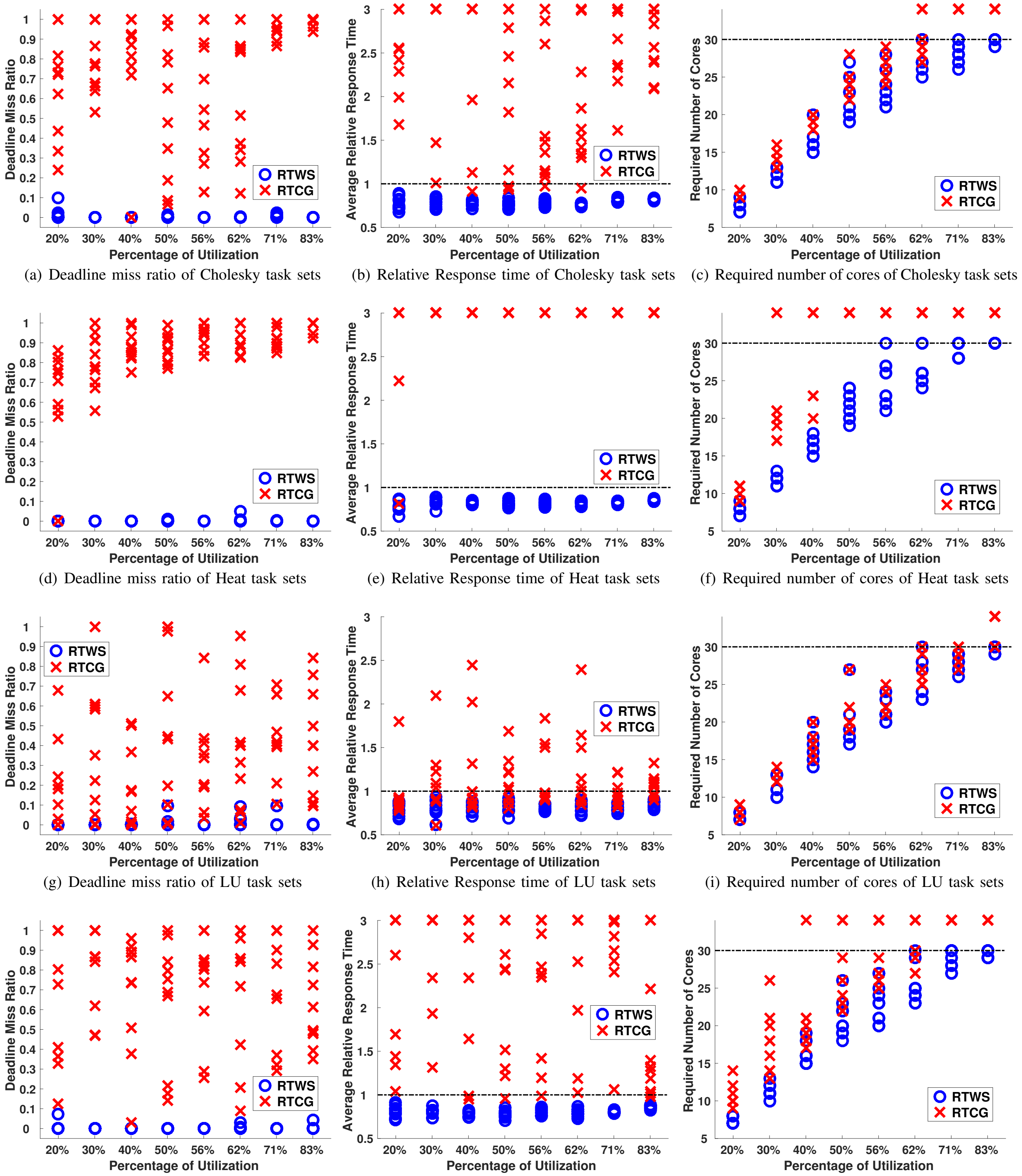
B. Evaluation Results

For each DAG task set, we record the **deadline miss ratio**, which is calculated using the total number of deadline misses divided by the total number of jobs in the task set. We also record the response time of each individual job during the execution to calculate a **relative response time**, which is the job's response time over its deadline. We then calculate the average relative response time for each task set.

In the first two comparisons between RTWS and RTCG, we'd like to see how the integration of federated scheduling and randomized work stealing performs compared with federated scheduling using a centralized greedy scheduler *given the same resource capacity* for soft real-time task sets. Therefore, for these experiments we use the *same core assignment* as described in Section V, which incorporates work stealing overheads into federated scheduling.

Since the centralized scheduler generally has larger overheads and takes longer to execute as shown in Section IV, it is not surprising to see that RTCG performs worse than RTWS given the same resource capacity. To further analyze the performance difference between the two approaches, in the last experiment we increase the resource capacity for RTCG. We'd like to see how much more resource capacity RTCG requires in order to *schedule the same task sets* compared with RTWS.

(1) Deadline miss ratio comparison: We first compare the deadline miss ratio in Figure 8(a),8(d),8(g) and 8(j) for Cholesky, Heat, LU and Mixed task sets, respectively. Notably, most of the task sets under RTWS has no deadline misses and all of the task sets have a deadline miss ratio no more than 10%. In fact, from all the experiments we run, there are only 2.25% tasks (28 out of 1243 tasks) having deadline misses. In contrast, given the same core assignment



(a) Deadline miss ratio of Cholesky task sets

(b) Relative Response time of Cholesky task sets

(c) Required number of cores of Cholesky task sets

(d) Deadline miss ratio of Heat task sets

(e) Relative Response time of Heat task sets

(f) Required number of cores of Heat task sets

(g) Deadline miss ratio of LU task sets

(h) Relative Response time of LU task sets

(i) Required number of cores of LU task sets

(j) Deadline miss ratio of Mixed task sets

(k) Relative Response time of Mixed task sets

(l) Required number of cores of Mixed task sets

Figure 8: Deadline miss ratio, average relative response time and required number of cores of different task sets (Cholesky, Heat, LU and Mixed task sets) with increasing total utilization under RTWS (providing federated scheduling service integrated with a randomized work-stealing scheduler in GNU Cilk Plus) and RTCG (providing federated scheduling service integrated with a centralized greedy scheduler in GNU OpenMP). For experiments that measure deadline miss ratios and average relative response time (i.e., the first two columns), RTWS and RTCG use the same core assignment. For experiments that examine the number of required cores (i.e., the last column), we increase the number of cores for each task under RTCG until it misses no more than 60% of deadlines.

RTCG misses substantially more deadlines, especially for Heat task sets where many tasks miss all of their deadlines.

(2) Relative response time comparison: In Figure 8(b),8(e),8(h) and 8(k), we observe that RTCG has much higher average relative response time than RTWS, given the same resource capacity. For all task sets, the average relative response time of RTWS is less than 1, while some tasks under RTCG even have relative response times larger than a hundred. In order to clearly see the relative response times smaller than 1, when plotting the figures we mark all the relative response times that are larger than 3 as 3.

(3) Required resource capacity: From the first two comparisons, we can clearly see that RTCG requires more cores (i.e., resource capacity) in order to provide the same real-time performance as RTWS. Thus, in Figure 8(c),8(f),8(i) and 8(l) we keep increasing the number of cores assigned to tasks under RTCG that have more than 25% of deadline misses. Note that all tasks under RTWS meet at least 80% of deadlines. We compare the required number of cores of RTCG and RTWS for the same task sets to meet most of their deadlines. If a task set misses most deadlines when allocated with all the 30 available cores, then we mark the number of required cores as 34. For Cholesky and LU task sets, RTCG requires about 1 to 3 additional cores. For some Heat task sets, even doubling the number of cores for RTCG is still not sufficient.

VIII. Related Work

Real-time multiprocessor scheduling for hard real-time systems has been well researched both for global and partitioned schedulers, and for fixed and dynamic priorities [16], [17]. Most prior work on soft real-time guarantees considers sequential tasks with worst-case parameters [18]. For these tasks, many different global schedulers [19]–[22] can provide bounded tardiness with no utilization loss.

For parallel tasks, almost all research focused on tasks with hard real-time requirements. Earlier work uses different task decomposition techniques to provide hard real-time bounds on different parallel task models [23]–[27]. Later, better bounds for schedulers without decomposition have been proved for general DAG tasks. In particular, global EDF and global deadline monotonic scheduling have been proved to have resource augmentation bounds of $2 - 1/m$ and $3 - 1/m$, respectively [28], [29]; they are also proved to have capacity augmentation bounds of 2.6 and 3.7 in [10]. In the same work, a federated scheduling paradigm is proposed, which provides the best capacity augmentation bound of 2.

For soft real-time scheduling of parallel tasks, Liu and Anderson [30] provide analysis for bounded response time of global EDF, and Nogueira et al. [31] investigate in priority-based work stealing scheduling scheme. The only work considering parallel tasks with soft real-time constraints that we are aware of analyzes bounded tardiness for stochastic parallel tasks for federated scheduling [13].

Regarding system implementation, we are aware of two systems [26], [32] that support parallel real-time tasks based on different decomposition-based analysis. Kim et al. [26] used a reservation-based OS to run a parallel real-time program of an autonomous vehicle. [32] implement decomposition-based fixed-priority scheduling, called RT-OpenMP, on standard Linux. However, both systems are based on task decomposition, which usually requires substantial modification to application programs, compilers, and/or the operating system. They also requires detailed information about task structure and subtask execution times.

For platforms without task decomposition, [33] presents a platform supporting partitioned fixed-priority scheduling for parallel real-time tasks on a special COMPOSITE operating system with significantly lower parallel overhead. Li [34] implements a prototype parallel global EDF platform, called PGEDF, which is based on LITMUS$^{\text{RT}}$ [35] patched Linux. Note that all of these systems only support scheduling synchronous (fork-join) tasks.

Recently, scalability of global scheduling was addressed with the use of message passing to communicate global scheduling decision [36]. Before that Brandenburg et al., [37] empirically studied the scalability of several scheduling algorithms for multiprocessors. However, none of them has targeted the scalability problem of parallel tasks.

A prototype version of our RTCG platform, which was tailored for the centralized greedy scheduler in OpenMP and programs with only parallel for-loops, has already been used in civil engineering domains. Ferry et al. [1] studied how to exploit parallelism in real-time hybrid structural simulations (RTHS) to improve real-time performance. The resulted parallelized RTHS program was executed and scheduled by our RTCG prototype. Experiments on RTHS in [1] thus illustrates how parallel real-time scheduling can practically help to improve performance in cyber-physical systems.

IX. Conclusions

In this paper, we present RTWS, the first parallel scheduling system for soft real-time tasks with DAG structures. RTWS adapts and integrates federated scheduling with work stealing. Integrated with the widely used Cilk Plus concurrency platform, RTWS can schedule standard Cilk Plus programs. Furthermore, RTWS does not require detailed knowledge of task structure. Instead, it only use coarse-grained task parameters that can be easily measured using existing Cilk Plus tools. Experimental results demonstrated that RTWS can considerably improve the response time of tasks on a given number of cores. Especially when running real benchmark programs, it siginificatly reduces the required resources for task set schedulability. Therefore it represents a promising step towards practical development and deployment of real-time applications based on existing programming languages, platform and tools.

Acknowledgment

This research was supported by Fullgraf Foundation, NSF grants CCF-1136073, CCF-1337218, CNS-1329861 and ONR grants N000141310800 and N000141612108.

References

[1] D. Ferry, G. Bunting, A. Maghareh, A. Prakash, S. Dyke, K. Agrawal, C. Gill, and C. Lu, "Real-time system support for hybrid structural simulation," in *EMSOFT*, 2014.

[2] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *Journal of parallel and distributed computing*, vol. 37, no. 1, pp. 55–69, 1996.

[3] Intel, "Intel CilkPlus v1.2," Sep 2013, https://www.cilkplus.org/sites/default/files/open_specifications/Intel_Cilk_plus_lang_spec_1.2.htm.

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

[5] O. Tardieu, H. Wang, and H. Lin, "A work-stealing scheduler for x10's task parallelism with suspension," in *PPoPP*, 2012.

[6] D. Leijen, W. Schulte, and S. Burckhardt, "The design of a task parallel library," in *Acm Sigplan Notices*, vol. 44, 2009, pp. 227–242.

[7] OpenMP, "OpenMP Application Program Interface v4.0," July 2013, http://http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[8] M. Tchiboukdjian, N. Gast, D. Trystram, J.-L. Roch, and J. Bernard, "A tighter analysis of work stealing," *Algorithms and Computation*, pp. 291–302, 2010.

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

[10] J. Li, J.-J. Chen, K. Agrawal, C.Lu, C. Gill, and A. Saifullah, "Analysis of federated and global scheduling for parallel real-time tasks," in *ECRTS*, 2014.

[11] Y. He, C. E. Leiserson, and W. M. Leiserson, "The cilkview scalability analyzer," in *Parallelism in algorithms and architectures*, 2010.

[12] T. B. Schardl, B. C. Kuszmaul, I. Lee, W. M. Leiserson, C. E. Leiserson *et al.*, "The cilkprof scalability profiler," in *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2015.

[13] J. Li, K. Agrawal, C. Gill, , and C. Lu, "Federated scheduling for stochastic parallel real-time tasks," in *RTCSA*, 2014.

[14] J. Kingman, "Inequalities in the theory of queues," *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 102–110, 1970.

[15] A. F. Mills and J. H. Anderson, "A stochastic framework for multiprocessor soft real-time scheduling," in *RTAS*, 2010.

[16] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Computing Surveys*, vol. 43, no. 4, p. 35, 2011.

[17] M. Bertogna and S. Baruah, "Tests for global edf schedulability analysis," *Journal of systems architecture*, vol. 57, no. 5, pp. 487–497, 2011.

[18] U. C. Devi, "Soft real-time scheduling on multiprocessors," Ph.D. dissertation, University of North Carolina at Chapel Hill, 2006.

[19] A. Srinivasan and J. H. Anderson, "Efficient scheduling of soft real-time applications on multiprocessors." in *ECRTS*, 2003.

[20] U. C. Devi and J. H. Anderson, "Tardiness bounds under global edf scheduling on a multiprocessor," *Real-Time Systems*, vol. 38, no. 2, pp. 133–189, 2008.

[21] J. Erickson, U. Devi, and S. Baruah, "Improved tardiness bounds for global edf," in *ECRTS*, 2010.

[22] H. Leontyev and J. H. Anderson, "Generalized tardiness bounds for global multiprocessor scheduling," *Real-Time Systems*, vol. 44, no. 1-3, pp. 26–71, 2010.

[23] S. Kato and Y. Ishikawa, "Gang edf scheduling of parallel task systems," in *RTSS*, 2009.

[24] K. Lakshmanan, S. Kato, and R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *RTSS*, 2010.

[25] A. Saifullah, J. Li, K. Agrawal, C. Lu, and C. Gill, "Multi-core real-time scheduling for generalized parallel task models," *Real-Time Systems*, vol. 49, no. 4, pp. 404–435, 2013.

[26] J. Kim, H. Kim, K. Lakshmanan, and R. R. Rajkumar, "Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car," in *ICCPS*, 2013.

[27] G. Nelissen, V. Berten, J. Goossens, and D. Milojevic, "Techniques optimizing the number of processors to schedule multi-threaded tasks," in *ECRTS*, 2012.

[28] V. Bonifaci, A. Marchetti-Spaccamela, S. Stiller, and A. Wiese, "Feasibility analysis in the sporadic dag task model," in *ECRTS*, 2013.

[29] J. Li, K. Agrawal, C. Lu, and C. Gill, "Analysis of global edf for parallel tasks," in *ECRTS*, 2013.

[30] C. Liu and J. Anderson, "Supporting soft real-time parallel applications on multicore processors," in *RTCSA*, 2012.

[31] L. Nogueira and L. M. Pinho, "Server-based scheduling of parallel real-time tasks," in *EMSOFT*, 2012.

[32] D. Ferry, J. Li, M. Mahadevan, K. Agrawal, C. Gill, and C. Lu, "A real-time scheduling service for parallel tasks," in *RTAS*, 2013.

[33] Q. Wang and G. Parmer, "Fjos: Practical, predictable, and efficient system support for fork/join parallelism," in *RTAS*, 2014.

[34] J. Li, Z. Luo, D. Ferry, K. Agrawal, C. Gill, and C. Lu, "Global edf scheduling for parallel real-time tasks," *Real-Time Systems*, 2014.

[35] B. B. Brandenburg, "Scheduling and locking in multiprocessor real-time operating systems," Ph.D. dissertation, The University of North Carolina at Chapel Hill, 2011.

[36] F. Cerqueira, M. Vanga, and B. B. Brandenburg, "Scaling global scheduling with message passing," in *RTAS*, 2014, pp. 263–274.

[37] B. B. Brandenburg, J. M. Calandrino, and J. H. Anderson, "On the scalability of real-time scheduling algorithms on multicore platforms: A case study," in *RTSS*, 2008, pp. 157–169.